INRIA Lille Nord-Europe
Equipe-projet MOSTRARE
Encadrant : Joachim NIEHREN

Enumération des réponses aux requêtes Xpath conditionnelles avec variables

Antoine VENANT

Villeneuve d'Ascq
15 mai - 15 juillet 2009

Table des matières

1 Introduction

XML est un langage de balisage qui permet de structurer des données sous forme arborescente. Il est de plus en plus utilisé sur le web comme standard de stockage et d'échange de données. Il en va de même des technologies associées, notamment XQuery et XSLT, qui servent respectivement à extraire des informations d'un document XML (interroger une base de données XML par exemple) et à effectuer des transformations de document XML (souvent vers un format XHTML, pour les présenter sur une page web). Ces technologies reposent toutes deux sur le langage XPath, utilisé pour naviguer au sein d'un document XML *i.e* pour sélectionner certains noeuds de l'arbre formé par le document. Xpath est en effet un standard recommandé par le World Wilde Web ConsortiumW3C[W3C99]. L'évaluation de requêtes XPath apparaît donc comme un point central dans l'utilisation des technologies XML.

Plusieurs versions du langage existent, et de nombreux fragments en ont été étudiés. Deux versions ont été définies par le W3C : XPath1.0 et Xpath2.0 qui étend beaucoup les fonctionnalités de la première version en apportant deux nouveautés : d'une part il définit un nouvel opérateur élevant l'expressivité du langage au niveau de celle de la logique du premier ordre[1], d'autre part, il introduit des variables permettant l'écriture de requêtes n-aires, c'est à dire de sélectionner des n-upplets de noeuds en utilisant n symboles de variables, là où Xpath1.0 ne permettait l'écriture que de requêtes monadiques, c'est à dire de sélectionner un simple ensemble de noeuds satisfaisant un filtre.

Cependant, si Xpath1.0 peut être évalué très efficacement [GKP05], le nouvel opérateur introduit par Xpath2.0 ne permet pas une évaluation aussi efficace[2]. Mais il existe une autre extension de Xpath1.0, Conditional Xpath, définie par Maarten Marx dans [Mar05] qui conserve la même efficacité pour l'évaluation tout en étant aussi expressive que Xpath2.0.

En introduisant des variables dans ce dernier langage on permet l'écriture de requêtes n-aires de façon tout à fait analogue à Xpath2.0. Nous avons cherché à déterminer si Conditinal XPath avec variables pouvait également être évalué plus efficacement que XPath2.0 avec variables.

Par ailleurs, l'ensemble des solutions d'une requête n-aire est un ensemble de n-upplets de noeuds. Pour un document de taille D, sa taille peut donc s'élever jusqu'à D^n, ce qui peut s'avérer difficile à stocker si n est grand. C'est pourquoi nous avons également cherché à énumérer les réponses une à une sans les mémoriser, plutôt qu'à calculer d'un bloc l'ensemble des solutions.

Nous présentons ici un algorithme permettant d'énumérer les réponses d'une expression Conditional Xpath avec variables mais sans disjonction avec un délai linéaire combiné (taille du document multiplié par la taille de la requête). Nous étendons ensuite cet algorithme en un autre capable de calculer l'ensemble des solutions d'une expression Conditional Xpath avec variables (et disjonctions) plus efficacement que les algorithmes existants ne le permettaient pour Xpath 2.0, mais en abandonnant toutefois l'énumération. Nous en venons enfin à un algorithme d'énumération fonctionnant pour un fragment de Conditional XPath avec variables qui ne permet l'apparition que d'une unique disjonction, au sommet de la requête, mais reste complet pour la logique du premier ordre. Celui-ci énumére les solutions avec un délai linéaire, mais nécessite tout de même un précalcul quadratique.

2 Préliminaires

2.1 Modèle de document

Nous adoptons un modèle simplifié des documents XML définis par le W3C [W3C08] : Les documents que nous considérons sont arbre étiqueté d'arité non bornée dont nous donnons ici la définition :

[1] qui offre un autre formalisme d'écriture de requêtes

[2] Mais permet tout de même une évaluation polynômiale

Définition 1 (Arbre étiqueté d'arité non bornée). Soit Δ un alphabet d'étiquettes. Un arbre étiqueté d'arité non bornée est défini par induction : Il s'agit soit

- d'un symbole $a \in \Delta^*$
- d'un noeud représenté par un tuple d'arité non bornée $(a, t_1, t_2, ..., t_n)$ avec $\begin{cases} a \in \Delta^* \text{ (l'etiquette du noeud)} \\ t_1, t_2, ..., t_n \text{ sont des arbres étiquetés d'arité non bornée (les fils du noeud)} \end{cases}$

On définit également étant donné un arbre t, l'ensemble $Nodes(t)$ des noeuds d'un arbre t : un noeud n est un index formé par un mot de $(\mathbb{N} \setminus 0)^*$ identifiant de manière unique une occurrence d'un sous arbre s de t. s est le sous arbre de t « enraciné »par n, noté n^t. Intuitivement, la racine est indéxée ϵ, son premier fils 1, son deuxième fils 2, le premier fils du deuxième fils de la racine 2.1 est ainsi de suite. Les deux définitions sont inductives :

Définition 2 ($Nodes(t)$)**.**

$$Nodes(a) = \{\epsilon\},\ \epsilon^a = a$$

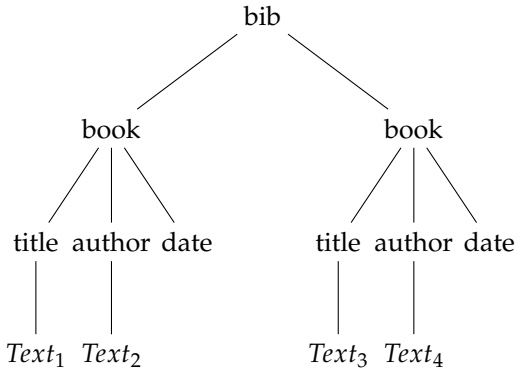
$$Nodes(t = (a, t_1, t_2, ..., t_n)) = \{\epsilon\} \bigcup_{k=0}^{n} k.Nodes(t_k),\ \begin{cases} \epsilon^t = t \\ \forall k \in [\![0; n]\!], \forall i \in Nodes(t_k)(k.i)^t = (i)^{t_k} \end{cases}$$

où l'ensemble $k.E$ est définit par $\{k.i \mid i \in E\}$ et où . désigne la concaténation pour les mots.

Définition 3 (Noeud étiqueté a**).** Soit $a \in \Delta^*$. Soit t un arbre étiqueté dans Δ^* on dit que $n \in Nodes(t)$ est étiqueté par a *ssi* $n^t = (a)$ ou $n^t = (a, t_1, \ldots, t_n)$.

La figure 1 montre un exemple de document XML.

FIG. 1 – Exemple de document



*Text*1 : Mémoires d'Outre Tombe
*Text*2 : Chateaubriand
*Text*3 : La Cantatrice Chauve
*Text*4 : Ionesco

$Nodes(t)$:

$Nodes(t)$	n^t
ϵ	(bib, book(title,author,date), book(title,author,date))
1	book(title,author,date)
2	book(title,author,date)
1.1	title
1.2	author
1.3	date
2.1	title
2.2	author
2.3	date

2.2 Ecriture de requêtes

Etant donné un document t correspondant au modèle défini précédemment, on souhaite écrire et répondre à des requêtes sélectionnant certains noeuds parmi $Nodes(t)$. Nous allons définir plus précisément la notion de requête, puis exposer différents formalismes en permettant l'écriture.

