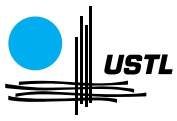
Modélisation et requêtes des documents semi-structurés : exploitation de la structure de graphe

Denis Debarbieux




Modélisation et requêtes des documents semi-structurés : exploitation de la structure de graphe

THÈSE

présentée et soutenue publiquement le vendredi 9 décembre 2005

pour l'obtention du

Doctorat de l'Université des Sciences et Technologies de Lille

(spécialité informatique)

par

Denis Debarbieux

Composition du jury

Président :	Rémi Gilleron, Professeur	Grappa, Université de Lille 3
Rapporteurs :	Véronique Benzaken, Professeur	LRI, Université Paris-Sud 11
	Francoise Gire, Professeur	CRI, Université Paris I
Examinateurs :	Stéphane Demri, Chargé de recherche au CNRS	LSV, ENS Cachan
	Mírian Halfeld Ferrari Alves, Maître de Conférences	LI, Université F. Rabelais Blois-Tours-Chinon
Directeurs :	Sophie Tison, Professeur	LIFL, Université de Lille I
	Anne-Cécile Caron, Maître de conférences	LIFL, Université de Lille I

UNIVERSITÉ DES SCIENCES ET TECHNOLOGIES DE LILLE
Laboratoire d'Informatique Fondamentale de Lille — UPRESA 8022
U.F.R. d'I.E.E.A. – Bât. M3 – 59655 VILLENEUVE D'ASCQ CEDEX
Tél. : +33 (0)3 28 77 85 41 – Télécopie : +33 (0)3 28 77 85 37 – email : direction@lifl.fr

Mis en page et composé avec LaTeX et la classe thlifl.

Remerciements

Tout d'abord, je tiens à exprimer ma sincère et profonde reconnaissance à ma directrice de thèse, Sophie Tison et à ma co-directrice, Anne-Cècile Caron pour leur soutien continu, leurs conseils avisés, leur disponibilité exceptionnelle et pour m'avoir "appris la recherche" pendant toutes ces années. Merci de m'avoir laissé libre d'aller là où mes idées m'ont mené et merci de m'avoir rattrapé avant qu'elles me mènent "tout droit dans le mur".

Je remercie chaleureusement tous les membres du jury. Véronique Benzaken et Francoise Gire pour avoir accepté d'être les rapporteurs de ma thèse, Rémi Gilleron pour avoir présider le jury. Je remercie Stéphane Demri et Mírian Halfeld Ferrari Alves e me faire l'honneur d'assister à ma soutenance.

Je souhaite aussi remercier les autres chercheurs avec qui j'ai travaillé. Merci en particulier à Yves André et à Yves Roos pour m'avoir accompagné pendant trois ans et pour les échanges de points de vue sur mon travail. J'ai aussi apprécié de pouvoir participer à de nombreux séminaires Mostrare, Tralala, L.I.F.L.... Ils m'ont permis d'élargir mes champs d'investigations.

Mes collègues de bureau ont largement participé à la bonne humeur quotidienne dans mon travail : Iovka Boneva et Laurent Planque. Je leur souhaite à tous les deux une belle carrière.

Merci à Monique pour sa relecture patiente et impitoyable de cette thèse. C'est certainement grâce à elle qu'il n'y a pas une dizaine de répétitions par page et autant de coquilles.

N'oublions pas les stagiaires, les ingénieurs, les techniciens, les secrétaires... pour leur aide et pour m'avoir permis de réaliser ce travail de la manière la plus agréable possible.

Je remercie aussi tous les membres de ma famille qui ont régulièrement pris des nouvelles de l'avancement de mon travail même si le sujet de ma thèse doit encore leur sembler obscur. Pour terminer, tout mes remerciements (et plus) pour Clotilde, ses encouragements et son soutien constant.

Table des matières

Table des figures

Liste des tableaux

Chapitre 1

Introduction

Les données semi-structurées sont des données dont la structure, inconnue *a priori*, doit être définie en l'inférant *a posteriori* à partir de la donnée elle-même (on peut dire qu'une donnée semi-structurée s'auto-décrit). Dans la mesure où la représentation choisie peut tenir compte du type de traitement qu'on souhaite appliquer à la donnée, ceci peut être vu comme un avantage. Un exemple de données semi-structurées sont les documents XML (eXtensible Markup Language [Bray et al., 1998]), ou plus largement un ensemble de documents et de liens permettant de passer de l'un à l'autre. Pour résumer, les données semi-structurées sont bien souvent complexes, hétérogènes et de formats variés.

De manière générale une donnée semi-structurée ne doit se conformer à aucun schéma. Pourtant, dans le but d'introduire de la sémantique sur les données, de comprendre la structure d'un document ou dans le but d'aider un algorithme à répondre efficacement à une requête... des schémas de données semi-structurées ont été introduits. Les DTDs (Document Type Definition) et XML-schema [Thompson et al., 2001] sont les standards retenus pour les documents XML. D'autres propositions existent comme les grammaires de graphes [Bidoit et al., 2004], ou les travaux de [Dalzilio and Lugiez, 2003], [Nestorov et al., 1998].... Bien que les documents valides vis-à-vis d'un schéma soient soumis à certaines contraintes, il reste de nombreux degrés de liberté pour les écrire. Par exemple la présence d'un **?** dans une DTD permet d'avoir des éléments optionnels alors que l'opérateur ANY permet d'avoir des nœuds de n'importe quel type. Il semble que la tendance actuelle soit plutôt à l'utilisation d'un schéma quitte à l'extraire après avoir écrit le document ([Goldman and Widom, 1997, Buneman et al., 1997]) ou à l'apprendre à partir d'un ensemble de documents qui se ressemblent ([Fernau, 2001, Bry et al., 2001, Chidlovskii, 2002]).

Une donnée semi-structurée ne doit se conformer à aucun schéma et pourtant il y a en son sein des éléments de structure (souvent écrits avec des balises) qu'il faut exploiter et qui sont utilisés par les langages de requêtes. Il est donc nécessaire de disposer de représentations (de modèles) adaptées à ces données. Il en existe deux grandes familles :

1. Une représentation arborescente, qui convient particulièrement à la modélisation d'un document XML. Cette idée est aussi valable pour représenter tous les langages à balises tels que HTML, LaTeX...
2. Une représentation par un graphe orienté étiqueté multi-racines, qui correspond à une modélisation de pages Web avec des liens hyper-textes entre ces pages. De ma-

nière générale ce modèle convient dès que les nœuds représentent des objets, simples ou complexes, composant la donnée et où les arcs représentent la composition de ces objets, par agrégation ou par association. Cette représentation est celle décrite dans [Abiteboul et al., 2000].

Les bases de données semi-structurées ne s'expriment que très difficilement dans les modèles de bases de données relationnelles ou objets [Rys et al., 2005]. Plusieurs raisons peuvent expliquer qu'un document semi-structuré est difficilement exprimable dans le modèle relationnel. Par exemple, les applications Internet produisent et manipulent de nombreux documents semi-structurés dont la structure évolue dynamiquement, ce qui est un véritable problème dans le modèle relationnel. Au niveau logique, un attribut peut être absent d'une donnée ou le même attribut peut avoir deux types différents en deux endroits différents d'un même document, ce qui est incompatible avec une base relationnelle. C'est pourquoi tous les systèmes actuels permettant de manipuler XML stockent les données de façon native.

Une base de donnée est dite native XML si elle est spécifiquement conçue pour XML. Le modèle utilisé est prévu pour le stockage et l'accès à des arbres ordonnés. Le document XML est l'entité centrale de la base (comme une relation d'une base de données relationnelle). Autrement dit, le document XML est une entité logique du système. Deux avantages sont associés à une base native XML : les algorithmes de chargement des gros documents et les algorithmes de mise à jour sont très efficaces.

Pour offrir les mêmes possibilités que le modèle relationnel plusieurs problèmes sont à résoudre dans les bases de données semi-structurées [XMLbase, 2004], [Klarlund et al., 2004]. Les références suivantes sont à considérer comme un échantillon des questions à se poser. Par exemple, la question de l'intégrité référentielle (notion intiment liée à celle de la normalisation) dans les documents XML est un sujet encore non résolu [Arenas and Libkin, 2004]. De même, beaucoup de questions intéressantes se posent lorsque l'on souhaite évaluer, à la volée, une requête sur un document XML (problème du streaming) [Segoufin and Vianu, 2002]. Citons enfin le problème de l'inclusion entre deux requêtes Xpath [Miklau and Suciu, 2002, Neven and Schwentick, 2002].

Dans cette thèse, une donnée semi-structurée est modélisée par un graphe. On exploite cette représentions pour étudier les trois thèmes suivants :

1. *Les requêtes :* comment définir un langage de requêtes qui sélectionne des nœuds d'un graphe ?
2. *Les contraintes d'intégrités :* comment exploiter des contraintes définies sur les chemins d'un graphe ?
3. *L'indexation :* Comment conserver la structure d'une donnée lors de son indexation ?

La structure interne d'un document XML est un arbre ordonné d'arité non bornée dont l'ensemble des balises est noté Σ. Une fonction τ associe à tout nœud d'un arbre un label de Σ. On dit que la fonction τ type les nœuds du document. Généralement, la racine de l'arbre a un label particulier : /. Cette structure est souvent enrichie en y ajoutant un ensemble de références entre deux nœuds. Le document peut alors être vu comme un graphe mais dont **la structure inhérente est celle d'un arbre**. Afin que cette structure reste visible, les arcs *arbres* et les arcs *références* n'ont pas le même type.

La figure 1.1 représente un document modélisé par un arbre. Cette donnée correspond à un journal. Ce journal est composé de deux articles qui contiennent un ou deux fils typés

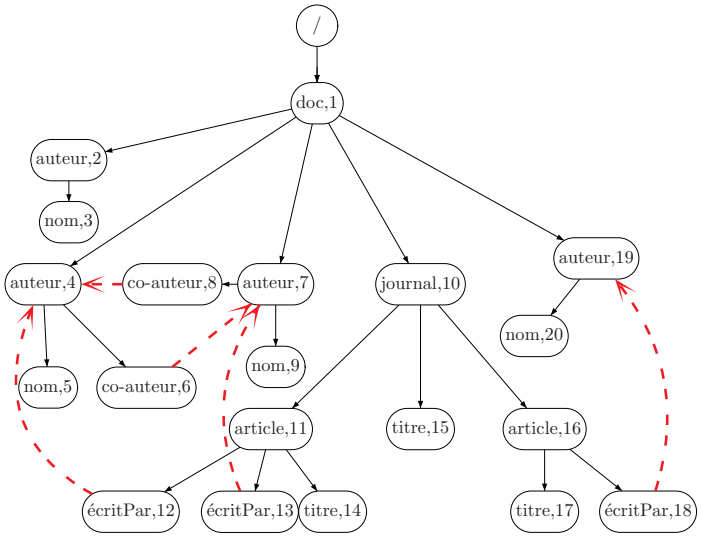


FIG. 1.1 – Représentation d'une donnée semi-structurée par un arbre et des références

par le label *écritPar*. Ce document XML contient aussi quatre auteurs avec leur nom et éventuellement un co-auteur. Cette base correspond à la structure d'arbre du document. On peut lui ajouter, par des arcs différenciés (en pointillés), des références. Ces références signifient soit que chaque article est écrit par un ou plusieurs auteurs, soit que deux auteurs d'un même article sont co-auteurs de cet article.

Structurer un document par un graphe étiqueté orienté **multi racines** revient à le représenter par un triplet $< N, R, T >$. Les étiquettes du graphe appartiennent à un alphabet A fini, N est un ensemble de nœuds. Certains de ces nœuds sont particuliers : ce sont les racines du document. On les regroupe dans l'ensemble R. Enfin T est un ensemble inclus dans $N \times A \times N$ et représente les transitions de ce graphe.

Au lieu d'ajouter des références dans le document de la figure 1.1, on peut aussi le représenter par un graphe (figure 1.2). Du point de vue de la structure, tous les nœuds de ce graphe ont un identifiant unique, l'unique racine est le nœud 0 (il pourrait y avoir plusieurs racines) et il est cyclique (grâce aux arcs co-auteur). En déplaçant les labels des nœuds vers les arcs entrants on peut transformer tout arbre en graphe ; la réciproque est fausse : un graphe peut avoir plusieurs racines. Le graphe est donc plus général que l'arbre mais il est moins informatif (par exemple l'information sur l'ordre des fils est perdue).

Interroger un document semi-structuré consiste à sélectionner des ensembles de nœuds de ce document (certains langages permettent de sélectionner des n-uplets de nœuds). Pour cela on navigue dans le document en utilisant les arcs et on restreint les nœuds visités grâce à des prédicats. En imposant des restrictions sur les alphabets utilisés, on va pouvoir définir des langages tels que Xpath [Berglund et al., 2002a], Xquery [Berglund et al., 2002b], les requêtes régulières de chemins [Abiteboul and Vianu, 1997]...Dans tous les cas, si q est une requête, D est un document et C un ensemble de nœuds du document, $D(C, q)$ représente l'ensemble des nœuds de D sélectionnés par q dans le contexte C. On peut ainsi combiner deux requêtes : combiner les requêtes p et q consiste à évaluer p, puis à évaluer q à partir du contexte $D(C, p)$. On évalue alors $D(D(C, p), q)$.

Xpath est proposé par le W3C comme le langage pour sélectionner des nœuds dans un document XML. Deux raisons font que Xpath est devenu un langage important. Premièrement il est adopté par de nombreuses technologies utilisées de la communauté XML. On peut par exemple citer XSLT, Xpointer ou Xquery. Deuxièmement l'utilisation de ces technologies dans le monde de l'entreprise et dans les systèmes de bases de données ne cesse d'augmenter.

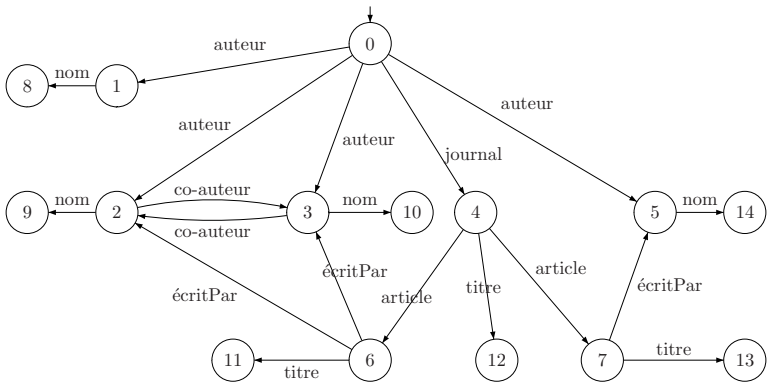


FIG. 1.2 – Représentation d'une donnée semi-structurée par un graphe

Pour répondre à une requête type Xpath ou Xquery, le document est vu comme un arbre dans lequel on ajoute des références. La sous-structure d'arbre étant connue il est très facile de naviguer dans le document. Onze axes sont définis à cet effet tels que *child, ancestor, descendent-or-self*... De plus, on enrichit Xpath de deux axes permettant de naviguer en suivant les références : *idref* permet de suivre un lien référence alors que *reverse idref* permet de le remonter.

Par exemple, on peut sélectionner, dans un document XML, les auteurs d'un article qui ont un nom (les nœuds 4, 7 et 19 dans la figure 1.1) à l'aide de la requête Xpath suivante : /descendent-or-self::écritPar/idref::auteur[child::nom]. La partie entre crochets correspond à un prédicat. Ce prédicat sélectionne les nœuds qui ont un fils étiqueté par le label *nom*. Elle s'exprime dans le contexte défini par le tronc de la requête. Le tronc de la requête (/descendent-or-self::écritPar/idref::auteur) est une requête absolue qui sélectionne les nœuds d'un document XML : on sélectionne les descendants de la racine étiquetés par *écritPar* puis les nœuds de type *auteur* accessibles par une référence.

Le deuxième type de requêtes étudié dans cette thèse sont les requêtes régulières qui s'utilisent sur le modèle de graphe. Contrairement aux requêtes Xpath, une requête régulière s'exprime exclusivement avec les labels des arcs. On forme des mots avec les labels de A. On associe à chaque mot les chemins étiquetés par ce mot dans le document. Le résultat est alors l'ensemble de nœuds accessibles par ces chemins depuis un nœud du contexte C. Si q est une expression régulière et D un document alors $D(C, q)$ est l'union des $D(C, u)$ pour tous les mots u de $L(q)$, le langage régulier décrit par q.

La requête régulière réduite au mot journal.titre sélectionne, depuis la racine, le nœud 12 dans le document de la figure 1.2. La requête régulière auteur.co-auteur*.nom sélectionne les nœuds 8, 9, 10 et 14.

On a montré la spécificité des documents semi-structurés et la place croissante qu'ils occupent dans les bases de données. Un point clé pour que cette extension continue est la qualité des algorithmes qui répondent aux requêtes. Trois facteurs peuvent influer sur cette qualité :

1. *Évaluation d'une requête sur un document* : étant donnés une requête q, un document D et un contexte C, comment évaluer $D(C, q)$ de manière efficace ? G .Gottlob, C. Koch et R .Picler ont proposé dans [Gottlob et al., 2002] un fragment de Xpath (contenant toutes les possibilités de navigation Xpath) pour lequel l'évaluation est linéaire dans la taille combinée de D et q (i.e. leur algorithme est $O(|D|.|q|)$). De même on peut répondre à une requête régulière en utilisant un algorithme type produit cartésien et avoir une complexité en $O(|D|.|q|^2)$ [Abiteboul et al., 2000]. Le problème est souvent de trouver un langage de requête le plus expressif possible dont on sait évaluer efficacement les requêtes.

2. *Optimisation|transformation des requêtes* : Peut-on réécrire la requête en une requête plus profitable (i.e. dont on peut tirer profit pour une évaluation rapide) ?

   Par exemple, on peut simplifier une requête q en recherchant des sous-requêtes de q qui sélectionnent le même ensemble de nœuds (i.e. enlever les redondances d'une requête). S. Amer-Yahia et al introduisent dans [Amer-Yahia et al., 2001] des algorithmes qui minimalisent des requêtes connaissant des éléments de schémas (par exemple, un nœud de type A a toujours un descendant de type B). On peut aussi utiliser des systèmes de clefs primaires | clefs étrangères comme le proposent P. Buneman et al dans [Buneman et al., 2000a] pour les documents XML sans référence ou W. Fan et J. Siméon mais en utilisant des références [Fan and Simeon, 2000].

   Une autre piste est d'utiliser un ensemble de contraintes satisfaites par le document pour trouver (tester si) des requêtes (sont) équivalentes. Pour cela on étudie souvent le problème d'inclusion (containment) : on dit qu'un document D satisfait une contrainte (dans un contexte C fixe) "p est inclus dans q" si le résultat $D(C, p)$ est inclus dans le résultat $D(C, q)$. Le problème étudié est : étant donné un ensemble de contraintes $\mathcal{C}$, tout document satisfaisant $\mathcal{C}$ satisfait-il aussi la contrainte "p est inclus dans q" ? Ce problème est par exemple étudié par S. Abiteboul et V. Vianu dans [Abiteboul and Vianu, 1997] mais aussi dans [Schwentick, 2004] ou dans [Miklau and Suciu, 2002].

   Une dernière voie est l'utilisation de vues. Les nœuds sélectionnés par une vue étant pré-calculés, il suffit d'interroger le document uniquement par rapport aux fragments de la requête qui ne sont pas exprimables avec des vues. Les auteurs de [Grahne and Thomo, 2001] et de [Grahne and Thomo, 2003] proposent de tels algorithmes.

3. *Indexation* :

   Peut-on transformer le document afin de rendre les algorithmes décrits dans le premier item plus efficaces ? Ces transformations reviennent en fait à indexer le document. Le document étant modélisé par un graphe, les algorithmes d'indexation doivent utiliser des techniques d'indexation de graphes. Deux grandes familles d'index existent dans la littérature.

   La première technique consiste à regrouper des nœuds du document qui sont indifférenciables par les requêtes. Deux nœuds n_1 et n_2 sont indifférenciables si quelque soit la requête q, n_1 est selectionné par q si et seulement si n_2 l'est aussi. Un avantage de cette méthode est que la taille de l'index est plus petite que la taille de la donnée. Les

auteurs de [Milo and Suciu, 1999], de [Amer-Yahia et al., 2001] ou de [Ramanan, 2003] proposent des algorithmes basés sur cette idée.

La seconde technique modifie la forme du document pour développer des algorithmes d'évaluation 'ad hoc'. La base de ces index est d'ajouter du déterminisme dans les graphes [Goldman and Widom, 1997], [Abiteboul et al., 2003] ou [Bertino et al., 2004]. Ce déterminisme peut aussi servir à définir un schéma, un guide (par exemple le data**guide**) du document. En ce sens l'indexation est aussi une recherche de schéma.

L'étude des requêtes sur les documents semi-structurés est donc très importante et c'est un domaine dans lequel la recherche fondamentale mais aussi pratique est très active. Reprenons les trois points précédents pour présenter l'organisation et les principaux résultats de ce mémoire.

Chapitre 2 : les requêtes graphes

Dans le chapitre 2 nous expliquons comment utiliser les documents XML pour modéliser une donnée qui a une structure de graphe. On présente le modèle classique (arbre ordonné, ensemble de références) mais aussi le XML coloré [Jagadish et al., 2004]. Le principe est de prendre en compte plusieurs documents XML qui se partagent des nœuds. Chaque document peut ainsi avoir une sémantique propre et être encore plus informatif que le XML classique.

On définit alors les requêtes graphes qui exploitent simultanément la structure d'arbre sous-jacente à chaque document, les références entre deux nœuds et les différents documents présents. Ces requêtes permettent de naviguer dans chaque arbre XML avec les axes classiques (*child*, *ancestor*...) mais aussi de changer d'arbre grâce aux nœuds communs à plusieurs documents XML. On montre ensuite que l'évaluation de telles requêtes est un problème NP-complet.

On étudie enfin l'expressivité de ce langage de requêtes. Le XML coloré étant une extension de XML, on replace nos résultats dans deux contextes connus. On montre par exemple que toute requête du Core Xpath [Gottlob et al., 2002] (fragment de Xpath qui contient tous les axes de navigation dans un document XML) peut s'écrire avec une requête graphe. On montre de même que les requêtes conjonctives pour XML [Gottlob et al., 2004] sont un cas particulier des requêtes graphes. On propose enfin le Core Xquery coloré avec égalité, un langage basé sur Xquery, qui permet de tester l'égalité entre deux nœuds du document. On montre que les requêtes graphes ont exactement la même expressivité que ce langage.

Chapitre 3 : documents représentés par un graphe : contraintes d'inclusions

Dans ce chapitre, les documents sont vus comme des graphes à plusieurs racines dont les arcs sont étiquetés. Les requêtes utilisées sont donc les requêtes régulières.

Dans le but d'évaluer ou d'optimiser des requêtes, on utilise des informations liées à la structure du graphe. Certaines de ces contraintes portent sur les chemins et ont été introduites dans [Abiteboul and Vianu, 1997]. On étudie plus particulièrement les contraintes d'inclusion mais d'autres classes ont déjà été étudiées ([Buneman et al., 1999, Alechina et al., 2003, Grahne and Thomo, 2003]).

On définit donc la notion de contrainte d'inclusion : soient p et q deux requêtes régulières, un document est modèle de la contrainte d'inclusion $p \preceq q$ si tout nœud solution de p est aussi solution de q. Si p est inclus dans q et vice-versa on dit que p et q sont équivalentes.

Soit $\mathcal{C}$ un ensemble de contraintes, on dit que $\mathcal{C}$ implique une contrainte $p \preceq q$ si tout document modèle de $\mathcal{C}$ est aussi un modèle de $p \preceq q$. Le problème de l'implication décide à partir d'un ensemble de contraintes d'inclusions $\mathcal{C}$ et deux requêtes p et q si $\mathcal{C}$ implique la contrainte $p \preceq q$.

La figure 1.2 satisfait la contrainte de l'ensemble $\mathcal{C}$ ={ auteur.co-auteur$\preceq$ auteur}. Tout document vérifiant cette contrainte vérifie aussi la contrainte auteur.co-auteur$^2 \preceq$ auteur. On étend même le raisonnement pour obtenir que $\mathcal{C}$ implique que auteur.co-auteur$^* \preceq$ auteur.

Grâce à des techniques inspirées de la théorie des automates, de la théorie des transducteurs et de la théorie de la réécriture préfixe on peut manipuler les contraintes d'inclusions et prouver les résultats suivants.

On montre que tout ensemble $\mathcal{C}$ de contraintes a un modèle exact : il existe un document $D_\mathcal{C}$ tel que $D_\mathcal{C}$ soit un modèle de $p \preceq q$ si et seulement si $\mathcal{C}$ implique que $p \preceq q$ [Debarbieux et al., 2003] et [Debarbieux et al., 2004]. Dans ces mêmes articles on affine le problème à l'étude des modèles exacts finis. De plus, on y montre que le problème de l'implication est PSPACE-complet dans le cas des contraintes bornées mais qu'il devient quasi-linéaire dans le cas très particulier des équivalences entre mots [Andre et al., 2004]. On utilise ce résultat pour chercher à partir d'une requête q une requête q_f telle que q et q_f soient équivalentes et $L(q_f)$ soit fini. C'est le problème dit de l'équivalence finie.

Chapitre 4 : requêtes régulières : indexation de chemins

Outre des contraintes d'intégrité, un logiciel qui gère beaucoup de données doit utiliser des index pour répondre de manière efficace à une requête. Par exemple, dans le modèle relationnel, des tables de hachage ou des B-arbres sont utilisés pour stocker les valeurs des attributs. De même, lore [Lore, 1997] ou eXist [eXist, 2000] sont des bases de documents semi-structurés qui utilisent des index. La forme de ces index dépend bien évidemment des requêtes considérées par ces logiciels.

On s'est donc consacré à l'indexation des documents graphes. Un index est lui-même un graphe orienté étiqueté à plusieurs racines. Grâce aux propriétés sur sa forme, sa taille, etc., il est plus efficace d'interroger l'index que le document d'origine. De nombreux index ont déjà été étudiés et nous nous sommes intéressés aux liens qui existent entre les contraintes d'inclusions satisfaites par un index et celles satisfaites par le document. Un index permet déjà de répondre à une requête sans consulter le document. On souhaite donc savoir s'il est possible d'utiliser la structure de l'index (ses contraintes d'inclusion) pour optimiser une requête sans se référer aux contraintes satisfaites par le document.

On extrait à partir du dataguide [Goldman and Widom, 1997] les contraintes satisfaites par un document [Andre et al., 2004]. Cet algorithme nous sert aussi à trouver les contraintes communes à plusieurs données. On utilise enfin cet algorithme pour montrer qu'un document et son 1-index [Milo and Suciu, 1999] satisfont exactement le même ensemble de contraintes[Andre et al., 2005].

Afin d'étudier de possibles optimisations de requêtes avec des contraintes d'égalité, nous utilisons un outil développé pendant cette thèse : Qric (Query Rewriting with inclusion constraints [Debarbieux and Luchier, 2004]). Cet outil permet d'interroger un document graphe avec une requête régulière, d'extraire les contraintes d'égalité de mots d'un document, de réécrire une requête régulière par rapport à un ensemble de contraintes, d'indexer un

document... On s'intéresse plus particulièrement à la problématique suivante : étant donné un ensemble de documents qui se ressemblent, quel est le meilleur moyen de les interroger ? Construire un index commun à tous les documents ? Les indexer séparément ? Réécrire les requêtes par rapport à des contraintes d'inclusions communes ?

Chapitre 2

Les requêtes graphes

2.1 Introduction

Ce chapitre présente une modélisation de données semi-structurées par des graphes et un langage de requêtes associé. Pour cela, on utilise le formalisme associé à XML, eXtensible Markup Language, un langage de description de documents semi-structurés. Un document XML s'écrit textuellement avec des balises. Mais sa structure interne est un arbre ordonné d'arité non bornée. Deux dérivations de ce modèle permettent de former un graphe :

1. On peut enrichir l'arbre XML de références et obtenir un graphe.
2. On peut utiliser, non pas un document XML, mais un ensemble d'arbres XML qui partagent des nœuds, comme c'est le cas pour le XML coloré [Jagadish et al., 2004].

Il est bien sûr possible de combiner ces deux solutions et de considérer un ensemble de documents XML avec références qui ont des nœuds en commun.

Supposons que quelqu'un désire écrire un document concernant un journal. Ce document doit contenir les informations suivantes : le titre du journal, l'ensemble des articles publiés dans le journal et un ensemble d'auteurs. On souhaite de plus, stocker l'information qui précise que deux auteurs d'un même article sont co-auteurs l'un de l'autre.

Utiliser un arbre XML pour modéliser un tel document est possible et le document n'est pas très difficile à construire. Il y a néanmoins un problème : la redondance d'information. En effet, un article pouvant avoir plusieurs auteurs et un auteur pouvant publier plusieurs articles, quelle que soit l'organisation de l'arbre il y a une redondance soit sur les articles soit sur les auteurs.

Une solution pour pallier ce problème est d'ajouter des références entre des nœuds de l'arbre. Le document peut alors être vu comme un graphe dans lequel les arcs *arbres* et *références* n'ont pas la même nature. Dans cette représentation, la structure d'arbre reste visible dans le graphe.

Par exemple, le document XML suivant (seule une partie des balises sont présentes) peut se représenter par le graphe de la figure 1.1 (page 3).

```
<doc id=1>
  <auteur id=2>
```

```
      <nom id=3/>
    </auteur>
    <auteur id=4>
      <nom id=5/>
      <co-auteur id=6 href=7/>
    </auteur>
    <auteur id=7>
      <co-auteur id=8 href=4/>
      <nom id=9/>
    </auteur>
    <journal id=10>
      <article id=11>
        <ecritPar id=12 href=4/>
        <ecritPar id=13 href=7/>
        <titre id=14/>
      </article>
    ...
    </journal>
    ...
  </doc>
```

La structure de ce document est basée sur un arbre qui contient un journal et des auteurs. Les auteurs ont éventuellement un co-auteur et le journal a un titre et des articles. Chaque article a un fils de type *titre* et des (un) fils de type *écritPar*. Il y a de plus des références entre des nœuds *écritPar* et des nœuds *auteur* (qui signifient que chaque article est écrit par un ou plusieurs auteurs) et des références entre des nœuds *co-auteur* et des nœuds *auteur* (qui signifient que deux auteurs d'un même article sont co-auteurs de cet article).

Les références dans un document XML sont représentées grâce à deux attributs : *id* et *href*. Les attributs d'un nœud sont généralement représentés comme des fils non ordonnés du nœud. Mais, dans ce chapitre, nous en avons une représentation différente. Tous les nœuds ont un attribut *id* qui est leur unique identifiant (dans la figure 1.1 chaque nœud contient son identifiant et le nom de la balise XML auquel il correspond). On considère qu'une égalité de valeur entre un attribut *href* d'un nœud n_1 et un attribut *id* d'un nœud n_2 permet de construire un arc de type référence entre les nœuds n_1 et n_2 (si l'attribut *href* d'un nœud ne correspond à aucun attribut *id*, alors cet attribut n'est pas représenté dans le graphe). Dans la figure 1.1, il y a un arc référence (en pointillé) du nœud 12 vers le nœud 4 car l'attribut *href* du nœud 12 a la valeur 4.

Ce modèle peut encore être critiqué. En effet, dans l'exemple ci-dessus, il est nécessaire de passer par un nœud *journal* pour accéder aux informations concernant les articles. Cette

difficulté peut être contournée en construisant un document plat (i.e. de profondeur un ou deux) et en utilisant uniquement le mécanisme id | href pour échanger de l'information entre les éléments. Ce type de document n'est pas la meilleure solution : on perd, en effet, la structure arborescente qui est l'essence d'un document XML. De plus, dans une implémentation classique du standard XML, les références sont des jointures entre un nœud *id* et un nœud *href*. Leur coût est beaucoup plus important que le coût d'une liaison père | fils dans un arbre.

Une troisième réponse, celle qui est considérée dans ce chapitre, est d'utiliser les documents XML coloré [Jagadish et al., 2004]. Un document XML coloré se représente par un ensemble d'arbres XML. Chaque arbre a une couleur différente (i.e. une sémantique propre) et, afin d'éviter la redondance, deux arbres différents peuvent se partager un même nœud (ce nœud est alors multicolore). Si on continue l'exemple précédent, notre donnée peut se représenter par un document à trois couleurs : le bleu pour les auteurs, le vert pour les articles et le rouge pour les journaux (figure 2.1). Pour la seconde fois, une structure de graphe apparaît dans le modèle XML. La figure 2.1 met aussi en évidence les références présentes dans un document XML coloré (les références ne sont autorisées qu'à l'interieur d'un même arbre). C'est un enrichissement du modèle proposé par Jagadish et al.

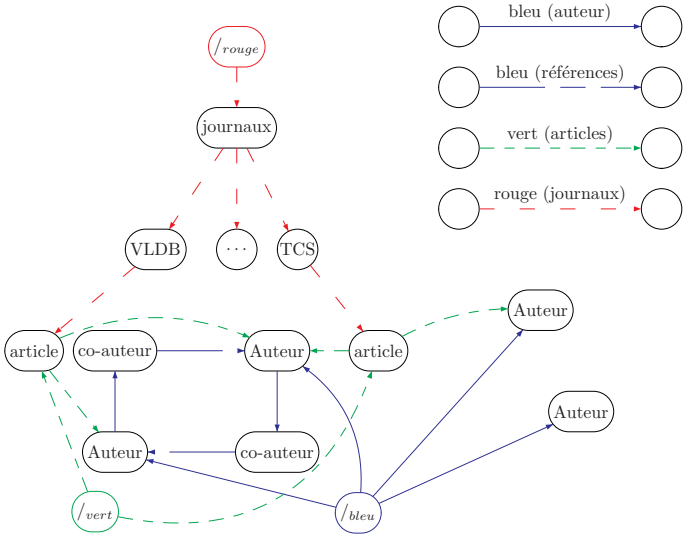


FIG. 2.1 – Document XML coloré représenté par un graphe

Un nœud pouvant être accessible par un arc *arbre* ou par un arc *référence*, un nœud pouvant appartenir à plusieurs arbres... on souhaite développer un langage de requêtes qui permette d'exprimer l'égalité entre deux nœuds. L'un des buts de ce chapitre est donc de définir les requêtes graphes qui permettent d'interroger un document XML coloré modélisé par un graphe. Si un document XML coloré ne contient qu'une couleur alors c'est un document XML avec des références. Un autre objectif de ce chapitre est donc de comparer l'expressivité de notre langage de requête avec XPath et XQuery, les langages associés à XML.

Par exemple, on peut sélectionner dans un document XML (figure 1.1), le titre des articles des journaux par la requête XPath suivante :

```
/descendent-or-self::journal/child::article/child::titre
```

Cette requête permet de naviguer dans l'arbre suivant les axes *descendent-or-self* et *child*. *journal*, *article* ou *titre* sont des tests sur les nœuds du document et permettent (respectivement) de sélectionner les nœuds du document qui ont le type (la balise) *journal*, *article* ou *titre*.

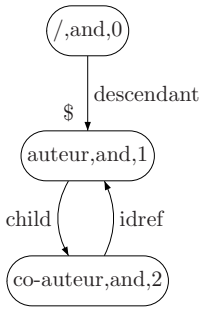


FIG. 2.2 – Exemple de requête graphe

Une requête XPath permet aussi d'utiliser une référence par une égalité de valeur entre un attribut *id* et un attribut *href*. La requête XPath suivante permet, par exemple, de sélectionner le nom des auteurs qui ont écrit un article dans un journal :

```
/descendent-or-self::nom[parent::auteur/@id=
    /descendent-or-self::écritPar/@href]
```

La partie entre crochets est un prédicat qui est vrai pour les nœuds dont le père est de type *auteur* et qui sont accessibles par une *référence* depuis un nœud typé par *écritPar*. Les prédicats se combinent entre-eux grâce aux opérateurs booléens *and* et *or*.

Bien que les références soient des arcs du graphe modélisant le document XML, XPath n'a pas d'axe prédéfini permettant de les utiliser. Il est néanmoins possible de suivre un lien de type référence en calculant une jointure entre un attribut de type *id* et un attribut de type *href*. On enrichit donc ce langage de deux axes [Ramanan, 2003] :

– *idref* qui permet de suivre un lien de type référence.
– *ridref* (pour *reverse idref*) qui permet de remonter un lien référence.

XPath permet de tester l'égalité entre deux nœuds grâce à l'opérateur **is**. Par exemple, la requête XPath suivante sélectionne les auteurs qui sont co-auteurs d'eux-mêmes.

```
/descendent::auteur[self::* is  child::co-auteur/idref::auteur]
```

Cette requête XPath est exprimable par la requête graphe de la figure 2.2. Les arcs de cette requête sont étiquetés par un axe de navigation dans le document, les nœuds sont étiquetés par un triplet (prédicat unaire, opérateur booléen choisi dans l'ensemble {and,or}, identifiant). Enfin, le nœud 1 est particulier puisqu'il est désigné par le symbole $.

Répondre à cette requête graphe consiste à construire toutes les fonctions β possibles des nœuds de la requête vers des nœuds du document tels que :

– $\beta(0)$ soit la racine du document et $\beta(1)$ soit un descendant de $\beta(0)$,

– $\beta(1)$ porte le label *auteur* et on doit atteindre depuis $\beta(1)$, le nœud $\beta(2)$ en suivant l'axe *child*,
– $\beta(2)$ est étiqueté par *co-auteur*. De plus, il existe un axe *référence* de $\beta(2)$ vers $\beta(1)$.

Un nœud n d'un document est sélectionné par cette requête graphe s'il existe une fonction β décrite ci-dessus telle que l'antécédent de n par β soit le nœud marqué par le symbole \$ (ici, tel que $\beta(1) = n$).

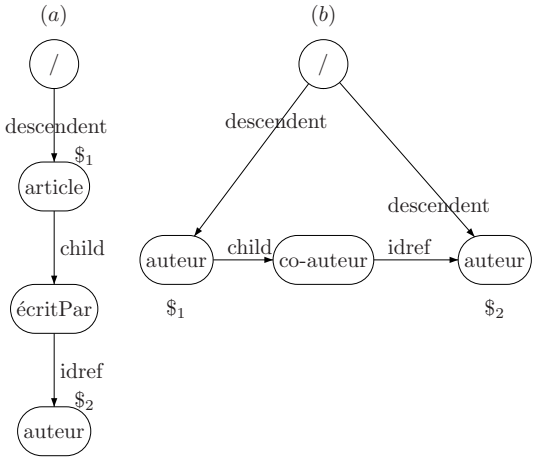


FIG. 2.3 – Exemple de requêtes graphes binaires

En plus du *and* et du *or*, un troisième opérateur booléen est utile : le *not*. Par manque de temps, nous ne l'avons pas intégré dans notre formalisme.

XQuery est un langage de transformation d'arbres XML (i.e. sélection de nœuds et reconstruction d'arbres). Dans cette thèse, nous ne considérons que la partie du langage qui sélectionne des nœuds. Elle est basée sur XPath pour définir des requêtes n-aires. La requête XQuery suivante sélectionne, dans le document 1.1, tous les couples $(article, auteur)$ du document :

```
for $article in /descendent::article
  for $auteur in $article/child::écritPar/idref::auteur
  return ($article, $auteur)
```

La figure 2.3 (a) (dans un souci de lisibilité, les opérateurs *and* ne sont pas représentés) montre que cette requête peut se représenter par une requête graphe dans lequel il y a deux sélectionneurs ($\$_1$ et $\$_2$). Cette requête est en fait une requête 'arbre'. Cette même figure (partie (b)) montre que l'on peut écrire des requêtes n-aires qui se représentent par des graphes. Celle-ci permet de sélectionner les couples d'auteurs[1] qui ont écrit un article en commun ce qui peut se traduire avec la requête XQuery suivante :

```
for $auteur1 in /descendent::auteur
   for $auteur2 in /descendent::auteur
   where $auteur1/child::co-auteur/idref::* is $auteur2
```

[1] Un couple peut être un doublon

```
return ($auteur1,$auteur2)
```

On propose d'étendre, sur le principe proposé par les auteurs de [Jagadish et al., 2004], la syntaxe de XPath et de XQuery pour naviguer dans un document XML coloré : l'idée est de préfixer l'axe de navigation utilisé par la couleur de l'arbre considéré. Par exemple, trouver les auteurs d'un article publié à TCS (cf. figure 2.1) peut se faire avec la requête suivante :

```
/{rouge}descendent::TCS/{rouge}child::article/{vert}child::auteur
```

Cette requête cherche parmi les journaux (arbre rouge), le nœud de type *TCS*. A partir de ce nœud, la requête localise tous les articles de TCS. Les nœuds *articles* étant communs à l'arbre rouge et l'arbre vert (description des articles), la sous requête `{vert}child::auteur` permet de trouver les auteurs d'un article à TCS.

On définit de la même manière des requêtes graphes sur des documents XML coloré : la figure 2.4 propose une requête qui sélectionne les auteurs qui ont une publication à VLDB et une publication à TCS. Elle se traduit en XPath coloré par :

```
/{rouge}descendent::VLDB/{rouge}child::*/{vert}child::auteur
[self::* is /{rouge}descendent::TCS/{rouge}child::*/{vert}child::auteur]
```

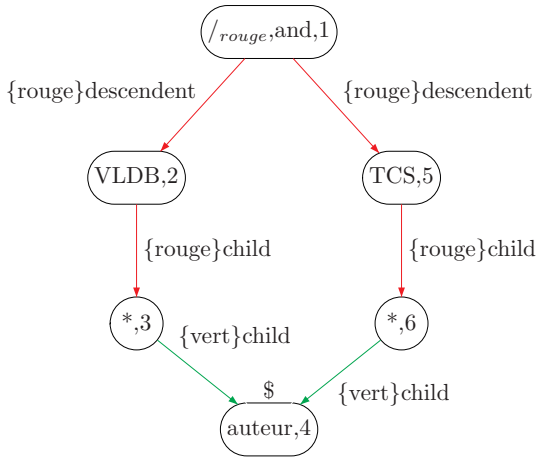


FIG. 2.4 – Exemple de requête graphe sur des documents XML coloré

Les requêtes graphes permettent aussi d'exprimer des requêtes disjonctives. Par exemple, si on remplace l'opérateur booléen *and* du noeud 1 (figure 2.4) par un *or*, alors la requête sélectionne les auteurs qui ont une publication à VLBD ou à TCS. Dans ce cas une fonction (partielle) β est un plongement de q vers un document D si elle est définie soit sur le chemin (1,2,3,4) soit sur le chemin (1,5,6,4).