Définition 4 (requête n**-aire).** Une requête n-aire Q formulée dans l'alphabet Δ est une fonction qui à un arbre t étiqueté dans Δ^*, associe un sous-ensemble $Q(t)$ de $Nodes(t)^n$.
Cas particulier : Si $n = 1$ on parle de requête monadique.

On peut se servir de la logique de premier ordre pour formuler des requêtes :

2.2.1 Requêtes exprimées en logique du premier ordre

Axes et label-test Afin de pouvoir définir un langage de requêtes en logique du premier ordre, il faut définir l'ensemble des symboles de prédicats spécifiques à ce langage : On définit l'ensemble

$$L = \{Lab_x \mid x \in \Delta^*\}$$

des symboles de prédicats d'arité 1, que l'on désigne également par le terme *label-tests* ainsi que l'ensemble

$$R = \{\text{child}, \text{child}^*, \text{next\_sibling}, \text{next\_sibling}^*\}$$

des symboles de prédicats d'arité 2, également appelés *axes*.

Soit *Vars* un ensemble dénombrable de symboles de variables. On note Φ l'ensemble des formules que l'on peut construire en logique du premier ordre en utilisant les prédicats de T et R et les variables de *Vars*. Les formules de Φ sont exprimées à l'aide de la grammaire suivante :

$$\Phi := l(x) \mid r(x,y) \mid \neg\Phi \mid \exists x\Phi \mid \forall x\Phi \mid \Phi \wedge \Phi \mid \Phi \vee \Phi$$

où l, r, x et y sont des meta-variables respectivement dans L, R et *Vars*.

Afin de pouvoir donner une sémantique aux formules ainsi construites il est nécessaire d'en donner une aux différents symboles de prédicats que nous avons introduits. On a :

$$Lab_a(x) \textit{ ssi } x \text{ est étiqueté } a$$

Intuitivement, $\text{child}(x,y)$ identifie les couples père-fils dans l'arbre. Plus formellement :

$$\text{child}(x,y) \textit{ ssi } \exists i \in \mathbb{N}\backslash\{0\} \mid y = x.i.$$

child^* désigne la clôture réflexive transitive de child.

La sémantique de next_sibling désigne le premier voisin horizontal à droite. Plus formellement :

$$\text{next\_sibling}(x,y) \textit{ ssi } \text{ si x a pour index } i_1.i_2.\ldots.i_p \text{ alors y a pour index } i_1.i_2.\ldots.(i_p+1)$$

La relation next_sibling^* est définie par clôture réflexive transitive sur next_sibling de manière identique à child^* sur child.

Il reste à voir de quelle manière une formule définit une requête :

Définition 5 (Q_ϕ)**.** Soit $\phi \in \Phi$ une formule et $\bar{x} = (x_1, \ldots, x_n)$ une séquence de n variables contenant au moins les variables libres de ϕ, on peut associer à ϕ une requête n-aire, notée Q_ϕ définie de la façon suivante :

$$Q_\phi(t) = \{(\alpha(x_1), \ldots, \alpha(x_n)) \mid [\![\phi]\!]^{\alpha,t} = true, \alpha : Vars \to t\}$$

De telles requêtes ne peuvent cependant pas être évaluées efficacement : en effet, étant donnée une requête Q_ϕ et un arbre t, le problème consistant à déterminer si $Q_\phi(t) = \emptyset$ est PSPACE complet [Sto74]. D'où l'intérêt de considérer d'autres langages de requêtes.

2.2.2 Requêtes Xpath

Comme nous allons le voir, le langage Xpath est une autre façon d'adresser des noeuds au sein d'un document XML. Les langages définis par les textes du W3C possèdent une très large gamme de fonctionnalités pratiques n'intervenant pas systématiquement dans la navigation proprement

FIG. 2 – Sémantique d'une expression Xpath

$$
\begin{aligned}
[\![/PathExpr]\!]^t &= \{n \in Nodes(t) \mid (\text{root}, n) \in [\![PathExpr]\!]^t_{path}\} \\
[\![r :: l]\!]^t_{path} &= \{(n,n') \in Nodes(t)^2 \mid r(n,n'), l(n')\} \\
[\![PathExpr_1/PathExpr_2]\!]^t_{path} &= \{(n,n'') \in Nodes(t)^2 \\
&\quad \mid \exists n' \in t (n,n') \in [\![PathExpr_1]\!]^t_{path}, (n',n'') \in [\![PathExpr_2]\!]^t_{path}\} \\
[\![PathExpr_1 \cup PathExpr_2]\!]^t_{path} &= [\![PathExpr_1]\!]^t_{path} \cup [\![PathExpr_2]\!]^t_{path}\} \\
[\![PathExpr[FilterExpr]]\!]^t_{path} &= \{(n,n') \in [\![PathExpr]\!]^t_{path} / n' \in [\![FilterExpr]\!]^t_{filter}\} \\
[\![PathExpr]\!]^t_{filter} &= \{n \in Nodes(t) \mid \exists n' \in Nodes(t), (n,n') \in [\![PathExpr]\!]^t_{path}\} \\
[\![\text{not } FilterExpr]\!]^t_{filter} &= Nodes(t) \backslash [\![FilterExpr]\!]^t_{filter}
\end{aligned}
$$

dite au sein du document XML, c'est pourquoi il est coutume de simplifier le nombre de cas à envisager en ne s'intéressant qu'à des fragments logiques du langage. En conséquence nous nous restreignons ici à un fragment largement utilisé : **Navigational Xpath**(*NavXpath*) et à ses extensions. Reprenons l'ensemble T des *label-test* et l'ensemble R des *axes* et étendons les respectivement en $L_{Xpath} = L \cup \{*\}$ et $R_{Xpath} = R \cup \{\text{child}^{-1}, \text{child}^{*-1}, \text{next\_sibling}^{-1}, \text{next\_sibling}^{*-1}, \text{self}\}$, où le prédicat $*$ est vérifié par tous les noeuds d'un document, la sémantique de l'*axe* r^{-1} est exprimée à partir de celle de r précédemment définie :

$$r^{-1}(x,y) \; ssi \; r(y,x).$$

et où la relation self dénote simplement l'ensemble des couples $(x,x) \mid x \in Nodes(t)$.

Il y a deux niveaux d'expressions Xpath :
- les chemins, qui désignent un ensemble de noeuds accessibles depuis un noeud de départ. (que l'on peut voir comme des relations binaires sur les noeuds du document)
- les filtres, qui désignent un ensemble de noeuds vérifiant un prédicat. (que l'on peut voir comme une relation unaire sur les noeuds du document).

Définition 6 (**Requête Xpath**)**.** Une expression Xpath se construit selon la grammaire suivante :

$$
\begin{aligned}
Xpath &:= /PathExpr \\
PathExpr &:= r :: l \mid PathExpr/PathExpr \mid PathExpr[FilterExpr] \mid PathExpr \cup PathExpr \\
FilterExpr &:= PathExpr \mid \text{not } FilterExpr
\end{aligned}
$$

où r est un axe de R_{Xpath} et l un label-test de L_{Xpath}.

La figure 2 définit la sémantique d'une expression Xpath sur un document t.

Xpath1.0 ne peut donc exprimer que des requêtes monadiques. Par exemple, pour sélectionner tous les livres de Ionesco dans un document organisé comme celui de la figure 1 :

$$/\text{child}^* :: book[\text{child} :: author[\text{child} :: Lab_{Ionesco})]].$$

Xpath2.0 Xpath2.0 apporte deux choses :
- des variables.

– un opérateur de complémentation sur les chemins.

Etant donné un ensemble dénombrable de symboles de variables *Vars*, la syntaxe des chemins est modifiée comme suit :

$$\begin{aligned} PathExpr \quad := \quad & r :: l \mid PathExpr/PathExpr \mid PathExpr[FilterExpr] \mid PathExpr \cup PathExpr \\ & \mid \boldsymbol{PathExpr^{cpl}} \mid \$x \end{aligned}$$

avec $x \in Vars$.