Remarquons que les requêtes graphes entrent dans la catégorie des "Pattern Query". On peut même dire que ce sont des "graph pattern queries". Dans ce sens on étend les travaux de [Amer-Yahia et al., 2001], de [Gottlob et al., 2004] ou de [Gottlob et al., 1998] (ou [Gottlob et al., 2001] pour la version journal).

Un objectif de ce chapitre est de définir le Core XQuery coloré avec des égalités de nœuds ($C_=$XQuery coloré) et de le modéliser par des requêtes graphes. Un mécanisme de variables (inspiré de XQuery) définit les nœuds d'une requête. Les arcs du graphe sont construits à partir d'égalité entre les variables.

Pour terminer cette introduction, résumons les apports de ce chapitre :

1. Nous définissons les requêtes graphes qui permettent d'interroger des documents XML coloré. Notre approche étend les requêtes conjonctives définies dans [Gottlob et al., 2004] sur deux points : on ne restreint pas les opérateurs aux *and* mais on prend en compte le *or*. De plus, on modélise un document par un graphe et non par un arbre. Signalons aussi que les requêtes graphes généralisent les travaux de [Ramanan, 2003] sur l'étude de Core XPath.
2. A partir de la notion de document XML coloré [Jagadish et al., 2004], nous proposons le Core XQuery coloré avec des égalités ($C_=$XQuery coloré). Ce langage, inspiré de XPath et de XQuery, permet de faire des requêtes n-aires en naviguant dans le document et en exprimant des égalités entre les nœuds d'un document. Il a la même expressivité que les requêtes graphes.
3. On montre que l'évaluation d'une requête graphe sur un document XML coloré est NP-complète. On en déduit que le $C_=$XQuery a la même complexité.

Signalons enfin que les points 2 et 3 recoupent des résultats publiés en Juin 2005 [Koch, 2005a, Koch, 2005b]. Dans ses travaux, C. Koch définit le Core XQuery (pour des documents XML monochrome et sans référence) et étudie la complexité de ce langage. Bien que ce chapitre ait été écrit indépendamment, la syntaxe, l'expressivité et les techniques utilisées pour étudier XQ^- (le Core XQuery sans composition) dans [Koch, 2005b] sont semblables aux nôtres.

Organisation de ce chapitre

La section 2.2 présente les requêtes graphes pour les documents XML coloré. Dans un premier temps, on rappelle les notions de XML, XPath et XQuery utiles pour notre langage de requêtes. Dans un second temps, la syntaxe et la sémantique des requêtes graphes sont données.

La section 2.3 se focalise sur deux exemples de requêtes (graphes) existant dans la littérature. Le Core XPath est le fragment de XPath qui contient toutes les possibilités de navigation. On explique comment une requête du Core XPath peut se représenter par une requête arbre (cas particulier des graphes). On donne enfin un algorithme très efficace qui évalue une telle requête sur un document XML. Le deuxième exemple est les requêtes conjonctives pour XML dont l'évaluation est NP-complète.

La section 2.4 étudie formellement les requêtes graphes. On y étudie le problème de l'évaluation d'une requête graphe : on prouve que le problème de décider si un n-uplet de nœuds est sélectionné par une requête graphe est NP-complet. On utilise les résultats de [Gottlob et al., 2004] pour étudier précisément la complexité des requêtes graphes. Du point de vue de l'expressivité, on fait le lien entre les requêtes graphes et XQuery : on propose le Core XQuery coloré avec des égalités. Ce langage permet d'exprimer des égalités de nœuds

à l'aide d'un mécanisme de variables. Il a exactement la même expressivité que les requêtes graphes.

2.2 Interroger un document XML coloré

Le but de cette section est de présenter les requêtes graphes pour les documents XML coloré. Dans un premier temps, on expose les grandes idées liées aux documents XML non coloré. On introduit notamment les deux principaux langages de requêtes pour XML que sont XPath et XQuery. Dans un second temps on définit les documents XML coloré et les requêtes graphes.

2.2.1 XML, XPath et XQuery

La particularité d'un document XML est de contenir en même temps des données et des informations permettant d'identifier la structure et le sens de ces données. Il est alors utile de pouvoir s'appuyer sur cette information pour désigner une partie d'un document XML.

La structure d'un document XML est celle d'un arbre ordonné d'arité non bornée. Si de plus, on modélise une jointure entre un attribut *href* et un attribut *id* par une référence on obtient la définition suivante :

Définition 2.1 *Un document XML D est représenté par un couple (A, ref) dans lequel A est un arbre ordonné d'arité non bornée et ref est un ensemble de références (orientées) entre des nœuds de l'arbre.*

Dans la suite de ce chapitre, si D est un document XML alors $/_D$ désigne sa racine, Nœuds(D) est l'ensemble de ses nœuds. Arcs(D) est l'ensemble des arcs de D.

XPath est un langage de requêtes qui désigne des ensembles de nœuds d'un document XML. Il a été créé pour définir une syntaxe et une sémantique aux fonctions communes à XPointer et XSL, mais il est un langage d'interrogation à part entière.

Pour désigner un nœud dans un document, XPath propose un langage d'adressage d'objets et un ensemble de fonctions permettant d'augmenter l'expressivité du langage.

Le langage d'adressage repose sur les notions d'axes et de sélection. Plus précisément, un chemin, qui peut être absolu ou relatif se décompose en :

- un axe de navigation, choisi parmi child (les fils), parent (les parents), following-sibling ou preceding sibling (les frères)...
- un test sur le type d'un nœud. Par exemple, le chemin `ancestor::chapitre` sélectionne les ancêtres de type chapitre ;
- un ou plusieurs prédicats. Par exemple, la requête suivante sélectionne le dernier des fils de type chapitre : `child::chapitre[position()=Last()]`.

Ces étapes se composent entre elles en utilisant l'opérateur "/". Par exemple, le titre du parent d'un nœud chapitre s'écrit `self::chapitre/parent::titre`.

Présentons les treize axes de navigation que nous utilisons dans cette thèse. Les onze premiers sont des axes qui existent dans la norme XPath. Les deux derniers ne sont pas

explicitement des axes XPath. Néanmoins, nous les utiliserons comme tels car on peut les définir en utilisant les attributs *id* et *href*.

Définition 2.2 *Soit $D = (A, ref)$ un document XML. Définissons treize relations sur les nœuds de D. Les onze premières se réfèrent uniquement à la structure d'arbre de D alors que les deux dernières n'utilisent que la relation ref.*

1. **s***elf met en relation un nœud avec lui-même,*
2. **c***hild met en relation un nœud avec ses fils,*
3. **d***escendent met en relation un nœud avec ses descendants*[2]*,*
4. **d***escendent-or-***s***elf est l'union de la relation* **d***escendent et de la relation* **s***elf*
5. **pa***rent met en relation un nœud avec son père,*
6. **a***ncestor met en relation un nœud avec ses ancêtres,*
7. **a***ncestor-or-***s***elf est l'union de la relation* **a***ncestor et de la relation* **s***elf,*
8. **p***receding met en relation un nœud et ses prédécesseurs. Un nœud n_1 est prédécesseur d'un nœud n_2 si la balise fermante de n_1 précède la balise ouvrante de n_2. Autrement dit, un nœud n est en relation avec tous les nœuds qui sont avant lui dans l'ordre (du parcours préfixe) du document, à l'exclusion de tout ancêtre*
9. **p***receding-***s***ibling met en relation un nœud avec ses frères gauchess,*
10. **f***ollowing met en relation un nœud avec ses successeurs. Un nœud n_1 est successeur d'un nœud n_2 si la balise fermante de n_2 précède la balise ouvrante de n_1. Autrement dit, un nœud n est en relation avec tous les nœuds qui sont après lui dans l'ordre (du parcours préfixe) du document, à l'exclusion de tout descendant.*
11. **f***ollowing-***s***ibling met en relation un nœud avec ses frères droits,*
12. *un couple de nœuds (n_1, n_2) appartient à la relation* **id***ref si (n_1, n_2) appartient à ref,*
13. **r***everse* **id***ref est la relation* $\mathbf{ir}^{-1}$

La syntaxe complète de XPath étant, par moment, trop lourde, la recommandation inclut une syntaxe abrégée. Par exemple, // est une abréviation de `descendent-or-self::node()`. On retrouve ainsi des idées semblables à celles de l'adressage du système de fichiers. Dans cette thèse, un choix un peu différent a été fait : les axes sont souvent nommés par leurs initiales.

Enfin, pour définir les prédicats, XPath propose un langage d'expressions, utilisant un ensemble de fonctions de base, telles la manipulation de chaînes de caractères (concat, contains, start-with...) ou l'étude de la position d'un nœud parmi ses frères (position, last, count...).

Le résultat d'une requête XPath est un ensemble de nœuds dans l'arbre XML du document, ensemble qui peut être réutilisé, par exemple par une requête XQuery.

XML Query ou XQuery est aussi une spécification du W3C. XQuery est un langage de requête permettant d'extraire des informations d'un document XML. Sémantiquement proche de SQL (il est souvent considéré comme un langage hybride à mi-chemin entre XPath et SQL), XML Query utilise la syntaxe XPath pour localiser des parties d'un document XML.

[2] descendent est le nom anglais de l'axe descendant.

XQuery permet d'écrire des requêtes plus complexes que XPath. Par exemple, XQuery permet de faire des requêtes imbriquées contenant des expressions FLWOR (elles assignent des valeurs à une ou plusieurs variables utilisées pour définir le résultat).

De plus, XQuery permet de faire des requêtes à la fois sur plusieurs sources XML et de créer un résultat XML. XQuery est donc plus qu'un langage de requêtes sur les arbres XML car il est un langage de transformation d'arbres.

2.2.2 Requête graphe sur un document XML coloré

Cette section donne la représentation d'un document XML coloré, introduit les requêtes graphes et définit le problème de l'évaluation.

Définition 2.3 *Soit Σ un alphabet de labels. Un document XML coloré est un triplet $D = \{N, \mathcal{C}, \{D_c\}_{c\in\mathcal{C}}\}$ tel que :*
- *N est un ensemble fini de nœuds étiquetés par un élément de Σ[3],*
- *$\mathcal{C}$ est un ensemble fini de couleurs,*
- *pour tout c de $\mathcal{C}$, D_c est un document XML.*
- *la famille $\{D_c\}_{c\in\mathcal{C}}$ est couvrante i.e. $N = \bigcup_{c\in\mathcal{C}} Nœuds(D_c)$.*

Plusieurs remarques sont à faire sur ce modèle :

Remarque 2.1
1. *Il n'y a aucune restriction sur les arbres considérés dans un document coloré. Par exemple, s'il existe une relation fils entre les nœuds n_1 et n_2 dans un arbre, il est possible d'avoir une relation fils entre n_2 et n_1 dans un autre arbre.*
2. *Si $\mathcal{C}$ est réduit à une couleur alors un document coloré est un document XML. Les requêtes pour les documents XML coloré peuvent donc être utilisées pour interroger des documents XML classiques.*
3. *Si les arbres d'un document D sont disjoints (les ensembles de nœuds de la famille $\{N_c\}_{c\in\mathcal{C}}$ sont deux à deux disjoints), alors D est un ensemble de documents XML.*

On définit maintenant les requêtes graphes. Pour cela, on se donne un ensemble de primitives qui contient un ensemble fini P de prédicats unaires et un ensemble fini R de relations binaires. P se décompose en deux sous ensembles disjoints : P_{absolu} et $P_{relatif}$

Soient Σ un ensemble fini de labels et $\mathcal{C}$ un ensemble fini de couleurs. L'ensemble $P_{\Sigma,\mathcal{C}}$ de primitives utilisées dans cette thèse est composé de l'ensemble de prédicats $P_{relatif} = \{*\} \cup \{\sigma, \sigma \in \Sigma\}$, $P_{absolu} = \{/_c, c \in 2^{\mathcal{C}}\}$ et de l'ensemble de relations $R = \{\{c\}axe \mid c \in 2^{\mathcal{C}} \wedge axe \in \{c, d, ds, pa, a, as, p, ps, f, fs, s, ir, rir\}\}$.

Une requête graphe est un graphe orienté dans lequel les arcs et les nœuds sont étiquetés :

Définition 2.4 *Soit $E = (P, R)$ un ensemble de primitives. Une requête graphe est un triplet $< N, A, \$ >$ où*

[3]En XML, l'étiquette d'un nœud correspond à une balise dans le texte XML. Cette étiquette est parfois appelée type du nœud

- *N désigne un ensemble de nœuds. L'étiquette d'un nœud v est un couple $(pred_v, bool_v)$. $pred_v$ est un prédicat appartenant à P. $bool_v$ est un opérateur booléen choisi parmi l'ensemble {and, or}*[4].
- *A est l'ensemble des arcs orientés (i.e. un sous ensemble de $N \times N$). L'étiquette d'un arc a est un élément de R qui est noté axe_a.*
- *$\$$ est un n-uplet de nœuds sélectionneurs (i.e. $\$$ est de la forme $(\$_1, \ldots, \$_n)$ avec $\$_i$ un nœud de N).*

En particulier, une requête DAG (Direct Acyclic Graph) est une requête graphe qui est acyclique et une requête arbre est une requête graphe dans laquelle (N,A) a une structure d'arbre.

Attention, deux graphes sont désormais utilisés. L'un représente un document XML coloré D. Ses nœuds sont soit étiquetés par un label d'un alphabet fini Σ soit une racine de l'un des arbres de D. Les arcs de D peuvent être des arcs *arbre* ou des arcs *références*. Dans les deux cas, ils ont une unique couleur. L'autre graphe est une requête. Ses nœuds sont étiquetés par un couple (prédicat unaire, opérateur booléen). Les étiquettes de ses arcs appartiennent à un ensemble de relations binaires.

Par commodité, dans les figures, les nœuds d'un graphe seront désignés par un identifiant numérique.

La réponse à une requête graphe $< N_q, A_q, \$_q >$, qui a pour primitive (P, R), sur le document $\{N, \mathcal{C}, \{D_c\}_{c \in \mathcal{C}}\}$ repose sur l'interprétation des prédicats de P en prédicat de $N \rightarrow bool$ et les relations de R en relation de $N \times N$.

L'interprétation des éléments de $P_{\Sigma,\mathcal{C}}$ est la suivante :
- $*$ est un prédicat qui est toujours vrai,
- σ est un prédicat qui est vrai pour les nœuds ayant le type σ,
- $/_c$ est un prédicat qui est vrai pour les nœuds n appartenant à l'ensemble $\{/_{couleur} \mid couleur \in c\}$,
- Si c est une couleur et a est un axe, alors la relation $\{c\}a$ contient tous les couples de nœuds (n_1, n_2) où n_2 est accessible depuis n_1 en suivant l'axe a[5] dans l'arbre de couleur c.
- Si c est un ensemble de couleurs alors $\{c\}a$ est l'union de toutes les relations $\{couleur\}a$ pour *couleur* une couleur de c. Quand $\{D_c\}$ est un singleton, cette relation est simplement notée a.

Dans un deuxième temps, il faut définir les nœuds d'un document qui sont sélectionnés par une requête. Dans ce but, on utilise les notations suivantes :
- pour tout prédicat p de P, pour tout ensemble de nœuds N' inclus dans N, $p(N')$ désigne l'ensemble des nœuds de N' pour lesquels le prédicat p est vrai (i.e. $p(N') = \{n' \in N' \mid p(n')\}$).
- Si r est une relation de R alors $r[N']$ désigne l'ensemble des nœuds n pour lesquels il existe un nœud n' dans N' tel que (n', n) soit dans la relation r (i.e. $r[N'] = \{n \in N \mid \exists n' \in N', (n', n) \in r\}$).

[4] Afin de simplifier les schémas, on considère que la valeur par défaut est le *and*

[5] Les axes sont représentés par leur initiale ; cf. définition 2.2

Définition 2.5 *Une fonction β des nœuds d'une requête vers les nœuds d'un document est un plongement si*

- *Pour tout nœud v de la requête, si $pred_v$ appartient à P_{absolu}, alors $\beta(v)$ est défini.*
- *Pour tout sélectionneur $\$_i$, $\beta(\$_i)$ est défini.*
- *Pour tout nœud v de la requête tel que $\beta(v)$ est défini*
  1. *β satisfait les prédicats i.e. $pred_v(\beta(v))$ est vrai.*
  2. *β préserve les arcs sortants et les opérateurs booléens :*
     - *Si bool_v l'opérateur booléen associé à v est un* and *alors pour tout arc sortant $a = (v, v')$ on doit avoir $\beta(v')$ dèfini et qui appartient à $axe_a[\beta(v)]$*
     - *Si bool_v l'opérateur booléen associé à v est un* or *alors il existe un arc sortant $a = (v, v')$ pour lequel $\beta(v')$ dèfini et appartient à $axe_a[\beta(v)]$*

On peut définir le résultat d'une requête sur un document comme suit :

Définition 2.6 *Soient D un document XML coloré et $q = (N, A, (\$_{q_1} \dots \$_{q_n}))$ une requête graphe, le résultat de q sur D est l'ensemble de n-uplets nœuds*

$$D(q) = \bigcup_{\beta : q \to D} \{(\beta(\$_{q_1}), \dots, \beta(\$_{q_n}))\}$$

β est une application partielle de q vers D car il peut exister des nœuds de q pour lesquels β n'est pas défini. En effet, si un noeud est étiqueté par un *or* alors β peut ne pas être défini sur un sous-graphe de q. En particulier, dans le résultat ci-dessus, il est possible qu'il n'existe aucun plongement entre la requête et le document. Dans ce cas $D(q)$ est l'ensemble vide. Au contraire, si β est défini pour tous les nœuds de q, β est dite totale.

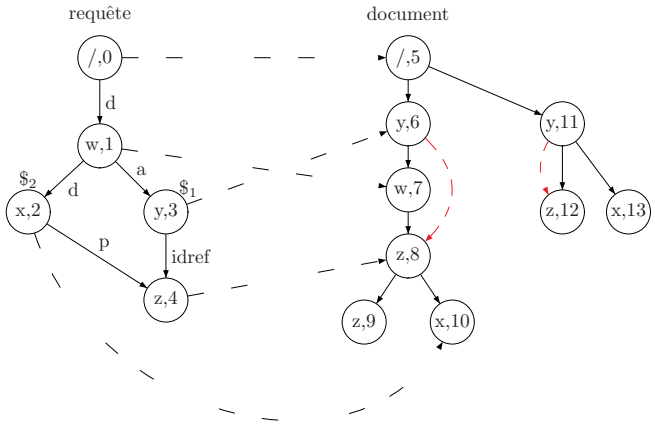


FIG. 2.5 – Plongement d'une requête graphe dans un document XML

Exemple 2.2.1 : La figure 2.5 représente une requête graphe q, un document D (un document XML à une couleur), et l'unique plongement de q vers D.

Décrivons le document : la racine du document est le nœud 5. Les nœuds peuvent être étiquetés par les lettres {w,x,y,z }. Les arcs décrivant l'arbre XML sont représentés par des arcs pleins, les arcs de l'ensemble ref sont en pointillés.

Décrivons la requête : la requête, dont tous les opérateurs sont des *and*, a deux sélectionneurs $\$_1$ et $\$_2$ (nœuds 3 et 2). Les nœuds et les arcs sont étiquetés par les primitives suivantes : /, w, x, y et z pour les prédicats et a, d, p et idref pour les relations. Interprétons ces primitives :

Le prédicat w (rep. x, y, z) est vrai pour tous les nœuds du graphe qui ont l'étiquette w (resp. x, y, z) et le prédicat / désigne la racine du document. Les relations permettent de naviguer dans le document en suivant les arcs du document : p met en relation un nœud avec son père, idref est l'ensemble $\{(n, n') \mid (n, n') \in ref\ \}\ldots$

L'unique plongement de la requête dans le document apparaît en pointillés entre la requête et le document. Comme $(3, 4)$ est un arc de la requête étiqueté par idref on a un arc $(\beta(3), \beta(4)) = (6, 8)$ de type référence dans le document. Comme $(1, 3)$ est un arc étiqueté par a, on a $\beta(3) = 6$ qui est un *ancêtre* de $\beta(1) = 7$ et ainsi de suite pour tous les arcs de la requête. La requête est binaire, son résultat est donc le couple $\{(\beta(3), \beta(2)\}$ qui vaut $\{(6, 10)\}$.

Le nœud 11 n'ayant pas de *descendant* de type w, une fonction définie par $\beta(3) = 11$ ne peut pas être un plongement.

Le but de l'exemple suivant est de montrer un plongement partiel d'une requête dans un document :

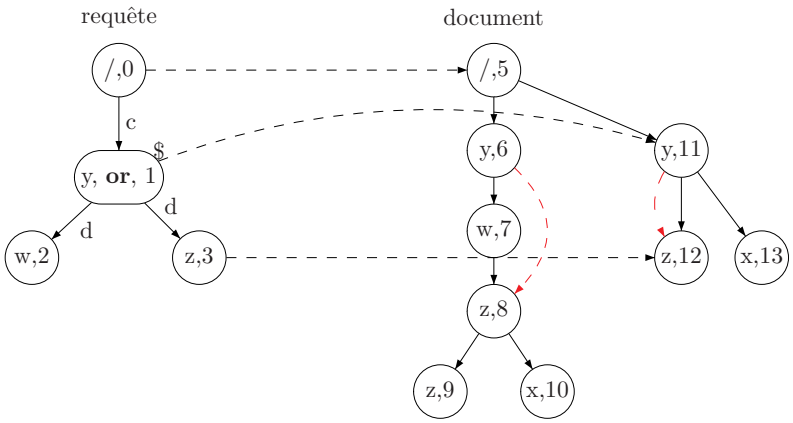


FIG. 2.6 – Plongement partiel d'une requête graphe dans un document XML

Exemple 2.2.2 : Reprenons le document de l'exemple 2.2.1 et considérons la requête disjonctive de la figure 2.6. Cette requête sélectionne les nœuds 6 et 11. En effet, le plongement partiel représenté en pointillés dans la figure 2.6 sélectionne le nœud 11.

Plusieurs plongements permettent de sélectionner le nœud 6. Tous ces plongements β ont en commum les valeurs $\beta(0) = 5$ et $\beta(1) = 6$. Par contre il y a plusieurs valeurs possibles pour $\beta(2)$ et $\beta(3)$. Si $\beta(u) = \perp$ signifie que le u n'est pas plongé alors $(\beta(2), \beta(3))$ peut prendre les valeurs suivantes :

- $(7, \perp)$ est un plongement partiel.
- $(7, 8)$ est un plongement total.
- $(\perp, 9)$ est un plongement partiel.

– $(\perp, 8)$ est un plongement partiel.
– $(7, 9)$ est un plongement total.

Pour terminer cette étude, définissons le problème de l'évaluation. Plusieurs questions sont intéressantes à résoudre :
– Comment calculer l'ensemble $D(q)$ des nœuds de D sélectionné par q ?
– L'ensemble $D(q)$ est-il vide (problème du *model checking*) ?
– Étant donné un n-uplet τ de nœuds du document, τ appartient-il à $D(q)$?

Les documents XML coloré étant finis, le troisième problème permet de résoudre les deux autres. Par contre, les résultats de complexité ne sont pas forcément les mêmes.

Définition 2.7 *Soit E un ensemble de primitives.*

Le problème de l'évaluation se définit par :
Entrée : une requête graphe q utilisant les primitives de E, un document D et un n-uplet τ de nœuds de D
Sortie : vrai si et seulement si le n-uplet τ est sélectionné par q

Afin d'étudier la complexité du problème, il est utile de préciser la taille de ses paramètres. La taille d'un graphe G (une requête ou un document) est définie par le somme de son nombre d'arcs et de son nombre de nœuds. Un n-uplet de nœuds a une taille n.

2.3 Deux exemples de requêtes graphes : le Core XPath et les requêtes conjonctives

Avant d'étudier les requêtes graphes dans toute leur généralité, on présente deux exemples de la littérature. Le premier consiste à considérer des requêtes arbres. Pour cela, on étudie un fragment de XPath : le Core XPath [Gottlob et al., 2002]. Le second exemple est consacré aux requêtes conjonctives pour interroger les documents XML [Gottlob et al., 2004]. Contrairement aux requêtes du Core XPath, les requêtes conjonctives peuvent se représenter par un graphe cyclique. Par contre, tous les opérateurs booléens utilisés seront des *and*.

Une différence existe entre la présentation que nous faisons ici et celle d'origine. Nous étendons en effet les résultats de [Ramanan, 2003] et de [Gottlob et al., 2004] au cas des documents XML coloré. Nous montrons que cette extension se fait sans modification de la complexité des algorithmes d'évaluation.

2.3.1 Le Core XPath

Complétons la présentation générale de XPath et de XQuery de la section 2.2.1 avec des résultats de complexité : l'évaluation d'une requête XPath est PTIME-dur [Gottlob et al., 2003] tandis que XQuery (ainsi que XSLT) est Turing complet [Kepser, 2002]. Dans le but d'avoir des algorithmes d'évaluation efficaces, on travaille uniquement sur des fragments de ces langages.

L'un des problèmes étudiés jusqu'à maintenant est de trouver le fragment de XPath le plus large possible qui ait de bons algorithmes d'évaluation. Trois fragments de XPath ont été définis dans ce but. Pour ces trois fragments, l'évaluation est linéaire dans la taille combinée du document et de la requête. On les présente ici par expressivité décroissante :

1. G. Gottlob, C. Koch et R. Picler ont défini dans [Gottlob et al., 2002] un fragment de XPath dans lequel le calcul est linéaire dans la taille combinée du document D et de la requête q (i.e. le calcul de $D(q)$ est $O(|D|.|q|)$). Ce fragment, le Core XPath, contient toutes les possibilités de navigation de XPath. On peut naviguer dans un document suivant les onze axes classiques (*self, child, descendent, descendent-or-self, parent, ancestor, ancestor-or-self, preceding, preceding-sibling, following et following-sibling*). Ces travaux s'étendent facilement (voir par exemple [Ramanan, 2003]) si on considère les axes idref et ridref (pour prendre en compte les références). Les prédicats du Core XPath se combinent avec les opérateurs *and*, *or* et *not*.

   Sans être exhaustif, donnons quelques exemples de requêtes XPath qui ne s'expriment pas avec le Core XPath : sélectionner les nœuds qui ont une position paire sous leur père (`[position() mod 2 = 0]`), sélectionner les nœuds qui ont plus de fils étiqueté par a que de fils étiqueté par b (fonction *count*), accéder au contenu d'un nœud... Enfin, une requête du Core XPath ne permet pas d'utiliser l'axe 'attribut' de XPath.
2. R .Kaushik et al. ont défini dans [Kaushik et al., 2002a] un fragment du Core XPath : BPQ (Branching Path Queries) dans lequel l'ordre des frères n'a pas d'importance. En effet seuls neuf des treize axes sont autorisés : les axes faisant appel à l'ordre des nœuds (*preceding, preceding-sibling, following et following-sibling*) ne sont pas considérés dans les requêtes BPQ.
3. Enfin un fragment de BPQ a été introduit par Amer-Yahia et al. dans [Amer-Yahia et al., 2001]. Ce sont les TPQ (Tree Pattern Queries) aussi connus sous le nom de twig. Les twigs sont des requêtes conjonctives et n'autorisent que l'utilisation des trois axes verticaux descendant :*self, child, descendent-or-self.* Signalons enfin les auteurs de [Jiang et al., 2004] enrichissent les requêtes twig avec l'opérateur booléens *or* et que les auteurs de [Lu et al., 2005] définissent des requêtes twig ordonnés.

Ces trois fragments de XPath ont la même complexité (pour le problème de l'évaluation) mais ils se différencient par les algorithmes d'optimisation de requêtes ([Amer-Yahia et al., 2001], [Bruno et al., 2002], [McHugh and Widom, 1999]) et les algorithmes d'indexation ([Ramanan, 2003], [Kaushik et al., 2002a]).

A partir du Core XPath, le plus expressif de ces trois fragments, le Core XPath coloré est défini. Ce langage permet de sélectionner des nœuds en naviguant dans un document XML coloré. Pour cela, chaque axe de navigation est préfixé par un ensemble de couleurs d'arbre auxquels il s'applique.

La table 2.1 montre qu'une requête du Core XPath[6] coloré est de la forme $/_c ch_1/ch_2/\ldots$ Chaque *ch* représente un chemin (*location steep*). Le $/_c$ initial signifie que la requête est absolue (i.e. considérée à partir de l'une des racines des arbres désignée par l'ensemble de couleurs c). Une requête de la forme $ch_1/ch_2/\ldots$ est relative à un contexte qu'il faut spécifier. Chaque *ch* est de la forme $\{c\}$axe : :test-noeud[predicat$_1$][predicat$_2$]... c est un ensemble de couleurs et

[6]Le Core XPath défini dans [Gottlob et al., 2002] contient l'opérateur *not*. Pour cela, on ajoute la règle predicat $\rightarrow$ not(predicat).

requêteCoreXPath	→	requêteAbsolue | requêteRelative
requêteAbsolue	→	$/_c$ requêteRelative
requêteRelative	→	ch | ch / requêteRelative
ch	→	**{**c**}axe::nt** predicats
predicats	→	**[**predicat**]** predicats | ϵ
predicat	→	predicat **and** predicat | predicat **or** predicat | requêteRelative

TAB. 2.1 – Description du Core XPath coloré

axe se réfère à l'un des treize axes. Deux *test-noeud* sont possibles : soit c'est le symbole $*$, soit c'est un type. Il peut y avoir zéro ou plusieurs prédicats. Chaque prédicat est soit une combinaison booléenne de prédicats (*and* et *or*) soit une expression du Core XPath relative.

Répondre à une requête du Core XPath coloré sur un document XML D consiste à calculer à partir d'un ensemble de nœuds (un ensemble de racines du document D pour une requête absolue $/_c$) l'ensemble de nœuds réponses à la requête. A partir d'un contexte on applique chaque chemin ch_i de gauche à droite. Appliquer un $ch_i = \{c\}axe$ :: test-noeud$[predicat_1][predicat_2]\ldots$ à un contexte c_0 consiste à calculer un nouveau contexte en

- Calculant c_1 l'ensemble des nœuds de D accessibles depuis un nœud de c_0 en suivant l'axe *axe* dans l'un des arbres de l'ensemble c.
- Sélectionnant dans c_1 l'ensemble c_2 des nœuds qui passent le test *test-noeud*. Si c'est le symbole $*$, dans ce cas tous les nœuds passent le test, si c'est un type de Σ, dans ce cas seuls les nœuds correctement typés passent le test.
- Sélectionnant dans c_2 l'ensemble c_3 des nœuds qui satisfont tous les prédicats. Un nœud n satisfait un prédicat réduit à une requête relative r si le résultat de r, évaluée dans le contexte réduit au nœud n, est non vide.
- Le nouveau contexte est l'ensemble c_3.

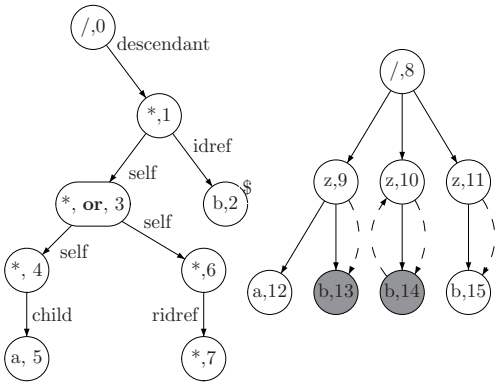


FIG. 2.7 – Arbre correspondant à la requête /d : :*[c : :a **or** ridref : :*]/idref : :b

Exemple 2.3.1 : La figure 2.7 contient un exemple de document XML à une couleur dont les nœuds 13 et 14 ont été grisés. Ces nœuds sont en effet solution de la requête du

Core XPath `/d::*[c::a or ridref::*]/idref::b`. Le nœud 13 est sélectionné car il est accessible depuis 9 en suivant une référence et que 9 a un fils étiqueté par a ; le nœud 14 est sélectionné car il est accessible depuis le nœud 10 par une référence et car 10 est lui-même accessible par une référence. Bien qu'il existe une référence du nœud 11 vers le nœud 15, 15 n'est pas sélectionné par la requête. En effet 11 n'a ni fils étiqueté par a, ni arc référence entrant.

Dans quel cas peut-on représenter une requête du Core XPath coloré par une requête graphe ? La proposition suivante répond à cette interrogation et permet même de restreindre la forme de la requête.

Proposition 2.1 *Toute requête du Core XPath coloré se représente avec une requête arbre définie avec les primitives de* $P_{\Sigma,\mathcal{C}}$.

Remarque 2.2 *Ce résultat étend une proposition de [Ramanan, 2003] du XML classique vers les documents XML coloré. Néanmoins, les techniques utilisées dans cette section sont très fortement inspirées de cet article.*

L'algorithme qui construit une requête arbre à partir d'une requête du Core XPath coloré (requête unaire et sans négation) est présenté dans la table 2.2. Cet algorithme travaille en deux phases :

1. Dans un premier temps, on ne considère pas les prédicats de la requête. On construit alors le tronc de l'arbre et on identifie le nœud $.
2. Dans un deuxième temps, la requête est considérée avec ses prédicats. On construit un arbre par prédicat, et on ajoute un axe *self* entre la racine de cet arbre et le nœud du tronc correspondant au prédicat.

Exemple 2.3.2 : La figure 2.7 donne un exemple d'un tel arbre pour la requête */d : :*[c : :a* **or** *ridref : :*]/idref : :b*. L'indice de chaque nœud correspond à l'ordre dans lequel l'algorithme l'a construit.

La branche du nœud 0 au nœud 2 correspond au tronc de la requête (i.e. /d : :*/idref : :b). Le nœud 2 est donc marqué par le $. C'est lui qui désigne les nœuds à sélectionner dans le document.

On construit ensuite le nœud 3 comme racine du prédicat [c : :a **or** ridref : :*]. Le prédicat étant un or, on construit deux fils au nœud 3 : un fils par opérande du or.

Théorème 2.1 *Soit q une requête du Core XPath coloré et D un document XML. Il existe un algorithme qui ait un temps de calcul $O(|q|.|D|)$ et qui évalue $D(q)$.*

Remarquons l'intérêt pratique de ce résultat. Les auteurs de [Gottlob et al., 2002] ont montré que trois processeurs commerciaux pour XML (Xalan, XT et Microsoft Internet Explorer) avaient des algorithmes exponentiels ($O(|D|^{|Q|})$) pour calculer les requêtes du Core XPath.

Ce bon résultat provient du fait qu'une requête du Core XPath peut se représenter par un arbre (proposition 2.1) et qu'elle est unaire (i.e. elle sélectionne des ensembles de nœuds).

CoreXPath2arbre(q)
% Entrée : une requête CoreXPath coloré q
% Sortie : une requête arbre correspondant à q
% Locale : i et j deux itérateurs
si q est absolue alors
  créer un nœud n_0 ($/_c$,and)
si q est relative alors
  créer un nœud n_0 (*,and)
% n_0 est la racine de la requête
% primaire$_q$:= q sans ses prédicats
% primaire$_q$ est de la forme axe$_1$: :nt$_1$... axe$_k$: :nt$_k$
pour i compris entre 1 et k faire
  construire un nœud $n_i = (nt_i, and)$
  ajouter un arc axe$_i$ entre n_{i-1} et n_i
fin pour
le nœud n_k est le nœud désigné $
% ajout des prédicats
% q est la forme $ch_1...ch_l$
pour i compris entre 1 et l faire
  % ch_i est de la forme $axe_i :: nt_i[pred_1]...[pred_m]$
  pour j compris entre 1 et m
    construire l'arbre a_j **predicat2arbre**(predicat$_j$)
    ajouter un arc *self* entre n_i et la racine de a_j
  fin pour
fin pour
retourner requête arbre de racine n_0
et de sélectionneur n_n

predicat2arbre(pred)
% Entrée : un predicat de CXpath coloré
% Sortie : un arbre représentant le prédicat
% Locale : $\emptyset$
si pred est une requête relative alors
  retourner **CoreXPath2arbre**(pred)
fin si
si pred est de la forme pred$_1$ op pred$_2$ alors
  créer un nœud n $(*, op)$
  construire a_1 = **predicat2arbre**($pred_1$)
  construire a_2 = **predicat2arbre**($pred_2$)
  ajouter un arc *self* entre n et a_1
  ajouter un arc *self* entre n et a_2
  retourner Arbre de racine n
fin si

TAB. 2.2 – Transformation d'une requête Core XPath en requête arbre

Ce théorème est bien connu pour les documents XML non coloré. Il en existe au moins trois preuves : [Gottlob et al., 2002] et [Marx, 2004] pour les arbres XML ; [Ramanan, 2003] pour les arbres XML avec des références. Chaque preuve a sa spécificité. Par exemple Gottlob et Al. prennent en compte la négation, les requêtes définies par M. Marx utilisent des boucles *while*... On se propose ici de s'inspirer de la preuve de P. Ramanan qui est basée sur la notion de simulation [Milner, 1980].

Une relation de simulation est souvent utilisée pour définir une notion de dominance (d'équivalence) entre deux graphes. L'une des utilisations les plus connues de la simulation concernent la vérification : un système G_1 raffine (implémente) une spécification G_2 si tout état initial de G_1 est simulé par un état initial de G_2. Dans le domaine des documents semi-structurés, cette notion est utilisée lorsque l'on parle de schéma (i.e. un document est valide par rapport à un schéma s'il existe une simulation entre le document et le schéma) ; cette idée est reprise dans le chapitre consacré à l'indexation. Dans cette section, on simule la requête sur le document pour calculer les nœuds sélectionnés par une requête.

Définition 2.8 *Une forward simulation $\prec$ de la requête $q =< N_q, A_q, \$_q >$ sur le document $D =< N_d, \mathcal{C}, \{D_c\}_{c\in\mathcal{C}} >$ est une relation sur $N_q \times N_d$ qui vérifie pour tout couple (v, n) de $\prec$:*

1. *Préservation des booléens : $pred_v(n)$ est vrai*

2. *Préservation des opérateurs booléens et des arcs sortants*
   - *Si* bool_v *le booléen associé à* v *est un* and *alors pour tout arc* $r = (v, v')$ *il existe un nœud* n' *tel que* n' *appartienne à* $\text{axe}_r[n]$ *et* $v' \prec n'$
   - *Si* bool_v *le booléen associé à* v *est un* or *alors il existe un arc* $r = (v, v')$ *il existe un nœud* n' *tel que* n' *appartienne à* $\text{axe}_r[n]$ *et* $v' \prec n'$

Notons maintenant $\prec_{Fs}$ la plus grande forward simulation (l'union de toutes les forward simulations) de la requête sur le document. Pour tout nœud v de la requête on note $\text{Fsim}_D(v)$ l'ensemble des nœuds n du document tel que $v \prec_{Fs} n$. Grâce aux ensembles Fsim_D, on a une première information concernant la requête. En effet, s'il n'y a aucun nœud en F-simulation avec la racine de la requête on sait alors que la réponse à la requête est vide. Par contre, la F-simulation n'est pas suffisante pour calculer exactement $D(q)$: on constate que l'on peut calculer $\text{Fsim}_D(v)$ en ne prenant pas en compte la manière dont on peut plonger dans D le contexte du nœud v (i.e. le calcul ne dépend que du sous-arbre de la requête dont v est la racine).

Dans l'exemple 2.3.1 (page 25), $\text{Fsim}_D(2)$ est égale à {13,14,15} ($\text{Fsim}_D(2)$ est l'ensemble de tous les nœuds de type b). Pourtant le nœud 15 n'est pas solution de la requête. C'est pourquoi la F-simulation nous garantit l'inclusion de $D(q)$ dans $\text{Fsim}_D(\$_q)$ mais pas la réciproque. On étend donc notre notion de simulation et on considère la Forward and Backward simulation (FB-simulation).

Une Backward simulation $\prec_{Bs}$ de la requête $q =< N_q, A_q, \$_q >$ sur le document $D =< N_d, \mathcal{C}, \{D_c\}_{c\in\mathcal{C}} >$ est une relation qui vérifie : pour tout couple (v, n) de $\prec_{Bs}$, si $r = (v', v)$ est l'arc entrant dans v il existe alors un nœud n' tel que n appartienne à axe_r[n'] et $v' \prec_{Bs} n'$.

La Forward and Backward simulation $\prec_{FBs}$ de la requête $q =< N_q, A_q, \$_q >$ sur le document $D =< N_d, \mathcal{C}, \{D_c\}_{c\in\mathcal{C}} >$ est la plus grande relation sur $N_q \times N_d$ qui soit une Forward simulation et une Backward simulation :

Comme dans le cas de la F-simulation, on note $\text{FBsim}_D(v)$ l'ensemble des nœuds n du document D tel que $v \prec_{FBs} n$. Contrairement à la F-simulation on peut calculer D(q) à partir de la FB-simulation.

La FB-simulation d'une requête du Core XPath coloré sur un document XML coloré vérifie une propriété très intéressante :

Lemme 2.1 *Considérons la FB-simulation d'une requête* q *monadique représentant une requête du Core XPath coloré sur un document XML coloré* D*. Soient* v *un nœud de* q *et* n *un nœud de* D*. Les deux propositions suivantes sont équivalentes :*

1. n *appartient à* $FBsim(v)$*.*
2. *il existe un plongement* β *de* Q *vers* D *tel que*
   - $\beta(v) = n$
   - *Pour tout ancêtre* v' *de* v, $\beta(v')$ *est défini.*

Preuve : Si β *est un plongement d'une requête arbre* q *sur un document* D *alors* β_c *est la restriction de* β *qui vérifie : pour tout nœud* v *de* q *si* $\beta_c(v)$ *est défini alors* β_c *est défini pour*

tous les ancêtres de v. Autrement dit, on ne considère que la partie de β qui est connectée à la racine de la requête.