La sémantique prend désormais en paramètre une valuation $\alpha : Vars \to Nodes(t)$ et comporte les deux ajouts suivants :

$$\begin{aligned} [\![\$x]\!]^{t,\alpha}_{path} &= Nodes(t) \times \{\alpha(x)\} \\ [\![PathExpr^{cpl}]\!]^{t,\alpha}_{path} &= Nodes(t)^2 \backslash [\![PathExpr]\!]^{t,\alpha}_{path} \end{aligned}$$

Une requête Xpath2.0 $expr$ à n variables libres $(x_1, \ldots, x_n)$ définit une requête n-aires qui sélectionne l'ensemble des n-upplets $\{(\alpha(x_1), \ldots, \alpha(x_n)) / [\![expr]\!]^{t,\alpha} \neq \emptyset\}$.

Conditional Xpath Nous présentons enfin une autre extension de Navigational XPath : Conditional XPath. Cette extension à été définie par M. Marx dans [Mar05] et modifie simplement la syntaxe en ajoutant un *axe conditionnel* :

$$\begin{aligned} PathExpr \quad := \quad & r :: l \mid \boldsymbol{(r[FilterExpr])^*} \mid PathExpr/PathExpr \mid PathExpr[FilterExpr] \\ & \mid PathExpr \cup PathExpr \end{aligned}$$

Intuitivement l'axe conditionnel correspond à une boucle tant que : on peut continuer à avancer dans le document suivant l'axe r tant que le filtre $FilterExpr$ est vérifié. Plus formellement, la sémantique $[\![(r[FilterExpr])^*]\!]^t_{path}$ est donnée par la clôture transitive de la relation binaire définie par $[\![r[FilterExpr]]\!]^t_{path}$. Ainsi on peut par exemple formuler une requête sélectionnant tous les noeuds x ayant un fils y tel que le mot formé par les étiquettes des noeuds du chemin de x à y est dans a^*b :

$$\text{child}^* :: *[(\text{child}[\text{self} :: a])^* / \text{child} :: b].$$

2.3 Expressivité

Toutes les requêtes formulables en logique du premier ordre peuvent elles être traduites en requêtes Xpath ?

Théorème 1 ([Mar05])**.** [Xpath 1.0] Il existe des requêtes monadiques exprimées à l'aide de formule logique à une variable libre qui ne peuvent pas être traduites en Xpath 1.0.

Théorème 2 (Xpath 2.0)**.** Navigational Xpath 2.0 est aussi expressif que la logique du premier ordre *i.e* toute requête $n - aire$ exprimée à l'aide d'une formule logique peut aussi être exprimée avec Navigational Xpath 2.0.

Théorème 3 ([Mar05][Conditional Xpath)**.** Toute requête définie par une formule logique à une variable libre peut être traduite en une requête Conditional Xpath et toute requête Conditional XPath peut être réciproquement traduite en une formule logique à une variable libre.

3 Evaluation de Xpath : état de l'art

Le problème qui nous interesse est celui de l'efficacité de l'évaluation des requêtes Xpath. Nous présentons d'abord briévement les résultats et techniques connus.

3.1 Xpath1.0 et Conditional XPath : Algorithme de Gottlob et Koch

Théorème 4 ([GKP05, Mar05])**.** Navigational Xpath et Conditional XPath peuvent tous deux être évalués en temps linéaire combiné *i.e* pour une requête Q définie par une expression $/PathExpr$ sur un document t, en temps $O(|t||PathExpr|)$ grâce à l'algorithme de Gottlob et Koch présenté ci-après.

Evaluation des axes

Proposition 1. On peut évaluer tous les axes et l'axe conditionnel en temps $O(|t|)$ *i.e*

- étant donnés un ensemble Π de noeuds de départ, une expression *location-step* $r :: l$ et un document t, on peut calculer $[\![r :: l]\!]^t_{path} \cap (\Pi \times Nodes(t))$ en temps $O(|t|)$.

- étant donnés un ensemble Π de noeuds de départ, une expression conditionnelle $(r[FilterExpr])^*$ et l'évaluation de l'expression de filtre $[\![FilterExpr]\!]^t_{filter}$, on peut caculer $[\![(r[FilterExpr])^*)]\!]^t_{path} \cap (\Pi \times Nodes(t))$ en temps $O(|t|)$.

Le principe consiste à « balayer »l'arbre suivant l'axe considéré, en retenant les noeuds sélectionnés au fil du balayage. Par exemple si l'axe à évaluer est child*, on parcourt parallèlement en profondeur toutes les branches de l'arbre et lorsqu'on passe sur un noeud de Π, on le collecte et on passe un drapeau à 1, indiquant que tous les noeuds visités par la suite sur la branche doivent être collectés.

Si l'axe est next_sibling, on procède de même avec un parcours en largeur de gauche à droite.

L'axe conditionnel ne pose pas plus de problème si on connaît l'ensemble $[\![FilterExpr]\!]^t_{filter}$: on procède de même mais on ne collecte un noeud que lorsque le drapeau est à 1 **et** que le noeud est dans $[\![FilterExpr]\!]^t_{filter}$[3]. Par ailleurs, dès que ce n'est pas le cas, on repasse le drapeau à 0.

Pour chaque axe r on peut donc définir deux fonctions calculables en $O(|t|)$:

- $\tilde{r} : 2^{Nodes(t)} \to 2^{Nodes(t)}$ qui à un ensemble de noeuds Π et un ensemble de noeuds « filtrant »F associe l'ensemble $r^t(\Pi)$ des noeuds accessibles depuis un noeud de Π.
- $\tilde{r}_{cond} : 2^{Nodes(t)} \times 2^{Nodes(t)} \to 2^{Nodes(t)}$ qui à un ensemble de noeuds Π et un ensemble de noeuds « filtrants »F associe l'ensemble $r^t(\Pi, F)$ des noeuds accessibles depuis un noeud de Π par la relation $(r[F])^*$.

Ces deux fonctions servent de cas de base à l'algorithme qui suit.

Algorithme de Gottlob et Koch L'algorithme s'appuie sur deux fonctions récurrentes[4] prenant en entrée un ensemble de noeuds de départ et renvoyant en sortie un ensemble de noeuds d'arrivée :

- la fonction $eval_\downarrow$ évalue les chemins par un parcours Top-down de la requête : $eval_\downarrow(\Pi, PathExpr)$ est l'ensemble des noeuds accessibles depuis un noeud de Π par la relation que définit $[\![PathExpr]\!]^t_{path}$.
- la fonction $eval_\uparrow$ évalue les filtres par un parcours Bottom-up de la requête : $eval_\uparrow(\Pi, PathExpr)$ est l'ensemble des noeuds tel que tout noeud de cet ensemble mène à au moins un noeud de Π par la relation que définit $[\![PathExpr]\!]^t_{path}$.

La figure 3 définit explicitement ces deux fonctions. Une induction simple montre que $eval_\downarrow(\{root\}, PathExpr)$ renvoie l'ensemble des noeuds sélectionnés par la requête Q définie par l'expression $/PathExpr$ en temps $O(|t||PathExpr|)$.

[3] Le test peut être éffectué en temps $O(|1|)$ avec une structure de données adaptée.

[4] La première (évaluation ascendante) fait appel à la seconde (évaluation descendante), mais la seconde n'appelle pas la première.

FIG. 3 – calcul de $eval_{\downarrow}$ et $eval_{\uparrow}$

$$
\begin{aligned}
eval_{\downarrow}(\Pi, r :: l) &= \tilde{r}(\Pi) \cap Lab_l(t) \\
eval_{\downarrow}(\Pi, (r[FilterExpr])^*) &= \tilde{r}_{cond}(\Pi, eval_{\uparrow}(Nodes(t), FilterExpr)) \\
eval_{\downarrow}(\Pi, PathExpr_1 / PathExpr_2) &= eval_{\downarrow}(eval_{\downarrow}(\Pi, PathExpr_1), PathExpr_2) \\
eval_{\downarrow}(\Pi, PathExpr[FilterExpr]) &= eval_{\downarrow}(\Pi, PathExpr) \cap eval_{\uparrow}(Nodes(t), FilterExpr) \\
eval_{\downarrow}(\Pi, PathExpr_1 \cup PathExpr_2) &= eval_{\downarrow}(\Pi, PathExpr_1) \cup eval_{\downarrow}(\Pi, PathExpr_2)
\end{aligned}
$$

$$
\begin{aligned}
eval_{\uparrow}(\Pi, r :: l) &= \tilde{r}^{-1}(\Pi \cap Lab_l(t)) \\
eval_{\uparrow}(\Pi; (r[FilterExpr])^*) &= \tilde{r}^{-1}_{cond}(\Pi, eval_{\uparrow}(Nodes(t), F)) \\
eval_{\uparrow}(\Pi, PathExpr_1 / PathExpr_2) &= eval_{\uparrow}(eval_{\uparrow}(\Pi, PathExpr_2), PathExpr_1) \\
eval_{\uparrow}(\Pi, PathExpr[FilterExpr]) &= eval_{\uparrow}(\Pi \cap eval_{\uparrow}(Nodes(t), FilterExpr), PathExpr) \\
eval_{\uparrow}(\Pi, PathExpr_1 \cup PathExpr_2) &= eval_{\uparrow}(\Pi, PathExpr_1) \cup eval_{\uparrow}(\Pi, PathExpr_2)
\end{aligned}
$$

3.2 Xpath2.0

3.2.1 Restriction sur les variables

Si on n'impose pas de restriction sur les variables, les requêtes en logique du premier ordre peuvent êtres traduites en temps linéaire en requête Xpath2.0 [FNTT07] ce qui rend ces dernières intraitables efficacement.

Théorème 5 ([FNTT07])**.** Avec les restrictions suivantes,
- pas de partage de variable dans les compositions de chemin (exp_1/exp_2 autorisé si $Vars(exp_1) \cap Vars(exp_2) = \emptyset$).
- pas de variable derrière les négations où dans les complémentations.
- pas de partage de variables lors d'un filtrage ($exp[f]$ autorisé si $Vars(exp) \cap Vars(f) = \emptyset$.

Navigational Xpath 2.0 avec ces restrictions reste complet pour la logique du premier ordre et peut être évalué en temps $O(|t|^3|PathExpr||A|)$ où A désigne l'ensemble des solutions.

4 Evaluation de Conditional Xpath avec variables

Nous nous intéressons maintenant à l'évaluation du langage Conditional XPath avec variables : les variables sont introduites de la même manière que dans XPath 2.0 et avec les restrictions présentées ci-avant. Etant donné que les noeuds sont sélectionnés par des variables, il n'est plus utile de conserver deux niveaux d'expression filtres et chemins, on peut se contenter du niveau de filtre, et remplacer les compositions de chemins par des compositions de filtres : Par exemple la requête suivante qui sélectionne tous les couples de noeuds (x, y) tels que les étiquettes des noeuds du chemin de x à y définit un mot de a^*b

$$/\text{child} :: */\$x/(\text{child}[\text{self} :: a])^*/\text{child} :: b/\$y$$

peut être réécrite :

$$/[\text{child} :: *[\$x[(\text{child}[\text{self} :: a])^*[\text{child} :: b[\$y]]]]]$$

avec des compositions de filtres.

Les requêtes Conditional Xpath avec variables que nous considérons sont écrites dans la grammaire suivante :

$$\begin{aligned} CondXpathExpr &= /[FilterExpr] \\ FilterExpr &= r :: l \mid (r[FilterExpr])^* \mid \text{not } FilterExpr \mid FilterExpr[FilterExpr] \\ &\quad \mid FilterExpr \text{ or } FilterExpr \mid \$x \end{aligned}$$

avec $r \in R_{Xpath}$, $l \in L_{Xpath}$, $x \in Vars$, et les restrictions suivantes :

- pas de variables dans une expression $(r[FilterExpr])^*$.
- pas de partage de variables dans une expression $FilterExpr_1[FilterExpr_2]$ *i.e* $Vars(FilterExpr_1) \cap Vars(FilterExpr_2) = \emptyset$.

à une expression $/[FilterExpr]$ et une séquence $(x_1, \ldots, x_n)$ de variables contenant au moins les variables de l'expression on associe une requête Q qui à un arbre t associe l'ensemble des solutions $Q(t) = \{\alpha(x_1), \ldots, \alpha(x_n) \mid root \in [\![FilterExpr]\!]^{\alpha}_{filter}, \alpha : Vars \to Nodes(t)\}$

Nous allons montrer dans cette section que l'on peut calculer l'ensemble des solutions d'une expression Conditional XPath avec variables plus efficacement que celui d'une expression Navigational XPath 2.0 :

Théorème 6. Etant donnée une requête Conditional Xpath avec variables Q définie par une expression $CondXpathExpr$ et un document t, on peut calculer l'ensemble $A = Q(t)$ des réponses en temps $O(|t||CondXpathExpr||A|log|A|)$.

Nous présentons maintenant un algorithme qui permet un tel traitement.

Nous définissons d'abord un pré-traitement qui permet de retenir un ensemble de noeuds « candidats »pour chaque occurrence d'une sous-expression de la requête[5].

4.1 Précalcul

On cherche à faire correspondre à chaque occurrence exp d'une sous-expression de la requête un ensemble E_{exp} de noeuds du document tel que si on impose une valeur parmi cet ensemble pour cette position, on peut compléter cette affectation en un chemin satisfaisant le filtre complet.

L'algorithme de Gottlob et Koch peut nous y aider. On constate par exemple qu'en remplaçant chaque symbole de variable par un axe self, puis en appliquant l'algorithme de Gottlob et Koch pour évaluer la requête comme un filtre, on peut déterminer s'il existe une solution : il faut et il suffit pour celà que l'ensemble obtenu contienne la racine de l'arbre. En s'inspirant de cette idée, on peut effectuer le précalcul à l'aide un double passage avec l'algorithme de Gottlob et Koch afin de sélectionner les « bons »candidats.

Considérons la requête suivante :

$$/\text{child}^* :: *[\$x[\text{child} :: *[\$y[\text{child} :: *[\$z]]]]]$$

évaluée sur le document de la figure 4 (où les noeuds sont étiquetés par leur index).

et construisons son arbre syntaxique : celui-ci est présenté par la figure 5.