L'ensemble $\{n \mid \exists\beta\ \beta_c(v) = n\}$ est inclus dans $FBsim_D(v)$. En effet, si β est un plongement, alors la relation $\{(v, \beta_c(v)) \mid \beta_c(v)$ est défini$\}$ est une forward and backward simulation.

Si n appartient à $FBsim(v)$ construisons par induction un plongement β tel que $\beta(v) = n$. Dans ce but, décrivons les trois algorithmes suivants :

*1)***construirePlongement***(v,n,$FBsim_D$)*
% Entrée : la FB-simulation d'une requête q sur un document D,
% v un nœud de q, n un nœud de D appartenant à $FBsim_D(v)$.
% Sortie : un plongement β de q dans D tel que $\beta(v) = n$
% Locale : $\emptyset$
sous-arbre*(v,n,$FBsim_D$)*
contexte*(v,n,$FBsim_D$)*

2) **sous-arbre***(v,n,$FBsim_D$)*
% construit un plongement β tel que $\beta(v) = n$ sur le sous-arbre de q dont v est la racine
si $\beta(v)$ n'est pas défini alors
  $\beta(v) = n$
  si $bool_v$ est un and alors
    pour tout fils v' de v faire
      choisir n' dans $axe_{(v,v')}[\{n\}]$
      sous-arbre*(v',n',$FBsim_D$)*
    fin pour
  fin si

  si $bool_v$ est un or alors
    il existe un fils v' de v tel que $axe_{(v,v')}[\{n\}]$ soit non vide
    soit n' un nœud de cet ensemble
    sous-arbre*(v',n',$FBsim_D$)*
  fin si
fin si

3) **contexte***(v,n,$FBsim_D$)*
% construit un plongement β tel que $\beta(v) = n$ dans le contexte de v
si v a un père v' alors
  Soit n' un nœud tel que n appartienne à $axe_{(v',v)}[\{n'\}]$
  contexte*(v',n',$FBsim_D$)*
  sous-arbre*(v',n',$FBsim_D$)*
fin si

Il faut montrer que le β construit ci-dessus est un plongement. Remarquons d'abord que la définition de Forward and Backward simulation garantit que tous les appels à **sous-arbre** *et à* **contexte** *sont valides (i.e. pour tous les appels à une procédure, n appartient à $FBsim_D(v)$). Il nous faut encore dire que :*

1. *$FBsim_D$ étant une (Forward) simulation, si $\beta(v)$ est défini alors $pred_v(\beta(v))$ est vrai.*
2. *$FBsim_D$ étant une Forward simulation, β préserve les arcs sortants.*

3. *FBsim$_D$ étant une Backward simulation, pour tout ancêtre v' de v $\beta(v')$ est défini. En particulier, $\beta(racine(q))$ existe.*
4. *La requête q étant construite à partir du Core XPath coloré, le nœud \$ de q est particulier : tous ses ancêtres sont étiquetés par l'opérateur and. On déduit de l'item précédent que $\beta(\$_q)$ est défini.*

◀

Tous les ancêtres du nœud $\$_q$ sont étiquetés par l'opérateur *and*. On a donc : pour tout plongement β et pour tout nœud v apparaissant dans le chemin entre la racine q et le nœud \$, $\beta(v)$ est défini. On peut donc spécialiser le lemme 2.1 au nœud $\$_q$, et obtenir le théorème suivant :

Théorème 2.2 *Soient $q =< N, A, \$ >$ une requête arbre monadique représentant une requête du Core XPath coloré et D un document XML coloré, la FB-simulation de q par D vérifie :*

$$D(q) = FBsim_D(\$_q)$$

Les auteurs de [Henzinger et al., 1995] et de [Bloom and Paige, 1995] proposent des algorithmes pour trouver les simulations d'un graphe par un autre. Nous proposons un algorithme en deux phases qui exploitent la structure d'arbre de la requête. On utilise des techniques qui ressemblent à des automates d'arbres [Comon et al., 1997]. Dans une phase montante on calcule les ensembles Fsim$_D$. On en déduit, dans une phase descendante, le calcul des ensembles FBsim$_D$:

1. Calculer Fsim$_D(v)$ pour tous les nœuds v de la requête en utilisant une méthode ascendante sur la requête du Core XPath (représenté par un arbre) :
   • Pour toute feuille v (pas d'arc sortant), si bool$_v$ est un *and* alors Fsim$_D(v)$ = pred$_v(N_d)$ sinon Fsim$_D(v)$=$\emptyset$.
   • Pour tout autre nœud, si *bool*$_v$ est un *and* alors

$$Fsim_D(v) = pred_v(\bigcap_{a=(v,v')\in Arcs(q)} axe_a^{-1}[Fsim_D(v')])$$

   sinon *bool*$_v$ est un *or* alors

$$Fsim_D(v) = pred_v(\bigcup_{a=(v,v')\in Arcs(q)} axe_a^{-1}[Fsim_D(v')])$$

2. Calculer, symétriquement, FBsim$_D(v)$ pour tous les nœuds v de la requête en utilisant une méthode descendante sur la requête.

Pour la racine v' de la requête FBsim$_D(v')$ = Fsim$_D(v')$.

Pour tout autre nœud v', soit v le père de v',

$$FBsim_D(v') = Fsim_D(v') \cap \text{axe}_{(v,v')}[FBsim_D(v)]$$

Exemple 2.3.3 : [suite de l'exemple 2.3.1 ; figure 2.7]

1. Phase ascendante : $Fsim_D(7) = \{8,9,10,11,12,13,14,15\}$, $Fsim_D(6) = \{10,13,14,15\}$, $Fsim_D(5) = \{12\}$, $Fsim_D(4) = \{9\}$,$Fsim_D(3) = \{9,10,13,14,15\}$, $Fsim_D(2) = \{13,14,15\}$, $Fsim_D(1) = \{9,10\}$ et enfin $Fsim_D(0) = \{8\}$.
2. Phase descendante : $FBsim_D(0) = \{8\}$, $FBsim_D(1) = \{9,10\}$, $FBsim_D(2) = \{13,14\}$, $FBsim_D(3) = \{9,10\}$, $FBsim_D(4) = \{9\}$, $FBsim_D(5) = \{12\}$, $FBsim(6) = \{10\}$, et enfin $FBsim_D(7) = \{14\}$.

Le nœud marqué par le \$ étant le nœud 2 on a d'après le théorème 2.2 que $D(q)$ est égal à $\text{FBsim}_D(\$_q)$ i.e. à l'ensemble $\{13,14\}$.

Pour conclure cette section, il faut montrer que cet algorithme est linéaire en data-complexité. Pour cela, nous donnons un résultat plus général que celui du théorème 2.1.

Théorème 2.3 *Soient (P,R) un ensemble de primitives, q une requête arbre utilisant ces primitives et D un document XML coloré dont l'ensemble de nœuds est N. Si*

1. *Pour tout prédicat p de P, tout sous ensemble N' de N on peut évaluer $p(N') = \{n' \mid n' \in N' \wedge p(n')\}$ en $O(|D|)$*
2. *Pour toute relation r de R, tout sous ensemble N' de N on peut évaluer $r[N'] = \{n \in N \mid \exists n' \in N \wedge (n',n) \in R\}$ en $O(|D|)$.*

alors on peut calculer $D(q)$ avec un algorithme en $O(|D|.|q|)$

Comme pour tout ensemble N' de nœuds de D et pour chaque relation r on peut calculer r[N'] en $O(|D|)$ la phase ascendante peut être exécutée en un temps $O(|q||D|)$ car l'algorithme utilise exactement une fois chaque arc et chaque nœud de la requête. De même, chaque $FBsim_D(v)$ de la phase descendante peut être calculé avec un algorithme $O(|D|)$. A condition d'utiliser une structure de données qui permette de tester en $O(1)$ si un élément appartient à un ensemble, cette seconde étape est en $O(|D||q|)$. L'algorithme en entier a donc la même complexité.

Il est connu de P. Ramanan [Ramanan, 2003] et de G. Gottlob et al. [Gottlob et al., 2002] que pour tout document XML D, tous les axes du Core XPath (y compris les idref et ridref) peuvent être évalués en $O(|arcs(D)|)$. Un arc d'un document coloré appartenant à exactement un arbre, on peut calculer pour un ensemble de couleurs c, $\{c\}axe$ en $O(|D|)$. On instancie donc le théorème 2.3 au cas des requêtes arbres construites à partir du Core XPath (sans négation) pour montrer le théorème 2.1.

2.3.2 Requêtes conjonctives

Cette section est consacrée aux requêtes conjonctives pour interroger les documents XML [Gottlob et al., 2004]. Contrairement aux requêtes du Core XPath, qui sont des requêtes arbres, certaines requêtes conjonctives nécessitent un graphe cyclique pour être représentées.

Bien que les requêtes conjonctives [Abiteboul et al., 1995] ne soient pas très expressives, elles correspondent en effet aux requêtes les plus fréquentes sur les bases de données i.e. les requêtes SQL (select-from-where) qui ne contiennent que des conjonctions.

Du point de vue théorique, on peut par exemple signaler que tester si deux requêtes conjonctives sont équivalentes est décidable. D'un coté plus pratique, il existe de nombreux algorithmes d'optimisation (réécriture de requêtes) tels que l'algorithme *chase*.

Dans un premier temps nous allons présenter formellement les requêtes conjonctives. Puis nous expliquons comment G. Gottlob, C. Koch et K. Schulz utilisent des requêtes conjonctives pour interroger des documents XML [Gottlob et al., 2004].

Présentation des requêtes conjonctives

Pour commencer nous donnons la définition d'une requête conjonctive :

Définition 2.9 *Soit S un ensemble de relations. Une requête conjonctive q sur S est une expression de la forme :*

$$ans(u) = R_1(u_1), \ldots, R_n(u_n)$$

où les R_i sont des relations dans S, ans est un nom de relation qui n'est pas dans S et u, $u_1, \ldots, u_n$ sont des tuples libres (i.e. on peut les utiliser soit avec des variables soit avec des constantes). De plus toute variable de u doit apparaître dans au moins l'un des u_i.

Dans la suite de cette thèse, on note $T(q)$ l'ensemble des tuples utilisés par la requête q. L'ensemble des variables présentes dans un tuple u_i est noté par $var(u_i)$.

Si par exemple S contient la relation binaire père, on peut définir la requête conjonctive `GrandPère(X,Y)` par `père(X,Z),père(Z,Y)`. Si notre schéma contient une relation ternaire films (titre, directeur, acteur), une relation binaire pariscope (cinéma,titre) et une relation ternaire localisation (cinéma, adresse, téléphone) on pourra écrire une requête conjonctive qui trouve l'adresse de tous les cinémas qui passent un film de 'Bergman' :

$$\text{q(a)} = \text{films(t,'Bergman',\_),Pariscope(c,t),localisation(c,a,\_).}$$

Le symbole _ est un joker. Il signifie que certaines valeurs des relations n'ont pas d'influence sur le résultat final. Les variables t et c apparaissent dans deux relations différentes. Elles permettent de calculer la jointure entre ces relations.

Répondre à la requête conjonctive $ans(e_1, \ldots, e_m) = R_1(u_1), \ldots, R_n(u_n)$ revient à calculer l'ensemble $\{(e_1, \ldots, e_m) \mid \exists x_1, \ldots, x_k R_1(u_1), \ldots, R_n(u_n)\}$. Les x_i sont des variables qui apparaissent dans le corps de la règle (i.e. $R_1(u_1), \ldots, R_n(u_n)$) mais pas dans la tête (i.e. ans$(e_1, \ldots, e_m)$).

Une sous-famille des requêtes conjonctives est très souvent étudiée : les requêtes conjonctives acycliques. Cette notion est basée sur la notion d'hypergraphe : l'hypergraphe $H(q)$ associé à une requête conjonctive q est défini par $H(q) =< N, E >$. N l'ensemble des nœuds est l'ensemble toutes les variables apparaissant dans le corps de la règle. E l'ensemble des hyper-arcs est l'ensemble $\{var(u_i) \mid u_i \in T(q)\}$.

Une requête q est cyclique (acyclique) si l'hypergraphe qui lui est associé est cyclique (acyclique). Nous utilisons la définition classique de cyclicité (aussi connue sous le nom de α-acyclicité [Fagin, 1983]) qui est utilisée en théorie des bases de données[Abiteboul et al., 1995, Maier, 1986, Ullman, 1989].

Etant donné un hypergraphe H la réduction GYO [Graham, 1979, Yu and Ozsoyoglu, 1979] de H est obtenue à partir de H en utilisant aussi longtemps que possible les deux règles suivantes :

1. Supprimer les hyper-arcs qui sont vides ou contenus dans un autre hyper-arc.
2. Supprimer les nœuds qui apparaissent dans au plus un hyper-arc.

Définition 2.10 *Un hypergraphe $H =< V, E >$ (V est un ensemble de nœuds ; E un ensemble d'hyper-arcs inclus dans 2^V) est acyclique si $GYO(H)=<\emptyset,\emptyset>$*

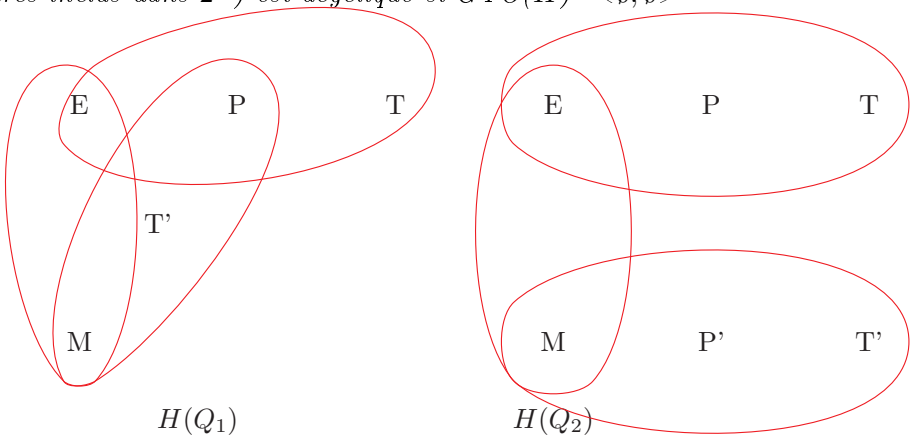


FIG. 2.8 – HyperGraphe

Exemple 2.3.4 : Considérons un ensemble de trois primitives :
- Travail(Emp,Proj,Tache) ; Un employé travaille sur une tâche dans un projet
- Manage(Emp,Proj,Tache) ; Un employé manage (est responsable) d'une tâche dans un projet.
- Chef(Emp1,Emp2) ; Emp1 est le chef de Emp2

Considérons les requêtes suivantes :
- Q_1 est une requête boolénne qui teste s'il existe des employés travaillant dans des projets managés par leur chef :

  ans ← Travail(E,P,T), Manage(M,P,T'), Chef(E,M).
- La requête booléenne Q_2 teste s'il existe un manager qui est le chef d'au moins un employé (travaillant sur un projet) :

  ans ← Travail(E,P,T), Manage(M ;P',T'), Chef(E,M).

Les hypergraphes de ces deux requêtes sont représentés dans la figure 2.8. On constate que Q_1 est cyclique alors que Q_2 ne l'est pas.

Remarque 2.3 *Si une requête q n'utilise que des prédicats unaires et binaires, la notion de cyclicité correspond à la notion de cyclicité dans les graphes non-orientés.*

L'un des intérêts majeurs des requêtes conjonctives acycliques concerne la complexité du problème de l'évaluation :

Théorème 2.4 ([Gottlob et al., 1998],[Gottlob et al., 2001]) *Etant donné une requête acyclique q, une base de donnée D et un tuple τ, décider si τ appartient à $D(q)$ est LOGCFL-complet.*

La classe de complexité LOGCFL contient tous les problèmes qui sont réductibles, avec un espace logarithmique, en un langage algébrique. La propriété qui nous intéressera par la suite est que LOGCFL est strictement inclus dans P.

Requêtes conjonctives sur les arbres XML

De nombreux travaux existent sur l'étude des requêtes conjonctives et sur l'étude de XPath. Dans [Gottlob et al., 2004], G. Gottlob, C. Koch et K. Schulz ont proposé une application des requêtes conjonctives à l'étude de XPath sur les documents arborescents.

L'ensemble de relations utilisé est la restriction de $P_{\Sigma,\mathcal{C}}$ au cas des documents à une couleur n'ayant pas de références (i.e. les axes *idref* et *reverse idref* ne sont pas pris en compte dans cette section). On le note P_Σ. L'interprétation est inchangée.

G. Gottlob, C. Koch et K. Schulz ont été les premiers à s'interroger sur les requêtes cycliques pour XML. Toutes les relations considérées étant binaires, on peut facilement représenter les requêtes conjonctives par des graphes. Par exemple, la figure 2.9 donne une représentation de la requête :

```
R(y,z)=A(x),descendent(x,y),ancestor(z,x),B(y),following(y,z)
```

Dans [Gottlob et al., 2004], les auteurs étudient la complexité de l'évaluation d'une requête conjonctive cyclique sur un arbre XML.

Proposition 2.2 ([Gottlob et al., 2004]) *Évaluer une requête conjonctive (utilisant l'ensemble de relations P_Σ) sur un arbre XML est NP-complet.*

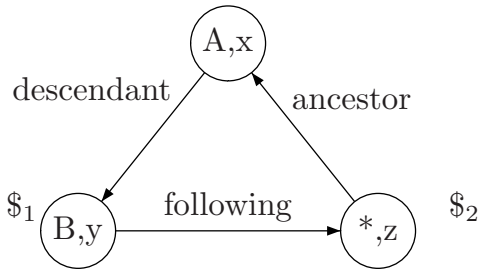


FIG. 2.9 – Requête conjonctive binaire

Une requête conjonctive (monadique) est équivalente à une requête du Core XPath. Par exemple, la requête

```
/descendent-or-self::A[child::B]/following::C
```

est équivalente à la requête conjonctive

```
Q(z)=
racine(w), descendent-or-self(w,x),A(x), child(x,y), B(y), following(x,z), C(z)
```

et à la figure 2.10.

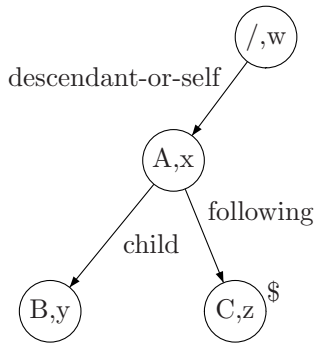


FIG. 2.10 – Requête conjonctive cyclique

L'équivalence entre les requête conjonctives monadiques et les requêtes du Core XPath n'est pas en contradiction avec les résultats montrés dans la section sur le Core XPath (en particulier le théorème 2.1). En effet, une requête conjonctive, dont l'évaluation est NP-dure, peut se représenter par une requête du Core XPath, dont l'évaluation est polynomiale. Mais il faut rappeler que, dans le pire des cas, il y a rapport exponentiel entre la taille des deux requêtes.

Cette section était consacrée aux requêtes conjonctives pour des documents à une seule couleur. Que peut-on dire des requêtes conjonctives colorées ? La réponse est donnée dans la section suivante.

2.4 Complexité et expressivité des requêtes graphes

Dans la section précédente, deux exemples de requêtes graphes ont été présentés. Ils donnent une idée sur l'expressivité et la complexité de ces requêtes. Le but de cette section est donc l'étude des requêtes graphes dans toute leur généralité.

2.4.1 Évaluation d'une requête graphe

La proposition 2.2 donne une borne inférieure pour le problème de l'évaluation d'une requête graphe : NP-dur. Nous prouvons dans cette section que le problème est NP. De plus, nous proposons une preuve de la dureté adaptée au cas des requêtes graphes. Elle est plus simple que celle de [Gottlob et al., 2004].

Théorème 2.5 *Le problème de l'évaluation d'une requête graphe utilisant les primitives de $P_{\Sigma,\mathcal{C}}$ sur un document XML coloré est NP-complet.*

Montrons que l'évaluation d'une requête graphe est NP-dure dans la taille de la requête. Pour cela on code le problème de 3-coloration d'un graphe.

Définition 2.11 *Le problème du 3-coloriage est défini par :*
Entrée : un graphe orienté G
Sortie : vrai si et seulement si on peut colorier les nœuds de G avec 3 couleurs tel que deux nœuds voisins n'aient pas la même couleur.

L'idée est de colorier un graphe G quelconque en le transformant en requête graphe q_G et en interrogeant le document $D_{couleur}$ réduit à l'arbre XML fixe suivant :

```
<doc>
  <ROUGE/>
  <BLEU/>
  <VERT/>
</doc>
```

Colorions G en construisant un plongement de ce graphe dans le document XML ci-dessus. Pour cela, il faut définir une interprétation des arcs de G qui garantisse qu'un plongement β ne colorie pas deux voisins de G de la même couleur. Cette négation peut se traduire avec la notion de voisin du document XML : deux nœuds n_1 et n_2 de G n'ont pas la même couleur si et seulement si $\beta(n_1)$ est un voisin (à droite ou à gauche) de $\beta(n_2)$. Dans le cas très particulier du document à trois couleurs, on interprète donc les arcs de G (vu comme une requête) par l'ensemble des couples (n, n') tels que n est un *preceding-sibling* de n' ou n est un *following-sibling* de n'.