Un appel à l'algorithme de Gottlob et Koch évalue la requête par un parcours ascendant : il construit d'abord $E_{exp_{11}} = Nodes(t)$ puis $E_{exp_{10}} = \{(1, (1.1), (1.2)\}$ l'ensemble des noeuds qui ont un fils, grâce à l'appel $eval_{\uparrow}(\text{child} :: *, E_{exp_{11}})$, qui retourne l'ensemble des pères des noeuds de $E_{exp_{11}}$. Il calcule ensuite $E_{exp_7} = E_{exp_8} = E_{exp_9} = E_{exp_{10}}$ (symbole $\$y$ emplacé par self) puis $E_{exp_6} = eval_{\uparrow}(\text{child} :: *, E_{exp_7}) = \{(1)\}$ et ainsi de suite en remontant dans l'arbre. On voit sur cet exemple que le noeud (1) n'est pas utile dans $E_{exp_{10}}$: si on affecte ce noeud à exp_{10}, on ne peut pas lui trouver de père, et on ne peut pas affecter de noeud à exp_6. Pour ne garder que des neuds « valides »pour chaque occurrence de sous-expression, on effectue le même chemin

[5] *i.e* pour chacun des noeuds de l'arbre syntaxique de la requête

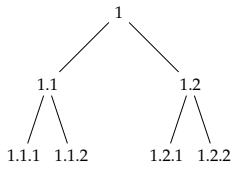


FIG. 4 – document exemple

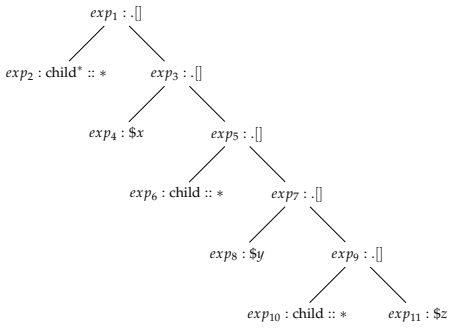


FIG. 5 – arbre syntaxique de la requête

« à l'envers », en évaluant les axes en descendant cette fois. Dans notre exemple, si on repart de E_{exp_6}, puis on fait un passage descendant suivant l'axe child *i.e* on prend l'ensemble des fils de E_{exp_6} et on remplace E_{exp_7} par son intersection avec cet ensemble, on retire bien le noeud (1) de E_{exp_7}. L'algorithme présenté en annexe (section 7.1) généralise ce procédé.

4.2 Enumération suivant un chemin « simple »

Considérons une expression Conditional Xpath conforme au restrictions suivantes (nous étudierons par la suite chacun des cas interdits ici) :

- On n'autorise que les sous-expressions $FilterExpr_1[FilterExpr_2]$ qui sont de la forme $\$x[FilterExpr]$ où qui ont un côté sans variable *i.e* $Vars(FilterExpr_1) = \emptyset$ ou $Vars(FilterExpr_2) = \emptyset$.
- Il n'y a pas de disjonction, *i.e* aucune sous-expression de la forme $FilterExpr$ or $FilterExpr$.

Théorème 7. Une fois effectué le précalcul précédent on peut définir un algorithme de backtrack qui énumére les solutions d'une telle requête avec un délai $O(|CondXpathExpr||t|)$ entre chaque solution.

Le principe est d'effectuer un parcours en profondeur de l'arbre syntaxique de la requête au cours duquel chaque position dans l'expression correspondant à un symbole de variable $exp : \$x$ définit un point de backtrack.

On procède de la manière suivante :

- Si le point de backtrack courant est exp on retire la première valeur π_1 de E_{exp} si celui-ci n'est pas vide. Sinon on retourne au point de Backtrack précédent.
- On fait un passage en descendant depuis exp à l'aide de l'algorithme de Gottlob et Koch avec comme ensemble de noeuds de départ $\{\pi_1\}$. Ce passage ne peut pas échouer car π_1 appartient à un ensemble établi par le précalcul précédent.
- On s'arrête lorsqu'on atteint exp' le point suivant de backtrack. On a alors calculé l'ensemble Π des noeuds affectables à exp' lorsque π_1 est affecté à exp. On remplace $E_{exp'}$ par $E_{exp'} \cap \Pi$ et exp' devient le point de backtrack courant.
- Si il n'y a plus de point de backtrack après exp l'ensemble des noeuds sélectionnés pour chaque point de backtrack donne une affectation qui est solution de la requête.

La requête de l'exemple précédent satisfait aux restrictions : Le premier point de backtrack est exp_4 avec l'ensemble de noeuds précalculés $E_{exp_4} = \{(1)\}$. Un passage descendant avec l'algorithme de Gottlob et Koch suivant l'axe child permet d'arriver au deuxième point de backtrack exp_8 avec l'ensemble de valeurs possibles $\{(1.1), (1.2)\}$ ici égal à l'ensemble précalculé. On choisit donc (1.1) puis on fait un deuxième passage suivant l'axe child pour arriver à exp_{11} le troisième et dernier point de backtrack, avec deux valeurs possibles $(1.1.1)$ et $(1.1.2)$ qui sont dans $E_{exp_{11}}$, ce qui nous donne deux solutions : $x = (1), y = (1.1), z = (1.1.1)$ et $(x = (1), y = (1.1), z = (1.1.2)$.

On retourne ensuite au deuxième point de backtrack, on choisit (1.2) et on obtient de la même façon deux autres solutions : $x = (1), y = (1.2), z = (1.2.1)$ et $(x = (1), y = (1.2), z = (1.2.2)$. On a bien obtenu toutes les solutions.

4.3 Traitement des conjonctions

Considérons toujours le cas sans disjonction, mais oublions l'autre restriction et observons la requête suivante :

$$/\text{child}^* :: *[\$x[(\text{child}^* :: *[\text{child}^* :: *[\$y]])[\text{child}^* :: *[\$z]]]]$$

Problème La sous-expression $(child^* :: *[\text{child}^*[\$y]])[\text{child}^* :: *[\$z]]$ ne vérifie pas la deuxième restriction. Voyons en quoi celà pose problème : La requête exprime une conjonction, elle demande que x ait un descendant π tel que y soit un descendant de π **et** que z soit également un descendant de π. Le problème empéchant d'appliquer simplement le backtrack est précisément ce noeud intermédiaire π : Il faut s'assurer que ce soit bien le même noeud qui soit assigné à π lorsqu'on « descend »de x et à y et de x à z.

Solution On pourrait considérer la position à laquelle se trouve la sous-expression problématique $(child^* :: *[\text{child}[\$y]])[\text{child}^* :: *[\$z]]$ comme un point de backtrack mais cela ne suffirait pas : aucune variable n'est affectée à cette expression, autrement dit deux valeurs différentes affectées à cette position peuvent mener à une même affectation globale des variables. On gonflerait donc ainsi artificiellement l'ensemble des solutions et donc la complexité du traitement. Dans l'exemple ci-dessus, on pourrait par exemple avoir comme solution $(x = (1), [\pi = (1.1)], y = (1.1.1), z = (1.1.2))$ et la même, avec une autre affectation de π :$(x = (1), [\pi = (1.1.1)], y = (1.1.1), z = (1.1.2))$.

La solution adoptée consiste à énumérer récursivement à droite, puis à revenir en arrière pour énumérer à gauche :

Les expressions posant problème sont les positions $conj : exp_1[exp_2]$ associées à des expressions qui ne sont pas de la forme $\$x[FilterExpr]$ dans lesquelles :

- une position b_1 dans exp_1 est un point de backtrack.
- une position b_2 de exp_2 est un point de backtrack.

Supposons que l'on sache énumérer récursivement toutes les solutions pour la position exp_1 à partir d'un ensemble Π de noeuds affectables à exp_1, et faisons la même hypothèse sur exp_2. On utilise la technique suivante pour déterminer l'ensemble des solutions de $conj$ à partir de l'ensemble E_{conj} de noeuds possibles pour $conj$:

1. On énumére récursivement une solution α pour exp_1 **à partir de l'ensemble** E_{conj}
2. Cette solution pose une contrainte sur les points de backtrack de exp_1. **On fait un parcours descendant de** $conj$ **à l'aide de l'algorithme de Gottlob et Koch en imposant le respect de ces contraintes, pour déterminer l'ensemble** $E^{\alpha}_{f_2}$ **des noeuds de** E_{f_2} **dont l'affectation à** f_2 **permet la solution** α **pour** exp_1.
3. On énumére les solutions pour exp_2 à partir de $E^{\alpha}_{f_2}$. Le point important est que chacune de ces solutions peut être obtenue à partir d'un noeud de $E^{\alpha}_{f_2}$, or par construction de cet ensemble, le même noeud mène à la solution α dans exp_1. Si on concatène les deux solutions partielles, on obtient donc bien une solution globale valide !

4.4 Traitement des disjonctions

Les positions $disj : exp_1$ or exp_2 correspondant à des disjonctions posent plus de problèmes car elles peuvent partager des variables. Il est donc possible que la même solution apparaisse

de chaque côté de la disjonction et si on se contente d'énumérer récursivement exp_1 et exp_2 en parallèle, chaque étape n'apporte pas forcément une solution nouvelle. Nous renonçons en fait ici à énumérer les solutions et nous allons présenter une façon de calculer d'un bloc l'ensemble des solutions. La prochaine section sera consacrée à l'énumération avec disjonction.

Considérons une expression Conditional Xpath avec variables.

Le point empêchant l'application de l'algorithme présenté auparavant est que l'on peut rencontrer une expression exp_1 or exp_2 lors d'une phase d'exploration depuis un point de Backtrack. Dans ce cas on procède de la façon suivante : On choisit d'abord de continuer suivant exp_1 et on énumére récursivement toutes les solutions sur ce chemin. On obtient un ensemble A_1 de solutions. On procède ensuite de même en choisissant exp_2. On obtient un ensemble A_2 de solutions, et on calcule enfin $A_1 \cup A_2$, et on peut retourner au point de backtrack de départ pour continuer le calcul.

4.5 Complexité

L'algorithme précédent calcule l'ensemble des réponses à une requête en temps $O(|t||CondXpathExpr|||A|log(|A|))$, sous réserve d'utiliser une bonne structure de données pour effectuer les unions.

Une version plus formelle et détaillée de l'algorithme et une preuve de complexité sont fournies en annexe.

Cet algorithme est plus efficace que ceux existants pour Navigational Xpath 2.0. Le traitement des disjonctions apporte cependant un aspect décevant par rapport aux cas précédents : la dernière modification de l'algorithme nous contraint à garder en mémoire l'ensemble A des solutions, dont la taille peut grimper jusqu'à t^n où n est le nombre de variables.C'est pourquoi on cherche à trouver un algorithme qui permet l'énumération sur des requêtes avec disjonctions.

5 Enumération avec disjonction

5.1 Idée

Le problème des disjonctions est l'apparition de la même solution de chaque côté de la disjonction, qui mène à des énumérations « inutiles »ou oblige à se souvenir de ce que l'on a déjà énuméré. Nous nous restreignons dans la suite à un fragment du langage qui reste complet pour la logique du premier ordre, dans lequel on n'autorise qu'une seule disjonction d'un nombre quelconque de cas, placée au sommet de la requête. On peut en fait transformer toute requête Conditional Xpath en une requête de ce type, mais celà peut faire exploser la taille de la requête.

La grammaire utilisée est la suivante :

$$\begin{aligned} CondXpathExpr &= /FilterExpr | CondXpathExpr \text{ or } CondXpathExpr \\ FilterExpr &= r :: l \mid (r[FilterExpr])^* \mid FilterExpr[FilterExpr] \mid \text{not } FilterExpr \mid \$x \end{aligned}$$

Proposition 2. Soit une expression exp_1 or exp_2 et supposons que l'on sache énumérer les solutions de exp_1 et exp_2 séparément **et que l'on connaisse le cardinal des deux ensembles de solutions**. Supposons par exemple que exp_1 ait plus de solutions que exp_2, alors on peut énumérer la disjonction de la manière suivante :

1. On énumére une solution de exp_1 et une solution de exp_2.
2. On teste si la solution énumérée pour exp_2 est solution pour exp_1[6]. Si oui on ne la sort pas, elle sera énumérée plus tard, sinon on la sort également.
3. On retourne à l'étape 1

[6]peut se faite en un passage sur la requête à l'aide de l'algorithme de Gottlob et Koch légérement modifié

4. Si on a fini d'énumérer exp_2 (ce qui se produit avant d'avoir terminé exp_1, grâce à l'hypothèse de cardinalité) il suffit de finir d'énumérer exp_1 : toutes les solutions seront nouvelles puisque les solutions en commun avec exp_2 n'ont pas étés énumérées !

Ce qui nous manque pour appliquer une telle méthode est le calcul du nombre de solutions. Nous allons présenter une méthode permettant de le réaliser.

5.2 Calcul du cardinal de l'ensemble des solutions d'un chemin « simple »

l'idée est de recourir à un précalcul un peu plus complexe qui dresse des tables entre deux points de backtrack successifs : Reprenons notre exemple de chemin « simple » :

$$/\text{child}^* :: *[\$x[\text{child} :: *[\$y[\text{child} :: *[\$z]]]]]$$

Les points de backtrack sont les suivants : $exp4 : \$x$, $exp_8 : \$y$, $exp_{11} : \$z$. On peut grâce à un parcours avec l'algorithme de Gottlob et Koch associer à chaque valeur possible pour exp_4 l'ensemble des valeurs possibles pour exp_8. De même pour exp_8 et exp_{11}. On obtient les tables suivantes :

$exp_4 : \$x$	$exp_8 : \$y$
(1)	{(1.1); (1.2)}

et

$exp_8 : \$y$	$exp_{11} : \$z$
(1.1)	{(1.1.1); (1..1.2)}
(1.2)	{(1.2.1); (1..2.2)}

On réécrit donc notre requête sous la forme d'une succession de tables reliant les points de bactrack entre eux : $Q : T_{exp_4 \to exp_8} \wedge T_{exp_8 \to exp_{11}}$.

On peut alors définir un algorithme récursif qui à chaque entrée d'une table associe le nombre de solutions auquel peut mener l'affectation qu'elle représente pour les points de backtrack situés plus bas dans la requête : Il suffit de sommer récursivement les nombres de solutions pour chaque noeud de l'ensemble associé à l'entrée.

Illustrons ce procédé sur notre exemple :

$exp_4 : \$x$	$exp_8 : \$y$	nb_sol
(1)	{(1.1); (1.2)}	nb_sol(1.1) + nb_sol(1.2)

et

$exp_8 : \$y$	$exp_{11} : \$z$	nb_sol
(1.1)	{(1.1.1); (1..1.2)}	2
(1.2)	{(1.2.1); (1..2.2)}	2

On a donc nb_sol($(1) \leftarrow exp_4$)= 2+2 =4 ce qui est aussi le nombre de solutions à la requête et correspond bien à ce qui avait été trouvé précédemment.

Pour un point de backtrack donné, il peut y avoir jusqu'à $Nodes(t)$ valeurs possibles. Il faut donc éffectuer $Nodes(t)$ passages de ce point au point de backtrack suivant avec l'algorithme de Gottlob et Koch pour dresser la première table, puis il faut dresser récursivement les suivantes. Le coût du précalcul est donc $O(|t| \times |t||CondXpathExpr|)$ *i.e* $O(|t|^2|CondXpathExpr|)$.

5.3 Traitement des conjonctions

Si on permet les expressions de la forme $conj : exp_1[exp_2]$ différentes de $\$x[exp_2]$ avec $Vars(exp_1) \neq \emptyset$ et $Vars(exp_2) \neq \emptyset$, Il survient le même problème que dans la sous-section 4.3 : La position $conj$ correspond à une expression problèmatique qui mène à deux points de backtrack : b_1 dans exp_1 et b_2 dans exp_2.

Pour traiter ce genre d'expressions, on adapte l'algorithme précédent à l'aide de la même astuce que nous avons employée pour l'énumération avec conjonctions dans la section précédente :

Supposons que l'on puisse récursivement pour chaque noeud π de E_{b_1} calculer le nombre de solutions pour b_1 auxquelles mène l'affectation de π à b_1, et faisons la même hypothèse pour b_2.