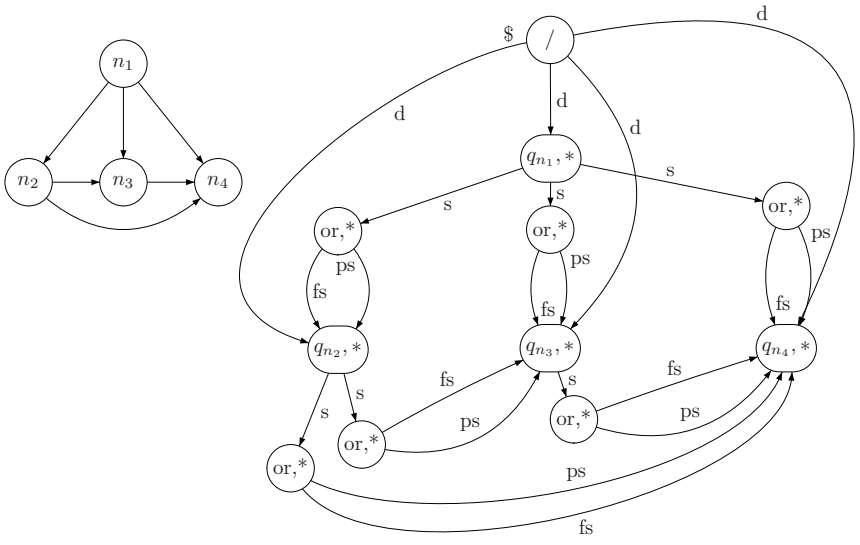


FIG. 2.11 – 3-coloration d'un graphe avec une requête graphe

function 3-coloriable(G) :requête
% voir la figure 2.12
% Entrée : un graphe orienté G
% Sortie : une requête graphe q_G telle que $/_{couleur}$ appartienne à $D_{couleur}(q_G)$
si et seulement si G est 3-coloriable
construire un nœud $n_/ = (/, and)$ dans la requête q_G
pour tous les nœuds n de G faire
construire un nœud $q_n = (*, and)$ dans la requête q_G
ajouter un arc *descendent-or-self* de $n_/$ vers q_n dans q_G
fin pour
pour tous les arcs $a = (n, n')$ de G faire
construire un nœud $q_a = (*, or)$ dans la requête q_G
ajouter un arc *self* de q_n vers q_a dans q_G
ajouter un arc *preceding-sibling* de q_a vers $q_{n'}$ dans q_G
ajouter un arc *following-sibling* de q_a vers $q_{n'}$ dans q_G
fin pour
le sectionneur de q_G est le nœud $n_/$
retourner q_G
fin

TAB. 2.1 – 3-coloration d'un graphe.

Malheureusement aucune relation de $P_{\Sigma,\mathcal{C}}$ n'a cette interprétation. L'algorithme suivant transforme donc un graphe G en une requête graphe q_G en combinant la relation *preceding-sibling*, la relation *following-sibling* et un nœud *or*. q_G sélectionne la racine de $D_{couleur}$ si et seulement si G est 3-coloriable. Par exemple, la figure 2.11 montre un exemple de graphe G qui n'est pas 3-coloriable et la requête q_G associée.

La requête q_G vérifie une propriété particulière : si un plongement β satisfait $\beta(n_/) = /$ (i.e. si le résultat de q_G est non vide) alors quelque soit le nœud n de q_G, $\beta(n)$ est défini. Cette caractéristique découle des arcs *descendent* qui lient le nœud $n_/$ et tous les nœuds de la forme q_n. Cette propriété étant réutilisée dans la suite dans cette thèse, nous allons la nommer :

Définition 2.12 *Une requête graphe q est dite totale si tout plongement β sélectionnant un n-uplet de nœuds est un plongement total (i.e. pour tout nœud v de la requête $\beta(v)$ est défini).*

Lemme 2.2 *La fonction* **3-coloriage** *décrite ci-dessus (voir aussi la figure 2.12) permet de réduire le problème du 3-coloriage au problème d'évaluation d'une requête graphe.*

Preuve : Si la racine du document $D_{couleur}$ est sélectionnée par la requête q_G, il existe alors un plongement de la requête dans le document. Comme la requête q_G est totale, chaque nœud q_n de la requête est associé à un nœud du document XML (i.e. il est colorié). Or, d'après notre discussion précédente, si n et n' sont deux nœuds voisins de G alors les nœuds q_n et $q_{n'}$ ne peuvent pas avoir la même couleur i.e. on peut 3-colorier le graphe : tout nœud n a la couleur de q_n.

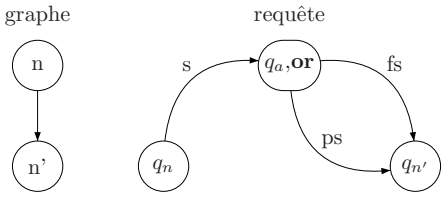


FIG. 2.12 – Algorithme de 3-coloriage

Inversement, si le graphe est 3-coloriable associons à chaque nœud n de G une couleur $col(n)$ (choisie parmi bleu, rouge ou vert). La fonction β qui associe au nœud q_n le nœud de $D_{couleur}$ correspondant à $col(n)$, au nœud q_a la couleur du nœud origine de l'arc a et au nœud $n_/$ la racine de $D_{couleur}$ est un plongement de la requête dans le document. On a donc prouvé que la racine est sélectionnée par la requête.

Il est donc possible de résoudre le problème du 3-coloriage avec les requêtes graphes utilisant les primitives de $P_{\Sigma,\mathcal{C}}$ ce qui prouve que l'évaluation de ces requêtes est un problème NP-dur. ◀

Corollaire 2.1 (de la réduction) *Evaluer une requête graphe totale utilisant les primitives de $P_{\Sigma,\mathcal{C}}$ est un problème NP-dur.*

On peut faire le point sur l'origine de la dureté du problème. Pour cela, utilisons les arguments de la preuve de [Gottlob et al., 2004] et les arguments donnés dans cette preuve :

1. Le problème est NP-dur même si on considère des requêtes graphes conjonctives [Gottlob et al., 2004].
2. La dureté du problème ne vient pas de la forme du document ; le document considéré est un arbre XML à une couleur.
3. La dureté du problème peut se montrer en utilisant les axes *preceding-sibling* et *following-sibling*, *child* et *following* ou avec les axes *child* et *descendent*. L'ordre entre les fils d'un nœud n'est pas la source de la complexité du problème.
4. Le problème est NP-dur même si on ne considère que des requêtes graphes sans sélectionneur [Gottlob et al., 2004].

Montrons maintenant que le problème est NP et généralisons les résultats de [Gottlob et al., 2004]. Montrons un résultat de complexité générale sur l'évaluation d'une requête graphe sur un document XML coloré.

Proposition 2.3 *Soit $E = (P, R)$ un ensemble de primitives. Si pour tout document XML coloré $D =< N, \mathcal{C}, \{D_c\}_{c\in\mathcal{C}} >$, pour tous nœuds n et n' de N les deux conditions suivantes sont satisfaites :*

1. *pour tout prédicat p de P, on peut décider si $p(n)$ est vrai en temps polynomial (dans la taille de D)*
2. *pour toute relation r de R, on peut décider si (n, n') appartient à r en temps polynomial (dans la taille de D)*

alors le problème de l'évaluation d'une requête graphe (utilisant des primitives de E) sur un document graphe appartient à la classe NP.

Preuve :

La méthode proposée est celle qui consiste à trouver, par un algorithme non déterministe, un plongement de la requête dans le document qui sélectionne le n-uplet τ candidat. On propose un algorithme qui étant donnès une requête q, un document D et un certificat β teste si β est un plongement de q dans D. Par défault, si $\beta(v)$ n'est pas défini alors $\beta(v) = \perp$ avec $\perp$ un nouveau nœud. Pour tous les prédicats p, $p(\perp)$ est faux. De même, pour toute relation r, $\perp$ n'est en relation avec aucun autre nœud de D.

est-un-plongement(β,q,D)
% *Entrée : une fonction β qui, à un nœud v d'une requête graphe q associe, soit*
% *un nœud n_v d'un document D soit le symbole $\perp$ (i.e. $\beta(v)$ est non défini)*
% *Sortie : vrai si et seulement si β est un plongement de q dans D*
plongement := *vrai*
pour tous les nœuds v de la requête faire
  si $pred_v = /_c \wedge \beta(v) = \perp$ *alors plongement* := *faux fin si*
  si $\neg[\beta(v) \neq \perp \Rightarrow pred_v(\beta(v)) \wedge (\underset{a=(v,v')\in arcs(q)}{bool_v} (\beta(v), \beta(v')) \in axe_a)]$
    alors plongement := *faux*
  fin si
fin pour
pour tous les sélectionneurs $\$_i$ *faire*
  si $\beta(\$_i) = \perp$ *alors plongement* := *faux fin si*
fin pour
retourner plongement
fin

La taille du certificat est la somme du nombre de nœuds de D, de la taille de la requête et de la taille du document D. Évaluons la complexité de la procédure **est-un-plongement** *: tous les $pred_v$ sont évalués exactement une fois et tous les $axe_{(v,v')}$ sont évalués exactement une fois. On déduit facilement des hypothèses que* **est-un-plongement** *est polynomial et donc que le problème est NP.* ◄

Il est connu de G. Gottlob et al. [Gottlob et al., 2002] que pour tout document XML arborescent D, tous les axes du Core XPath peuvent être évalués en $O(|D|)$. P. Ramanan [Ramanan, 2003] étend ce résultat au cas des axes *idref* et *rir*. On en déduit que tous les prédicats et toutes les relations présents dans l'ensemble $P_{\Sigma,\mathcal{C}}$ vérifient les hypothèses de la proposition 2.3 et donc l'évaluation d'une requête graphe utilisant ces pritimitives est NP.

Résumons les principaux résultats de complexité concernant les requêtes graphes :

Proposition 2.4 *1. Evaluer une requête graphe sur un document XML coloré est un problème NP-complet.*

2. *Si le document est fixe, le problème est NP-dur dans la taille de la requête.*
3. *Si la requête est fixe, le problème est polynomial dans la taille du document : $O(|D|^{|q|})$.*

Trois corollaires découlent de la preuve que nous venons de faire :

Corollaire 2.2 *Evaluer une requête conjonctive colorée est un problème NP-complet.*

L'appartenance du problème à la classe NP a été montrée dans cette section et la dureté a été montrée dans la section précédente.

Corollaire 2.3 *Décider si le résultat d'une requête graphe sur un document XML coloré est vide est un problème NP-complet.*

Pour prouver le théorème 2.5, on devine par un algorithme non déterministe un plongement qui sélectionne un n-uplet de nœuds du document. Si on modifie la preuve pour deviner le n-uplet à sélectionner, on montre le corolaire 2.3.

Nous avons donc mis en évidence un saut de complexité entre l'évaluation d'une requête arbre (du core XPath) et l'évaluation d'une requête graphe. Afin d'avoir une hiérarchie complète du problème de l'évaluation en fonction de la forme de la requête, étudions deux autres familles de requêtes.

Corollaire 2.4 *Evaluer une requête DAG (direct acyclique graph) est un problème NP-complet.*

En effet, on peut modifier la fonction **3-coloriable** (en choisissant astucieusement l'ordre des nœuds du graphe à partir d'un arbre couvrant du graphe) pour construire uniquement des requêtes graphes orientées acycliques.

Pour terminer notre étude de complexité, il faut dire qu'évaluer une requête conjonctive acyclique (au sens de la définition 2.10 ; page 31) sur un document XML coloré est polynomial. C'est un corollaire du théorème 2.4 (page 32). On peut en effet construire en temps polynomial la base de données qui contient tous les axes utilisés dans une requête (en fait, on pré-calcule les relations du core XPath).

Un enseignement important de cette section est que la complexité du problème de l'évaluation dépend de la forme de la requête mais pas de la forme du document. Rappelons en effet que les résultats de complexité énoncés dans cette section :

Forme de la requête	Forme du document	Complexité de l'evaluation
Arbre (core XPath)	XML coloré	$O(\|D\|.\|q\|)$ (pour le calcul de $D(q)$)
Acyclique (conjonctive)	XML coloré	polynomial
DAG	Arbre XML	NP-complet (document fixé)
Graph	Arbre XML	NP-complet (document fixé)

2.4.2 Expressivité des requêtes graphes

Dans les sections précédentes on a établi le lien entre le Core Xpath et les requêtes arbres. Les requêtes graphes sont donc plus expressives que le Core Xpath coloré (sans négation). L'opérateur booléen *or* ne pouvant pas s'exprimer avec l'opérateur booléen *and*, les requêtes

C$_=$Xquery	$\rightarrow$	for_clause where_clause return_clause
for_clause	$\rightarrow$	for_clause for_clause_e | for_clause_e
for_clause_e	$\rightarrow$	**for** *var* **in** /{c}descendent-or-self::tn
where_clause	$\rightarrow$	**where** condition | ε
condition	$\rightarrow$	condition **and** condition_e | condition_e
condition_e	$\rightarrow$	(égalité$_{or}$) | (égalité$_{and}$) | var == /$_c$
égalité$_{or}$	$\rightarrow$	égalité$_{or}$ **or** égalité | égalité
égalité$_{and}$	$\rightarrow$	égalité$_{and}$ **and** égalité |égalité
égalité	$\rightarrow$	var/{c}axe::* == var
return_clause	$\rightarrow$	$(var, \ldots, var)$

TAB. 2.2 – Description du Core Xquery coloré avec opérateurs d'égalité

graphes sont plus expressives que les requêtes conjonctives. On souhaite donc trouver un fragment de Xquery qui ait exactement la même expressivité que nos requêtes.

Pour construire des requêtes graphes, il faut écrire des requêtes contenant des égalités entre les nœuds d'un document. Pour cela, on utilise le mécanisme de variables utilisé dans les expressions *flwor* de Xquery. Le but de cette section est donc de présenter la syntaxe de Core Xquery coloré avec égalité (voir la grammaire algébrique de la table 2.2), la sémantique associée et le lien avec les requêtes graphes.

Quel est la différence entre ces deux langages ? Les requêtes graphes peuvent-être définies graphiquement et, en ce sens, elles sont simples d'utilisation (user friendly). Les requêtes du Core Xquery avec égalités conviennent mieux pour un utilisateur habitué à Xquery ou même à tout langage de programmation.

Remarque 2.4 *C. Koch a très récemment défini le Core Xquery dans [Koch, 2005a]. Ses requêtes sont plus expressives que les nôtres dans le sens où elles permettent la composition de requêtes. De plus, son langage est un langage de transformation d'arbre alors que C$_=$Xquery coloré est un langage de requête n-aire. Néanmoins, sa définition du Core Xquery sans composition donnée dans [Koch, 2005b] ressemble à la notre car elle intègre la notion d'égalité. Nous avons toujours la spécificité de travailler sur des documents colorés.*

Commentons la syntaxe d'une requête C$_=$Xquery coloré (cf. table 2.2) :

- *axe* fait référence à l'un des treize axes de navigation de Xpath[7]. *tn* est un test sur les nœuds (i.e. soit le symbole $*$, soit un le nom d'un label). Enfin $\{c\}$ fait référence à un ensemble de couleurs.
- Une requête contient une suite de *for*. Chaque *for* définit une nouvelle variable *var*. Ensuite la requête contient éventuellement une condition dans une clause *where*. Elle s'exprime comme une condition sur des *égalité*. La requête se termine par une clause *return* qui est une suite ordonnée de variables.

Deux variables apparaissant dans deux clauses *for* différentes doivent avoir un nom différent. On prend donc la notation classique, qui consiste à indicer une variable par la profondeur du *for* auquel elle se rapporte. Le n-uplet de variables de la clause *return* se note

[7] Les treize axes sont : self, child, descendant, descendant-or-self, parent, ancestor, ancestor-or-self, preceding, preceding-sibling, following, following-sibling, idref et ridref ; cf. définition 2.2

$(var^1, \ldots, var^n)$. De plus chaque variable utilisée dans la clause *where* ou dans la clause *return* doit être définie dans la clause *for*. Une dernière contrainte concerne les racines. Une variable x doit apparaître au plus une fois dans une clause var $== /_c$.

La sémantique de C$_=$XQuery est la suivante : un n-uplet $\tau = (\tau_1 \ldots \tau_n)$ de nœuds d'un document D est sélectionné par une requête q s'il existe une valuation σ entre les variables de la requête et les nœuds du document vérifiant :

1. Pour toute clause *for* var_i *in* $/\{c\}$*descendent_or_self*::*tn*, le nœud $\sigma(var_i)$ appartient à un arbre dont la couleur appartient à c. De plus, $\sigma(var_i)$ satisfait tn. Rappelons que le test $*$ est toujours vrai et que le test σ sélectionne les nœuds dont le label est σ.
2. La condition de la clause *where* est satisfaite.
   Une égalité $var_i/\{c\}axe::* == var_j$ est satisfaite si le couple de nœuds $(\sigma(var_i), \sigma(var_j))$ appartient à la relation {c}axe. De même $var_i == /_c$ est vraie si $\sigma(var_i)$ satisfait le prédicat $/_c$ (page 19).
   Une condition égalité$_{or}$ est vraie si toutes les variables gauches utilisées sont les mêmes et si l'une des égalités est vraie. Une condition égalité$_{and}$ est vraie si toutes les variables gauches utilisées sont les mêmes et si toutes les égalités sont vraies.
   Une condition est vraie si toutes les conditions égalité$_{or}$ et égalité$_{and}$ sont vraies.
3. Le n-uplet $(\tau_1 \ldots \tau_n)$ est égal à $(\sigma(var^1), \ldots, \sigma(var^n))$

graphe2C$_=$Xquery(q)
% Entrée : une requête graphe totale q
% Sortie : une requête C$_=$Xquery coloré équivalente à q
% Locale :
pour tous les nœuds v de q faire
  si $pred_v$ n'est pas le symbole $/_c$ alors
    ajouter une clause for : *for* var_v *in* $/\{\mathcal{C}\}$*descendant-or-self*::$pred_v$
  sinon
    ajouter une clause for : *for* var_v *in* $/\{\mathcal{C}\}$*descendant-or-self*::*
    ajouter une égalité : $var_v == /_c$
  fin si
  ajouter une clause égalité$_{bool_v}$: $\underset{a=(v,v')\in arcs(q)}{bool_v}$ $\text{var}_v/\text{axe}_a::*==\text{var}_{v'}$
  si v est un nœud $\$_i$ alors
    var_v est la i-ème composante de la clause *return*
  fin si fin

TAB. 2.3 – D'une requête graphe vers une requête C$_=$Xquery coloré

Remarque 2.5 *Dans la sémantique de C$_=$Xquery avec des égalités une seule égalité entre les nœuds est possible : le == qui désigne l'égalité d'identifiant entre deux nœuds (opérateur*

is *de Xpath). La syntaxe et la sémantique de notre langage pourraient être étendues (sans complication) en intégrant d'autres opérateurs d'égalités. En Xquery on peut aussi exprimer l'égalité de valeur, l'égalité de profondeur... entre deux nœuds.*

Pour terminer cette section, faisons le lien entre les requêtes graphes et le Core Xquery colorés avec égalité. Le passage d'un modèle vers l'autre est dirigé par l'idée suivante : une clause *for* d'une requête C$_=$Xquery coloré correspond à un nœud d'un graphe et une égalité var_i/axe : :* == var_j correspond à arc étiqueté par "axe" entre les nœuds var_i et var_j. Enfin l'opérateur $bool_{var_i}$ est défini par la condition égalité$_{or}$ (ou égalité$_{and}$) dont var_i est le membre gauche. Il ne faut pas oublier que toutes les variables d'une requête C$_=$Xquery doivent être affectées à un nœud du document. On retrouve donc la notion de requête graphe totale. En fait, le théorème suivant énonce l'équivalence entre l'expressivité de ces deux langages de requêtes.

Théorème 2.6 *Les requêtes graphes totales définies avec les primitives de l'ensemble $P_{\Sigma,\mathcal{C}}$ ont la même expressivité que les requêtes du Core Xquery coloré avec égalité.*

Caractérisons les requêtes graphes totales. Notons racines(q) l'ensemble des nœuds d'une requête graphe q ayant une étiquette de la forme $/_c$ où c est un ensemble de couleurs.

Proposition 2.5 *Toute requête graphe totale est équivalente à une requête graphe q satisfaisant la propriété suivante : pour tout nœud v de q il existe un chemin uniquement étiqueté par l'opérateur and entre un nœud de racines(q) et v.*

Si q satisfait la condition décrite ci-dessus, alors q est une requête totale. Réciproquement, soit $q =< N, A, \$ >$ est une requête totale. Construisons la requête $\bar{q} =< N \cup \bar{N}, A \cup \{(n, \bar{n}) \mid n \in N\}, \$ >$ où $\bar{N}$ est une copie de N. Les nœuds de $\bar{N}$ sont étiqueté par le couple $(/_{\mathcal{C}}, and)$ où $\mathcal{C}$ est l'ensemble de toutes les couleurs. Les arcs $(\bar{n}, n)$ sont étiqueté par $\{\mathcal{C}\}$descendant-or-self. On exprime ainsi que tout nœud est le descendant de l'une des racines. La requête $\bar{q}$ est équivalente à q et satisfait la condition syntaxique donnée dans la proposition ci-dessus. On remarquera que toutes les requêtes totales utilisées dans cette thèse vérifient cette condition.