Appelons b_0 le point de backtrack qui précède cette expression. Soit π_{b_0} un noeud de E_{b_0}. On peut calculer le nombre de solutions auxquelles mène l'affectation de π_{b_0} à b_0 de la façon suivante :

1. On fait un passage descendant avec l'algorithme de Gottlob et Koch entre b_0 et b_1 depuis l'ensemble de valeurs de départ $\{\pi_{b_0}\}$, et on en profite pour retenir l'ensemble $E_{conj}^{\pi_{b_0}}$ des noeuds possibles pour $conj$, et l'ensemble $E_{b_1}^{\pi_{b_0}}$ des noeuds possibles pour b_1.
2. Pour chaque noeud $\pi_{b_1} \in E_{b_1}^{\pi_{b_0}}$ on fait un retour ascendant avec l'algorithme de Gottlob et Koch entre b_1 et $conj$ depuis l'ensemble de valeurs $\{\pi_{b_1}\}$, on retient au passage l'ensemble $E_{conj}^{\pi_{b_1}}$ des noeuds de E_{conj} rendant possible l'affectation $b \leftarrow \pi_{b_1}$. On prolonge ce passage en un passage descendant entre $conj$ et b_2 depuis l'ensemble $E_{conj}^{\pi_{b_1}} \cap E_{b_1}^{\pi_{b_0}}$, et on calcule ainsi l'ensemble $E^{\pi_{b_0},\pi_{b_1}}$ des noeuds de b_2 des noeuds E_{b_2} **tels qu'il existe un noeud possible pour** $conj$**, permis par l'affectation de** π_{b_0} **à** b_0 **et permettant à la fois l'affectation de ce noeud à** b_2 **et celle de** π_1 **à** b_1
3. Pour chaque noeud π_{b_2} de E_{b_2}, on fait nb_sol(π_0)+=nb_sol(π_1)×nb_sol(π_2).

6 Conclusion

Les résultats obtenus confirment Conditional Xpath comme une extension naturelle de Navigational Xpath : l'algorithme d'évaluation de Conditional Xpath avec variables présenté ici montre que la possibilité d'évaluer Conditional Xpath aussi facilement que Navigational Xpath[7] et donc plus facilement que Xpath 2.0[8] s'étend au cas avec variables. En effet Conditional Xpath avec variables peut être évalué en temps $O(|t||CondXpathExpr||A|log(|A|))$ avec précalcul en $O(|t||CondXpathExpr|)$ tandis que les algorithmes existants évaluent Xpath 2.0 en $O(|t^3||PathExpr||A|)$. Toutefois l'introduction des variables soulève également quelques problèmes : L'algorithme d'évaluation du langage complet parait un peu décevant vis à vis du fragment sans disjonction que nous avons d'abord introduit : On peut dans ce cas, non seulement évaluer, mais aussi énumérer les réponses une à une, avec un délai $O(|t||CondXpathExpr|)$ entre chaque réponse. Il s'ensuit que l'énumération de toutes les réponses dans ce cas nécessite un temps $O(|t||CondXpathExpr||A|)$, meilleur que ce que nous avons pû produire avec les disjonctions. Par ailleurs, nous avons tout de même pu proposer un algorithme d'énumération pour un fragment complet pour la logique du premier ordre (avec une seule disjonction placée au sommet de la requête), mais cet algorithme nécessite un précalcul quadratique, et le fragment considéré est assez restrictif vis à vis du langage complet. Il reste donc à déterminer si notre algorithme d'énumération pourrait être étendu au cas général, autorisant l'imbrication des disjonctions. Enfin, les disjonctions posent beaucoup de problèmes pour les langages avec variables, car elles introduisent une forme de non-déterminisme : une même solution peut se retrouver sélectionnée par les deux termes d'une disjonction, et si l'on n'élimine pas ces doublettes on risque souvent de faire enfler inutilement la compléxité du traitement. Une idée pourrait être d'essayer de restreindre les schémas d'arbres permis (comme interdire la récursivité dans les étiquettes) pour limiter les possibilités d'apparitions de doublettes.

[7] $O(|t||PathExpr|)$ avec l'algorithme de Gottlob et Koch

[8] $O(|t^3||PathExpr|$ l'opérateur de complémentation induit une évaluation au moins aussi coûteuse qu'un produit de matrice $|t| \times |t|$;

7 Annexes

7.1 Algorithme de précalcul

Procedure Precalcul(Q)
Precalcul_Ascendant(Q,$Nodes(t)$)
Precalcul_Descendant(Q,$E_Q \cap \{root\}$)

Procedure Precalcul_Ascendant(Q,Π)
Match Q **With**
| r : :l $\Rightarrow$
$E_Q \leftarrow \tilde{r}^{-1}(\Pi \cap l(t))$

| $(r[f])^* \Rightarrow$
Precalcul_Ascendant(f,Π)
$E_Q \leftarrow \tilde{r}^{-1}_{cond}(\Pi, E_f)$

| $(f_1[f_2] \Rightarrow$
Precalcul_Ascendant(f_2,$Nodes(t)$)
Precalcul_Ascendant(f_1,$E_{f_2} \cap \Pi$)
$E_Q \leftarrow E_{f_1}$

| $f_1 or f_2 \Rightarrow$
Precalcul_Ascendant(f_1,Π)
Precalcul_Ascendant(f_2,Π)
$E_Q \leftarrow E_{f_1} \cup E_{f_2}$

| $\$x \Rightarrow$
$E_Q \leftarrow \Pi$

Function Precalcul_Descendant(Q,Π)
$E_Q \leftarrow E_Q \cap \Pi$
Match Q **With**
| $r :: l \Rightarrow$
RETURN $r(E_Q) \cap l(t)$

| $(r[f])^* \Rightarrow$
RETURN $r_{cond}(E_Q, E_f)$

| $(f_1[f_2] \Rightarrow$
let $\Pi' =$ Precalcul_Descendant(f_1, Π)
Precalcul_Descendant(f_2,Π')
RETURN Π'

| $f_1 or f_2 \Rightarrow$
RETURN Precalcul_Descendant(f_1,Π) $\cup$
Precalcul_Descendant(f_2,Π)

| $\$x \Rightarrow$
RETURN E_Q

7.2 Algorithme de calcul des solutions pour Conditional XPath avec variables

Cet algorithme nécessite une fonction auxiliaire **ensure_constraints(α,Q)** prenant en paramètre une requête Q et une affectation $\alpha : Vars(Q) \rightarrow Nodes(t)$, et renvoyant l'évaluation de la requête Q considérée comme une requête Conditional Xpath sans variables : les symbôles de variables $\$x$ sont interprétés comme des filtres sélectionnant l'ensemble de noeuds $\alpha(x)$. La fonction renvoie en fait $[\![Q]\!]^{t,\alpha}$ et peut être très simplement mise en oeuvre par une légère adaptation des fonctions $eval_\downarrow$ et $eval_\uparrow$.

7.2.1 Pseudo-code

Function Eval(Q)
Precalcul(Q)
RETURN compute_all_answers(Q,$\{root\} \cap E_Q$)
Function compute_all_answers(Q,Π)
IF $\Pi = \emptyset$ **THEN**
RETURN $(\emptyset, \emptyset)$
endIF
Match Q **With**
| $r :: l \Rightarrow$

RETURN $(\{\epsilon\}, \tilde{r}(\Pi) \cap lab_l(t))$

| $(r[f])^* \Rightarrow$
RETURN $(\{\epsilon\}, \tilde{r}_{cond}(\Pi, E_f))$

| $\$x \Rightarrow$
RETURN $(\{(x \leftarrow \pi) \mid pi \in \Pi\}, \Pi)$

| not $f \Rightarrow$
RETURN (ϵ, E_Q)

| $f_1[f_2] \Rightarrow$
IF Vars(f_1)=$\emptyset$ **THEN**
let (A_{f_1}, Π_{f_2})=compute_all_answers(f_1,$\Pi \cap E_{f_1}$) in
RETURN compute_all_answers(f_2,$\Pi_{f_2} \cap E_{f_2}$)
endIF
let (A_{f_1}, Π_{f_2}) =compute_all_answers($f_1, \Pi \cap E_{f_1}$) in
$A \leftarrow A_{f_1}$
for $\alpha \in A_{f_1}$ **do**
let $\Pi^{\alpha}_{f_2}$ = ensure_constraints(α, f_1) in
let ($A^{\alpha}_{f_2}$,Π')=compute_all_answers(f_2,$\Pi^{\alpha}_{f_2} \cap E_{f_2}$)
$A \leftarrow A \cup (\alpha.A^{\alpha}_{f_2})$
end for
RETURN (A, Π_{f_2})

| f_1 or $f_2 \Rightarrow$
RETURN compute_all_answers(f_1,Π) $\cup$ compute_all_answers(f_2,Π)

7.2.2 Preuve de complexité

Etant donné une position exp : $FilterExpr$ contenant les variables $(x_1, \ldots, x_n)$ dans une requête Q, un arbre t et un ensemble Π de noeuds de t, Notons $exp(t, \Pi) = \{\alpha(x_1), \ldots, \alpha(x_n) \mid [\![FilterExpr]\!]^{\alpha}_{filter} \cap \Pi \neq \emptyset\}$.

Voyons maintenant la complexité du traitement précédent :

Le précalcul coûte $O(|t||Q|)$.
Montrons par induction sur Q qu'un appel à $compute\_all\_answer(Q, \Pi)$ termine en $O(|t||Q||Q(t, \Pi)||log(|Q(t, \Pi)|))$

- Si la requête Conditional Xpath Q est une axe simple $r :: l$ où un axe conditionnel $(r[FilterExpr])^*$ l'algortihme l'évalue simplement en $O(|t|)$ à l'aide de la méthode présentée dans la proposition 1.
- Si Q est un symbôle de variables, on renvoie simplement les solutions partielles constituées des affections $\{x \leftarrow \pi \mid \pi \in \Pi\}$. Il y en a $|\Pi|$ donc au plus $|t|$ et l'appel se termine en $O(|t|)$.
- Si Q est une négation not $FilterExpr$ il est inutile de poursuivre son évaluation, puisqu'elle ne contient pas de variables et à été effectuée par le précalcul. On renvoie immédiatement le résultat déjà calculé, en $O(|t|)$.
- Si Q est une composition de filtre $f_1[f_2]$ avec f_1 et f_2 verifiant l'hypothèse d'induction, on distingue deux cas :
  1. Si f_1 ne contient pas de variables, on l'évalue ($O(|t||f_1|$ par hypothèse et puisque $|Q(f_1)| \leq 1$ car f_1 n'a pas de variables), puis on passe sur f_2 en $O(|t||f_2||f_2(t, \Pi_{f_2})|log(|f_2(t, \Pi_{f_2})|))$ par hypothèse. Au final on respecte bien la borne

$O(|t||Q||Q(t,\Pi)||log(|Q(t,\Pi)|))$[9].

2. Sinon on commence par évaluer f_1 en $O(|t||f_1||f_1(t,\Pi)|log(|f_1(t,\Pi)|))$ par hypothèse. On effectue ensuite un tour boucle pour chaque solution partielle α de $f_1(t,\Pi)$ dans lequel on fait :
   (a) un appel à ensure_constraints qui se termine en $O(|t||Q|)$ et détermine l'ensemble $\Pi^{\alpha}_{f_2}$.
   (b) un appel à compute_all_answer sur f_2.
   (c) une concatènation de la solution partielle de f_1 avec celles trouvée pour f_2 dans ce tour de boucle.

   Comme il ya moins de $|Q(t,\Pi)|$ tours de boucles, on peut borner la complexité des appels à ensure_constraints par $O(|t||Q||Q(t,\Pi)|)$.

   notons p le cardinal de $f_1(t,\Pi)$ et $\alpha_1,\ldots,\alpha_p$ les éléments de $f_1(t,\Pi)$. La complexité du reste de la boucle est donnée par $O(\sum_{k=0}^{p}|t||f_2||f_2(t,\Pi^{\alpha_k}_{f_2})|log(|f_2(t,\Pi^{\alpha_k}_{f_2})|))$ et comme pour tous les k $|f_2(t,\Pi^{\alpha_k}_{f_2})| \leq |Q(t)|$, le traitement s'effectue en $O(|t||f_2|log(|Q(t)|)\sum_{k=0}^{p}|f_2(t,\Pi^{\alpha_k}_{f_2})|)$.

   Chacune des solutions de $f_2(t,\Pi^{\alpha_k}_{f_2})$ trouvées pour f_2 lors du $k^{\text{ième}}$ tour de boucle apporte une solution globale nouvelle après concaténation avec α_k, c'est pourquoi $\sum_{k=0}^{p}|f_2(t,\Pi^{\alpha_k}_{f_2})| = O(|Q(t,\Pi)|)$.

   Enfin, la concaténation du $k^{\text{ième}}$ tour de boucle s'effectue en $O(|f_2(t,\Pi^{\alpha_k}_{f_2})|)$. Par le même argument, le traitement de toutes les concaténations se fait donc en $O(|Q(t,\Pi))|$.

   On retrouve donc bien le résultat voulu : le traitement complet s'effectue en $O(|t||Q||Q(t,\Pi)|log(|Q(t,\Pi|))$.

– Si Q est une disjonction f_1 or f_2, On calcule les solutions pour f_1 en $O(|t||f_1||f_1(t,\Pi)|log(|f_1(t,\Pi)|)))$ donc a fortiori en $O(|t||f_1||Q(t,\Pi)|log(|Q(t,\Pi)|)))$. De même, on calcule les solutions pour f_2 en $O(|t||f_2||Q(t,\Pi)|log(|Q(t,\Pi)|)))$. Puis on fait l'union des deux ensembles $f_1(t,\Pi)$ et $f_2(t,\Pi)$ la taille de ces deux ensembles est bornée par $|Q(t,\Pi)|$. Si on utilise une structure de données adaptée, comme des arbres équilibrés, cette opération peut être réalisée en temps $O(|Q(t,\Pi)|log(|Q(t,\Pi)|)))$, d'où un traitement global en $O(|t||Q||Q(t,\Pi)|log(|Q(t,\Pi)|))$.

[9] dans ce cas $f_2(t,\Pi_{f_2}) = Q(t,/Pi)$

Références

[FNTT07] Emmanuel Filiot, Joachim Niehren, Jean Marc Talbot, and Sophie Tison. Polynomial time fragments of xpath with variables. In *PODS*, 2007.

[GKP05] Georg Gottlob, Christoph Koch, and Reinhard Pichler. Efficient algorithms for processing xpath queries. In *TODS*, 2005.

[Mar05] Maarten Marx. Conditional xpath, the first order complete xpath dialect. In *TODS*, 2005.

[Sto74] L J Stockmeyer. *The Complexity of Decision Problems in Automata Theory*. PhD thesis, Department of Electrical Engineering, MIT, 1974.

[W3C99] W3C. *XML Path Language (XPath) 2.0*. http ://www.w3.org/TR/xpath20, 1999.

[W3C08] W3C. *Extensible Markup Language (XML) 1.0*. http ://www.w3.org/TR/REC-xml/, 2008.