Pour montrer le théorème 2.6, utilisons les algorithmes de la table 2.4 et de la table 2.3 : **graphe2C$_=$Xquery** transforme une requête graphe totale en requête C$_=$Xquery et **C$_=$Xquery2graphe(q)** fait le travail inverse.

Les deux exemples suivants ont pour but d'illustrer l'équivalence entre les deux types de requêtes. Pour cela, on montre le fonctionnement des algorithmes donnés dans la table 2.4.

Exemple 2.4.1 :

La figure 2.13 montre la requête graphe associée à la requête C$_=$Xquery suivante :

for x_1 in /{c_1}descendent-or-self: :a
  for x_2 in /{c_1}descendent-or-self: :*
    for x_3 in /{c_2}descendent-or-self: :*
    where (x_1/{c_1}*ancestor*: :* == x_2 or x_1/{c_2}*ancestor*: :* == x_3) and
      (x_2/{c_1}*child*: :* == x_3)
return x_1

C$_=$Xquery2graphe(q)
% Entrée : une requête C$_=$Xquery coloré q
% Sortie : une requête graphe (totale) équivalent à q
% Locale :
pour toutes les clauses *for* var_i *in* /{c}*descendent-or-self*::tn faire
  créer un nœud $n_{/_c} = (/_c, and)$
  créer un nœud $\text{var}_i = (tn, and)$
  ajouter un arc {c}*descendent-or-self* de $n_{/_c}$ vers var_i
fin pour
pour toutes les conditions égalité$_{op}$ de variables gauche var_i faire
  créer un nœud $e = (*, op)$
  ajouter un arc *self* entre var_i et e
  pour toutes les égalité var_i/{c}axe::* == var_j faire
    ajouter un arc {c}axe entre e et le nœud var_j
  fin pour
fin pour
pour toutes les égalités $var_i == /_c$ faire
  $/_c$ est le booléen associé au nœud var_i
fin pour
pour tous les nœuds var^k de la clause *return* faire
  le nœud var_k est le sélectionneur $\$_k$
fin pour
fin

TAB. 2.4 – D'une requête C$_=$Xquery coloré vers une requête graphe

Exemple 2.4.2 :

La procédure **graphe2C$_=$Xquery** donnée dans la table 2.3 transforme la requête de la figure 2.14 en la requête C$_=$Xquery suivante :

for x_1 in /{$\mathcal{C}$}descendent-or-self::*
  for x_2 in /{$\mathcal{C}$}descendent-or-self::*
    for x_3 in /{$\mathcal{C}$}descendent-or-self::a
      where $x_1 == /_{c_1}$ and (x_1/{c_1}descendent-or-self::* == x_2) and
      (x_2/{c_1}child::* == x_3 or x_2/{c_2}idref::* == x_3)
return (x_2, x_3)

Pour étudier la complexité de l'évaluation d'une requête C$_=$Xquery coloré avec égalité, utilisons le corollaire 2.1 et le théorème 2.6 pour montrer le théorème suivant :

Théorème 2.7 *Évaluer une requête C$_=$Xquery coloré avec égalité sur un document XML coloré est un problème NP-complet.*

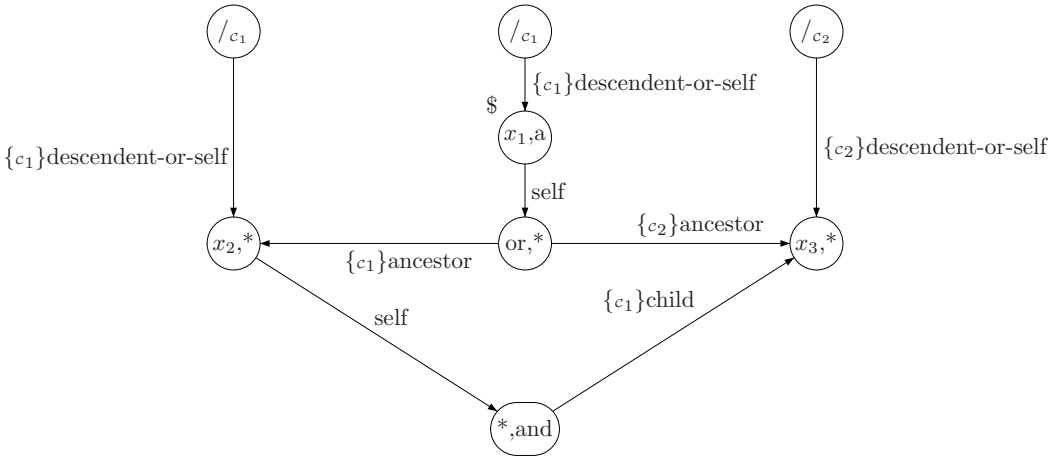


FIG. 2.13 – Requête graphe construite à partir d'une requête C$_=$Xquery coloré

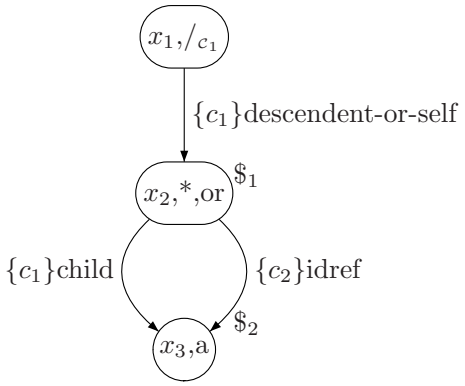


FIG. 2.14 – Requête graphe totale

Preuve : Le point clé est que les algorithmes **graphe2C$_=$Xquery** *et* **C$_=$Xquery2graphe** *sont PTIME.*

On peut donc résoudre le problème de l'évaluation d'une requête graphe totale en évaluant une requête C$_=$Xquery coloré. Ce premier problème étant NP-dur (corollaire 2.1), l'évaluation de C$_=$XQuery l'est aussi.

Comme il est possible d'évaluer une requête C$_=$Xquery coloré avec une requête graphe, le problème est NP (théorème 2.5).

◀

Remarque 2.6 *Il est possible d'étendre la syntaxe de C$_=$Xquery coloré pour ajouter un opérateur de négation. On ajoute la règle suivante à la grammaire de la table 2.2 :*

condition → *not(C$_=$Xquery)*

Si q est une requête, le prédicat not(q) est vrai si le résultat de q est vide.

Du point de vue de la complexité, l'évaluation devient PSPACE-complet. La preuve n'est pas présentée car elle n'utilise pas les requêtes graphes. Elle est de plus très semblable à celle présentée dans [Koch, 2005b].

2.5 Conclusion

Le but de ce chapitre était d'étudier les requêtes graphes pour les documents XML coloré. Un document XML coloré est un ensemble d'arbres XML qui se partagent des nœuds. Nos requêtes sont définies à partir des axes de navigations de XPath et XQuery. On utilise en plus deux axes qui permettent de naviguer dans un document en utilisant ses références.

Deux types de requêtes graphes existent déjà : les requêtes du Core XPath et les requêtes conjonctives. On a étendu leur expressivité des documents XML vers le XML coloré. Cette transformation s'est faite sans modification de la complexité du problème de l'évaluation.

Nous avons enfin montré qu'évaluer une requête graphe est un problème NP-complet. Pour étudier l'expressivité de notre langage, nous avons construit un fragment de XQuery, $C_{=}$XQuery coloré, équivalent aux requêtes graphes. Ce fragment permet de définir l'égalité entre les nœuds.

Travaux en cours

Nous nous intéressons aux points suivants :

- Dans ce chapitre, les primitives de $P_{\Sigma,\mathcal{C}}$ ont été restreintes aux axes du Core XPath et aux tests sur les balises. La question à se poser est donc : quel est le plus grand fragment de XPath (XQuery) qui puisse être exprimé avec des requêtes graphes ? Comment garantir que l'évaluation reste un problème NP ? Pour l'instant, nous travaillons sur la fonction *count*.
- La seconde piste, peut être à plus long terme, est de proposer des algorithmes pour optimiser les requête graphes. L'idée qui semble la plus prometteuse est celle qui s'inspire de l'algorithme *chase* pour les requêtes conjonctives (voir [Amer-Yahia et al., 2001] pour une application aux requêtes twig). Le principe est de réécrire la requête en une requête équivalente : pour cela, on utilise des redondances qui proviennent soit de la sémantique des axes (les requêtes /descendant-or-self : :* et /descendant-or-self : :*/ancestor-or-self : :* sont équivalentes) soit de contraintes vérifiées par le document (si tous les nœuds *auteur* ont un fils *nom* alors le prédicat [auteur/child : :nom] peut se simplifier en [auteur]). Sur le même principe, mais beaucoup plus ambitieux, on peut s'inspirer des travaux de N. Bruno, N. Koudas et D. Srivastava [Bruno et al., 2002] qui améliorent la complexité en espace (en diminuant le nombre de jointures) des algorithmes de [Al-Khalifa et al., 2002] pour l'évaluation des requêtes twig.

Chapitre 3

Documents représentés par un graphe : contraintes d'inclusions

3.1 Introduction

L'objectif de cette thèse est d'étudier les propriétés des données semi-structurées modélisées par un graphe. Les chemins étant une caractéristique principale des graphes (ils permettent en effet de naviguer dans une donnée et d'en visiter les noeuds) la notion de requêtes régulières permet d'interroger une donnée à partir de ses chemins. Le but de ce chapitre est donc de présenter ces requêtes et d'étudier des contraintes sur les chemins d'une donnée.

Dans ce chapitre, le cadre des données semi-structurées n'est pas limité aux cas des documents XML colorés. Nous utilisons en effet le modèle d'échange de données (OEM- Object Exchange Model) proposé dans [Abiteboul et al., 2000]. Si A désigne un alphabet fini de labels, une donnée semi-structurée est un triplet $< N, Racines, T >$ où N est un ensemble de noeuds, *Racines* est un sous-ensemble de N et désigne les racines de la donnée. Enfin T inclus dans $N \times A \times N$ désigne l'ensemble des arcs de la donnée. On considère donc les données semi-structurées comme des graphes orientés étiquetés à plusieurs racines. Par exemple, des pages HTML (une page est un noeud, un lien est un arc, la page d'accueil est une racine) se modélisent par des graphes. Enfin, les données XML (Extensible Markup Language [Bray et al., 1998]) peuvent être vues comme des graphes. Cette vision ne permet plus d'utiliser les axes du Core Xpath, tel que *preceding*, qui utilisent l'ordre des frères.

La donnée représentée dans la figure 3.1 décrit un ensemble de compagnies d'assurance et d'assurés. Chaque assuré peut être client de plusieurs compagnies : il n'est pas obligé d'assurer sa voiture et son habitation au même endroit. Enfin, un assuré peut avoir des ayants-droits qui peuvent, eux aussi, être assurés.

Requêtes :

Pour naviguer dans une donnée, on utilise des expressions régulières qui permettent de former des chemins dans le graphe qui représente la donnée. On retrouve de telles requêtes dans UnQL [Buneman et al., 1996] ainsi que dans Lorel [Abiteboul et al., 1997] langage de requêtes défini dans le cadre du projet Lore. Enfin, diverses propositions faites pour interroger un document XML utilisent fortement la notion de chemins. On peut par

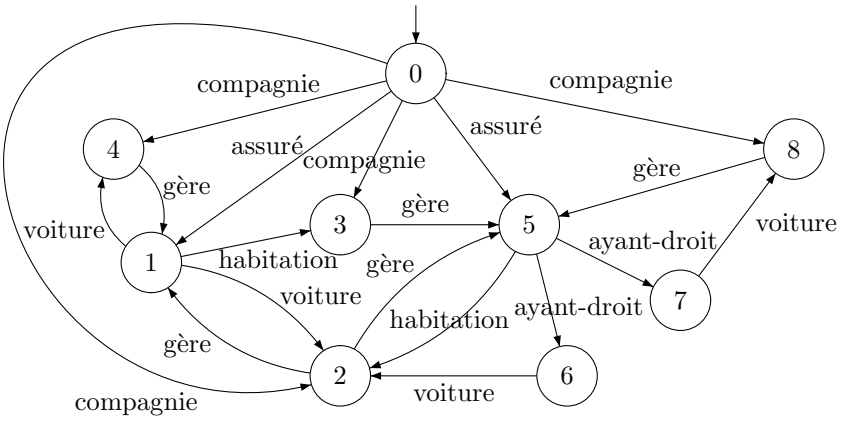


FIG. 3.1 – Une représentation de donnée semi-structurée.

exemple citer XML-QL [Deutsch et al., 1999], XQL [Rogie et al., 1998], Quilt ou plus récemment XQuery [Berglund et al., 2002b].

Le résultat de la requête régulière q sur une donnée D est l'ensemble $D(q)$ de noeuds accessibles depuis l'une des racines par un chemin étiqueté par un mot de q. Un nœud n d'une donnée D appartient à $D(q)$ s'il existe un mot u du langage décrit par q et un chemin étiqueté par u joignant l'une des racines de D et le noeud n. Dans la suite de ce chapitre, on ne fait plus la différence entre le mot u d'une requête régulière et les chemins étiquetés par u dans une donnée. Par exemple, `compagnie`, `compagnie.gère`, `assuré.ayant-droit.voiture` sont des chemins de la donnée décrit dans la figure 3.1. L'expression régulière `compagnie.(gère.voiture)`* est une requête régulière dont le résultat est $\{2, 3, 4, 8\}$.

Contraintes de chemins :

Il est utile d'avoir des informations sur les chemins d'une donnée, sur les noeuds qu'ils atteignent et sur la structure sous-jacente de ces chemins (pour optimiser une requête régulière, pour décrire la donnée...). Les contraintes de chemins sont une partie de cette information. Trois grandes classes de contraintes existent [Alechina et al., 2003] : les contraintes d'inclusions, les contraintes inverses et les contraintes *lollipop* (figure 3.2).

Une contrainte d'inclusion s'écrit $p \preceq q$ où p et q sont des requêtes régulières. Une donnée D satisfait (est un modèle d') une telle contrainte si tout noeud résultat de p est aussi résultat de q. Une contrainte inverse s'écrit $p \preceq_i q$ où p et q sont des requêtes régulières. Une donnée D satisfait une telle contrainte si de tout noeud accessible par p à partir d'une racine r on peut rejoindre le nœud r avec un mot de q. Enfin une contrainte *lollipop* est une contrainte d'inclusion qui s'exprime dans un contexte c : la contrainte $c \rightsquigarrow p \preceq q$ (c, p et q sont des requêtes régulières) est satisfaite par une donnée D si la contrainte $p \preceq q$ est satisfaite à partir de tout noeud appartenant à $D(c)$.

Nous illustrons maintenant comment les contraintes de chemins permettent d'optimiser des requêtes. Cette optimisation se base sur la recherche de requêtes équivalentes : deux requêtes p et q sont équivalentes par rapport à un ensemble de contraintes $\mathcal{C}$ si quelque soit la donnée D modèle de $\mathcal{C}$ les ensembles $D(p)$ et $D(q)$ sont égaux. Par exemple, toute donnée satisfaisant la contrainte `compagnie.gère.voiture` $\preceq$ `compagnie` satisfait aussi la contrainte

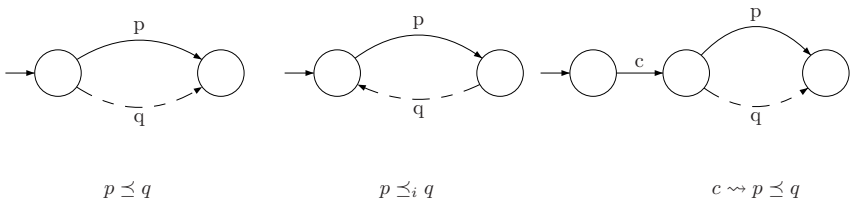


FIG. 3.2 – Différents types de contraintes de chemins.

`compagnie.(gère.voiture)`$^* \equiv$ `compagnie`. Dans ce cas, on peut réécrire une requête décrivant un langage infini (`compagnie.gère.voiture`*) en une requête équivalente qui décrit un langage fini.

Afin de réaliser ces optimisations, plusieurs problèmes sont à résoudre : l'un de ces problèmes consiste à manipuler les contraintes impliquées par un ensemble de contraintes. C'est le problème de l'*implication* . Un autre problème est de trouver, à partir d'un ensemble de contraintes $\mathcal{C}$ et d'une requête q, une requête q_f équivalente à q, décrivant un langage fini. C'est le problème de *l'équivalent fini* par rapport à un ensemble de contraintes $\mathcal{C}$.

Le problème de l'implication :

Un ensemble de contraintes de chemins $\mathcal{C}$ implique une contrainte de chemin c si tout modèle de $\mathcal{C}$ est aussi un modèle de c. On note cette implication par $\mathcal{C} \models c$.

Un moyen très classique d'étudier les contraintes d'inclusions est d'utiliser la logique. Le principe de base est d'écrire une formule logique qui décrit la contrainte. Dans ce cadre, une contrainte est vérifiée par une donnée si cette donnée est un modèle de la formule. De même le problème de l'implication se traduit facilement en logique. En effet si $\mathcal{C} = \{c_i, 1 \leq i \leq n\}$ est un ensemble de contraintes et c une contrainte alors $\mathcal{C} \models c$ si et seulement si la formule $c_1 \wedge c_2 \wedge \ldots \wedge c_n \Rightarrow c$ est valide (i.e. vraie pour toutes les données).

Quelles sont les logiques qui permettent de capturer les contraintes d'inclusions ? Il est bien connu que la sémantique de la famille des logiques modales est définie sur les graphes grâce aux modèles de *Kripke* [Kripke, 1971]. Ces logiques permettent d'exprimer des propriétés simples sur les graphes. Le problème des logiques modales est que l'on ne peut pas nommer un noeud (pris individuellement) ce qui nous empêche d'exprimer les contraintes d'inclusions.

Deux solutions sont proposées pour contourner ce problème : l'utilisation de PDL [Alechina et al., 2003] ou l'utilisation d'une logique hybride [Bidoit et al., 2004].

N. Alechina, S. Demri et M. de Rijke définissent dans [Alechina et al., 2003] la logique PDL^{path} pour exprimer les contraintes d'inclusions. PDL^{path} est un fragment de CPDL qui contient des nominaux [De Giacomo, 1995]. La spécificité de cette logique réside dans l'utilisation d'un opérateur inverse qui permet, entre autre, de remonter un chemin (i.e. des noeuds accessibles vers une racine). Une contrainte d'inclusion $p \preceq q$ peut alors s'exprimer par une formule dont l'interprétation est : il existe à partir de tout noeud accessible par p un chemin *inverse* de q qui atteint une racine.

N. Bidoit, S. Cerrito et V. Thion proposent dans [Bidoit et al., 2004] une logique hybride multimodale qui permet d'exprimer les contraintes d'inclusions. Cette logique introduit un opérateur qui permet de nommer un noeud avec une variable. Par exemple, la contrainte

$p \preceq q$ s'exprime avec une formule signifiant que tout noeud x accessible depuis la racine par p doit aussi être atteint, depuis la racine, par q. Cette logique hybride permet aussi de définir une extension de la notion de DTD dans laquelle on peut explicitement associer un type à une référence.

La limite de ces solutions est que les algorithmes *canoniques* associés sont génériques et ne sont pas forcément efficaces pour traiter le cas particulier des contraintes d'inclusions. Déjà dans [Alechina et al., 2003], les auteurs introduisent des algorithmes *ad hoc* pour étudier finement la complexité de certains problèmes.

L'un des buts de ce chapitre est donc de proposer des algorithmes optimaux pour l'implication de contraintes. Dans cette optique, nous nous restreignons aux contraintes d'inclusions. Nous restreignons également le type des contraintes d'inclusions étudiées afin de définir des algorithmes spécialisés et d'étudier la complexité du problème de l'implication. Soient p et q deux requêtes régulières, u et v deux mots :

- la contrainte $p \preceq q$ est une *contrainte d'inclusion*,
- si p et q décrivent un langage fini, la contrainte est dite contrainte d'*inclusion finie* ,
- une contrainte de la forme $p \preceq u$ est dite *contrainte bornée*,
- une *contrainte de mots* est de la forme $u \preceq v$,
- et enfin la contrainte $u \equiv v$ (i.e. $u \preceq v \wedge v \preceq u$) est dite *égalité de mots*.

Il existe des contraintes sur les chemins d'un graphe qui sont plus générales que les contraintes d'inclusion. Nous ne les étudirons pas dans ce chapître, car le problème de l'implication est déjà connu pour être indécidable [Alechina et al., 2003].

Deux modèles de graphe co-existent dans la littérature : les données peuvent soit avoir une unique racine ([Buneman et al., 1999, Abiteboul et al., 2000, Alechina et al., 2003]), soit avoir plusieurs racines [Milo and Suciu, 1999]. Nous nous plaçons ici dans le cadre le plus général, celui des données à plusieurs racines. La majorité des résultats et des techniques proposés sont néanmoins similaires au cas des données enracinées. En effet, nous prouvons que le problème de l'implication pour des graphes enracinés est équivalent au problème de l'implication pour les graphes à plusieurs racines dès lors que toutes les requêtes apparaissant dans l'ensemble de contraintes $\mathcal{C}$ ne contiennent pas ε (ε-free). Comme dans le cas enraciné, le problème de l'implication est 2-EXPTIME. Nous donnons aussi un algorithme, PSPACE, de décision pour l'implication d'une contrainte $p \preceq q$ par un ensemble de contraintes bornées de la forme $p_i \preceq u_i$ où les p_i sont des requêtes et les u_i sont des mots. Nous montrons de plus que le problème est PSPACE-complet.

Dans le cas particulier des égalités de mots, nous avons un algorithme *ad-hoc* pour décider de l'implication de contraintes. On améliore l'algorithme cubique de [Buneman et al., 1999] qui utilise des contraintes de mots *inverse* en proposant un algorithme quasi-linéaire. On prouve que dans le cas très particulier des contraintes d'égalités de mots, le problème de l'implication est équivalent au problème de l'implication sur les modèles déterministes et complets.

Recherche d'un équivalent fini :

Une requête régulière a une borne finie (resp. a un équivalent fini) par rapport à un ensemble $\mathcal{C}$ de contraintes d'inclusions, s'il existe une requête régulière f telle que $\mathcal{C} \models p \preceq f$ (resp. $\mathcal{C} \models p \equiv f$) et $L(f)$, le langage décrit par f, est fini. Par exemple, comme toute donnée satisfaisant la contrainte `assuré.ayant-droit.voiture.gère` $\equiv$ `assuré.(ayant-droit.voiture.gère)`^{2} satisfait aussi la contrainte

`assuré.(ayant-droit.voiture.gère)`$^+$ $\equiv$ `assuré.ayant-droit.voiture.gère`, on déduit que la requête `assuré.(ayant-droit.voiture.gère)`$^+$ a un équivalent fini par rapport à l'ensemble de contraintes {`assuré.ayant-droit.voiture.gère` $\equiv$ `assuré.(ayant-droit.voiture.gère)`^{2}}. Plus généralement, si une requête q a un équivalent fini, on peut trouver une requête finie f dont le résultat coïncide avec le résultat de q.

Nous étendons les travaux de [André et al., 1999] qui étudie le problème des bornes finies à partir d'un ensemble de contraintes d'inclusions bornées. Nous donnons deux algorithmes. L'un calcule une borne finie de q si la requête q en possède une. L'autre calcule, si possible, une requête finie équivalente à q. Dans ces deux cas, nous construisons un transducteur qui transforme une requête en une requête plus grande (équivalente) au sens de l'inclusion de contraintes.

Comme dans le cas de l'implication, la complexité du problème est meilleure dans le cas des contraintes d'égalités de mots. En effet, il existe un algorithme PTIME qui décide si une requête régulière q a un équivalent fini (si la réponse est positive, on sait construire cette requête finie). Pour évaluer ce bon résultat théorique nous avons développé QRIC (Query Rewriting with Inclusion Constraints, [Debarbieux and Luchier, 2004]) un outil qui implémente cet algorithme. Nous avons utilisé QRIC, pour évaluer les optimisations possibles avec des contraintes d'égalités sur des *benchmarks*.

Modèle exact (fini) :

Le troisième point qui est abordé concerne les modèles de contraintes d'inclusions. Il est évident que le graphe complet réduit à une racine satisfait n'importe quel ensemble de contraintes. La problématique étudiée concerne les modèles exacts : un modèle de $\mathcal{C}$ est dit exact s'il satisfait uniquement les contraintes satisfaites par tous les modèles de $\mathcal{C}$. Autrement dit, une donnée D_c est un modèle exact de $\mathcal{C}$ si $D_c \models p \preceq q$ si et seulement si $\mathcal{C} \models p \preceq q$.

Nous expliquons pourquoi tout ensemble de contraintes d'inclusions a un modèle exact. On se pose donc la question : un ensemble de contraintes $\mathcal{C}$ a-t-il un modèle exact fini ? Ce modèle fini permet de tester si une contrainte est impliquée par un ensemble de contraintes. Il garantit aussi que toutes les requêtes régulières ont un équivalent fini par rapport à l'ensemble de contraintes $\mathcal{C}$.

Dans un premier temps, nous considérons le cas de contraintes bornées ($p \preceq u$ où p est une requête régulière et u est un mot). Dans ce cas, nous proposons une caractérisation décidable des ensembles $\mathcal{C}$ qui ont un modèle exact fini. De plus, nous donnons un algorithme effectif qui calcule un tel modèle quand il existe.

Deuxièmement, nous considérons uniquement les égalités de mots ($u \equiv v$ où u et v sont des mots). Dans ce cas, nous donnons un algorithme plus efficace pour décider de l'existence d'un modèle exact fini et pour la construction éventuelle d'un tel modèle.

3.2 Requêtes régulières de chemins

Cette section présente les données modélisées par un graphe orienté multi-racines dont les arcs sont orientés. A partir des chemins de ce graphe, on présente les requêtes régulières

et les contraintes d'inclusions. Pour terminer, quelques résultats généraux sur le problème de l'implication et sur les modèles exacts sont présentés.

Les notions suivantes sont celles utilisées par S. Abiteboul et V. Vianu dans [Abiteboul and Vianu, 1997].

Soit A un alphabet fini de labels.

Définition 3.1 *Une donnée semi-structurée est un triplet $D =< N, \texttt{Racines}, T >$ dans lequel :*
- *N est un ensemble dénombrable de nœuds*
- *`Racines`, l'ensemble des racines, est un sous-ensemble de N,*
- *T, l'ensemble des transitions, est un sous-ensemble dénombrable de $N \times A \times N$.*

Remarquons que si l'ensemble de nœuds N est fini alors la donnée D est finie. Dans ce cas, la taille de D, notée $|D|$ est la somme de son nombre de nœuds et de son nombre de transitions. De plus, comme N et T sont des ensembles dénombrables, tout nœud d'une donnée semi-structurée peut avoir un nombre infini d'arcs entrants et un nombre infini d'arcs sortants. Enfin, si l'ensemble de racines `Racines` est un singleton la donnée est dite **enracinée**.

Définition 3.2 *Une donnée semi-structurée $< N, \texttt{Racines}, T >$ est*
- déterministe *si elle est enracinée et si pour tout nœud n dans N, tout label l dans A, il y a au plus une transition (n, l, n') dans T.*
- complète *si l'ensemble* `Racines` *n'est pas vide et si pour tout nœud n dans N, tout label l dans A, il y a au moins une transition (n, l, n') dans T.*

Maintenant que les données semi-structurées sont définies, il est possible de définir les requêtes régulières.

Définition 3.3
- *Soient $D =< N, Racines, T >$ une donnée semi-structurée, u un mot de A^* et N' un sous-ensemble de N. On définit $D(N', u)$ par :*
  1. *si $u = \varepsilon$, alors $D(N', u) = N'$*
  2. *si $u = u'a$ (u' est un mot et a est un label appartenant à A)*
     $D(N', u) = D(D(N', u'), a) = \{n \in N \mid \exists n' \in D(N', u'), (n', a, n) \in T\}$
- *Une requête régulière p est une expression régulière sur l'alphabet A.*
- *$L(p)$ désigne le langage défini par p.*
- *Le résultat de la requête p sur la donnée D à partir de l'ensemble de noeuds N' est l'ensemble $D(N', p) = \bigcup_{u \in L(p)} D(N', u)$.*

Remarque 3.1 *Dans la suite de cette thèse, par similitude avec la définition donnée ci-dessus, si $\mathcal{A}$ est un automate de mots et u est un mot, alors $\mathcal{A}(u)$ désigne l'ensemble des états de $\mathcal{A}$ accessibles par le mot u (à partir d'un état inital de $\mathcal{A}$).*

Par convention, si N' est l'ensemble des racines, on note $D(N', u)$ par $D(u)$.

Cette définition permet de définir la composition de deux requêtes. Soient p et q deux requêtes régulières, évaluer la requête $p \circ q$ (composition de p par q) sur la donnée $D =< N, R, T >$ dans le contexte N', revient à calculer $D(N', p \circ q) = D(D(N', q), p)$.

Il reste à donner un algorithme, qui étant donnée une requête q, une donnée finie D et un ensemble de nœuds N', calcule $D(N', q)$ [Debarbieux, 2004]. Pour cela, nous utilisons la notion de produit cartésien, non pas comme outil qui permet de calculer l'intersection de deux langages, mais comme un algorithme qui construit une relation entre les nœuds de la donnée et les nœuds d'un automate A_q qui reconnaît la requête q. L'idée est d'utiliser le lien entre les chemins du produit et les chemins des automates d'origine, pour mettre en relation les nœuds sélectionnés par une requête q et les états finaux de A_q. En effet, on peut représenter le graphe fini D et l'ensemble N' comme un automate $D_{N'}$: transformons D en automate dans lequel tous les états sont finaux et dans lequel les noeuds de N' sont des états initiaux. On calcule le produit cartésien entre l'automate $D_{N'}$ et un automate $A_q =< A, Q_q, I_q, F_q, \delta >$ qui reconnaît la requête régulière q. On définit ensuite la relation R sur $N \times Q_q$ par (n, q_q) appartient à R si l'état (n, q_q) est un état accessible du produit cartésien. On construit enfin $D(N', q)$ en prenant l'ensemble des nœuds n tels qu'il existe un état final q_f de A_q vérifiant (n, q_f) appartient à R :

$$D(N', q) = \{n \in N \mid \exists q_f \in F_q \wedge (n, q_f) \in R\}$$

L'algorithme de Gluskov [Gluskov, 1961] permettant de construire en temps cubique, à partir d'une requête régulière p, un automate (dont la taille est quadratique dans la taille de p) qui reconnaît le langage $L(p)$, l'algorithme qui évalue une requête p sur un donnée D a une data complexité qui est en $O(max(|D|, |p|).|p|^2)$.

3.3 Les contraintes d'inclusions

Dans cette section, on donne la définition des contraintes d'inclusions et on présente les problèmes de l'implication et des modèles exacts finis :

Définition 3.4

- *Une* contrainte d'inclusion *est une expression de la forme $p \preceq q$ où p et q sont des requêtes régulières.*
- *Une donnée D* satisfait *(est modèle d') une contrainte d'inclusion $p \preceq q$, noté $D \models p \preceq q$, si l'ensemble de noeuds $D(p)$ est inclus dans $D(q)$.*
- *D satisfait un ensemble $\mathcal{C}$ de contraintes d'inclusions, noté $D \models \mathcal{C}$, si D satisfait toutes les contraintes présentes dans $\mathcal{C}$.*

Dans la suite de cette thèse, si p et q sont deux requêtes régulières, alors $p \equiv q$ désigne la conjonction $p \preceq q$ et $q \preceq p$.

3.3.1 Le problème de l'implication

Dans cette section, le problème de l'implication et le problème de l'équivalence entre requêtes sont prouvés comme étant 2-EXPTIME.

Pour commencer, définissons le problème de l'implication d'une contrainte d'inclusion $p \preceq q$ par un ensemble de contraintes $\mathcal{C}$.

Définition 3.5 *Un ensemble de contraintes d'inclusions $\mathcal{C}$ implique une contrainte d'inclusion $p \preceq q$ (ce qui se note $\mathcal{C} \models p \preceq q$) si tout modèle de $\mathcal{C}$ satisfait aussi la contrainte $p \preceq q$.*

Définition 3.6

- *Problème de l'implication :*
  Entrée : un ensemble de contraintes $\mathcal{C}$ et deux requêtes régulières p et q
  Sortie : vrai si et seulement si $\mathcal{C} \models p \preceq q$.
- *Problème de l'équivalence :*
  Entrée : un ensemble de contraintes $\mathcal{C}$ et deux requêtes régulières p et q
  Sortie : vrai si et seulement si $\mathcal{C} \models p \equiv q$.

Une propriété bien connue est que le problème de l'implication et le problème de l'équivalence sont équivalents. En effet, comme l'union $D(p) \cup D(q)$ est égale à $D(p+q)$, D est un modèle de $p \preceq q$ si et seulement si D est un modèle de $p+q \equiv q$. La propriété suivante montre comment transformer le problème de l'implication en un problème d'équivalence :

$$\mathcal{C} \models p \preceq q \text{ si et seulement si } \mathcal{C} \models q + p \equiv q \tag{3.1}$$

Dans le but d'étudier la décidabilité et la complexité de ce problème on ne considère, à partir de maintenant, que des ensembles finis de contraintes d'inclusions. Si $p \preceq q$ est une contrainte, alors $|p \preceq q|$ est la taille de cette contrainte. C'est la somme $|p| + |q|$ où $|p|$ est la longueur de l'expression régulière p. Pour un ensemble de contraintes $\mathcal{C}$, la taille de $\mathcal{C}$ est la somme des tailles de toutes les contraintes présentes dans $\mathcal{C}$.

La notion d'équivalence utilisée dans l'équation 3.1 est une "logspace many-one transformation". En effet construire l'expression $p+q$ à partir de p et q requiet un espace suplémantaire constant. Tous les résultats de compléxité valide pour l'implication le seront pour l'équivalence et vice-versa.

Dans [Abiteboul and Vianu, 1997], S. Abiteboul and V. Vianu ont prouvé le résultat suivant dans le cas des données enracinées et pour lesquelles chaque noeud a un nombre fini d'arcs sortants. Leur preuve peut facilement être adaptée au type de donnée que nous considérons.

Proposition 3.1 *Un ensemble $\mathcal{C}$ de contraintes d'inclusions implique une contrainte d'inclusion $p \preceq q$ si et seulement si pour toute donnée* **finie** *D telle que $D \models \mathcal{C}$, $D \models p \preceq q$.*

Preuve : Soit $\mathcal{C} = \{p_i \preceq q_i, 1 \leq i \leq n\}$ un ensemble de contraintes d'inclusions.

Si $\mathcal{C} \models p \preceq q$ alors pour toute donnée finie D telle que $D \models \mathcal{C}$, $D \models p \preceq q$ est le sens facile de l'équivalence.

Soient p_0 et q_0 deux requêtes régulières telles que $\mathcal{C} \not\models p_0 \preceq q_0$. Soit $D =< N_D, R_D, T_D >$ un contre exemple à $C \models p_0 \preceq q_0$ i.e. une donnée telle que $D \models \mathcal{C}$ et $D \not\models p_0 \preceq q_0$. Construisons, à partir de D, une donnée finie D_f telle que $D_f \models \mathcal{C}$ et $D_f \not\models p_0 \preceq q_0$.

Soit $\equiv$ la relation d'équivalence définie sur $A^* \times A^*$ par $u \equiv v$ si pour tout mot w, pour toute requête régulière q dans $\{p_i, q_i, 0 \leq i \leq n\}$ le mot uw appartient à $L(q)$ si et seulement

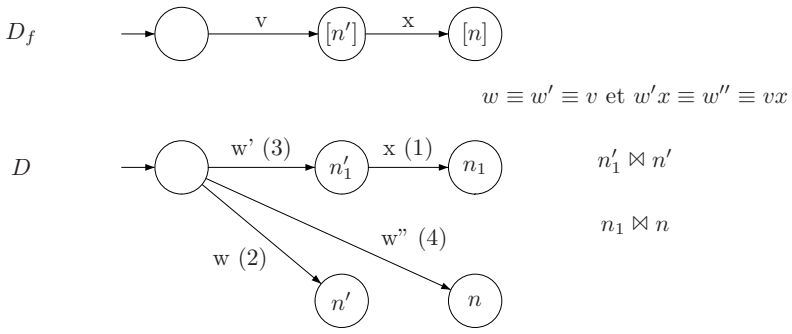
si vw appartient à $L(q)$ (la semi-congruence de Nerode). Soit $\bowtie$ la relation d'équivalence (la semi-congruence) définie sur $N_D \times N_D$ par $n \bowtie n'$ si pour tout mot u tel que n appartient à $D(u)$ il existe un mot v équivalent à u tel que n' appartienne à $D(v)$ (et vice versa). Notons $[n]$ la classe d'équivalence du noeud n.

Il est alors possible de définir $D_f =< N_f, R_f, T_f >$ une donnée finie par :
- *$N_f = \{[n] \mid n \in N_D\}$*
- *$R_f = \{[n] \mid \exists r \in R_D \; r \bowtie n\}$*
- *$T_f = \{([n'], x, [n]) \mid \exists n'_1, n_1 \; (n \bowtie n_1) \wedge (n' \bowtie n'_1) \wedge (n'_1, x, n_1) \in T_D\}$*

Dans un premier temps, on montre par récurrence la propriété suivante :

$$[n] \in D_f(u) \Rightarrow \exists v \equiv u \mid n \in D(v) \tag{3.2}$$

Travaillons sur la longueur de u. Si u est le mot vide alors $[n]$ est l'une des racines de D_f. Il existe donc un noeud r tel que r soit une racine de D et $n \bowtie r$. $n \bowtie r$ implique qu'il existe un mot v équivalent à ε tel que n appartienne à $D(v)$.



Si $u = vx$ il existe deux noeuds $[n]$ et $[n']$ de D_f tel que $[n']$ est accessible par v et $([n'], x, [n])$ est une transition de T_f (la figure ci-dessus représente les quatre étapes de la preuve).

1. *D'après la définition de T_f il existe deux noeuds n'_1 et n_1, équivalents respectivement à n' et à n, tel que (n'_1, x, n_1) appartienne à T.*
2. *L'hypothèse de récurrence garantit que n' est accessible par un mot w équivalent à v.*
3. *Comme n' est équivalent à n'_1 il existe un mot w' équivalent à w tel que n'_1 appartienne à $D(w')$ donc n_1 appartient à $D(w'x)$.*
4. *Comme n et n_1 sont équivalents il existe un mot w'' équivalent à $w'x$ tel que n appartienne à $D(w'')$. Comme w' est équivalent à v, $w'x$ est équivalent à vx et donc w'' est équivalent à u. On a montré que n est accessible par w''.*

Une seconde propriété importante est :

$$n \in D(u) \Rightarrow [n] \in D_f(u) \tag{3.3}$$

Raisonnons par récurrence sur la longueur de u. Si u est le mot vide alors n est l'une des racines de D et $[n]$ est l'une des racines de D_f. Si $u = vx$ alors n appartient à $D(u)$ ce qui signifie qu'il existe un noeud n' tel que n' appartienne à $D(v)$ et (n', x, n) est l'une des transitions de T_D. Par récurrence, $[n']$ appartient à $D_f(v)$ et par définition de T_f, $([n'], x, [n])$ est une transition de T_f i.e. $[n]$ est accessible par u.

En utilisant la définition de $\equiv$, il est maintenant possible d'utiliser 3.2 et 3.3 avec toutes les requêtes q de l'ensemble $\{p_i, q_i, 0 \leq i \leq n\}$. On obtient alors :

$$[n] \in D_f(q) \text{ si et seulement si } n \in D(q) \tag{3.4}$$

En effet, si $[n]$ appartient à $D_f(q)$, il existe un mot u_q de $L(q)$ tel que $[n] \in D_f(u_q)$. D'après la relation (3.2) il existe un mot w équivalent à u_q qui atteint n dans D. Par définition de $\equiv$, u_q appartient à $L(q)$ implique que w appartienne à $L(q)$ et donc n appartient à $D(q)$.

On peut alors terminer la preuve. En effet, $D_f \models \mathcal{C}$ car pour i plus grand que 1, $[n] \in D_f(p_i)$ implique que $n \in D(p_i)$ et donc $n \in D(q_i)$ (car $D \models \mathcal{C}$) d'où $[n] \in D_f(q_i)$. Enfin $D_f \not\models p_0 \preceq q_0$, car si $D_f \models p_0 \preceq q_0$ alors $D \models p_0 \preceq q_0$ (p_0 et q_0 satisfont 3.4) ce qui est contradictoire avec les hypothèses. On a donc réussit à construire une donnée finie D_f telle que $D_f \models \mathcal{C}$ et $D_f \not\models p_0 \preceq q_0$. ◄

Cette proposition est essentielle pour montrer que le problème de l'implication est décidable. En effet, si un ensemble de contraintes d'inclusions $\mathcal{C}$ n'implique pas une contrainte d'inclusion $p_0 \preceq q_0$, il existe une donnée finie D_f telle que D_f soit un modèle de $\mathcal{C}$ et que D_f ne soit pas un modèle de $p_0 \preceq q_0$. Nous savons, grâce à la preuve de la proposition 3.1, que la taille de D_f peut être exponentiellement plus grande que la somme $|\mathcal{C}| + |p_0| + |q_0|$ (i.e. $|D_f|$ est en $O(2^{|\mathcal{C}|+|p_0|+|q_0|})$). Dans le but de décider si $\mathcal{C} \models p_0 \preceq q_0$ il suffit de tester sur un nombre fini de données s'il existe un contre exemple à $\mathcal{C} \models p_0 \preceq q_0$. On a en fait montré le théorème suivant :

Théorème 3.1 *Le problème de l'implication pour $\mathcal{C}$, p, q est 2-EXPTIME.*

La preuve de ce théorème montre en fait que le problème est NEXPTIME (classe qui regroupe les problèmes qui peuvent être résolus en temps exponentiel pour une machine non déterministe : NEXPTIME $= \cup_{k \geq 0} NTIME(2^{n^k})$).

La proposition suivante compare le cas des graphes enracinés et celui des graphes à plusieurs racines. Pour faire cette comparaison, nous avons besoin de distinguer le cas où les langages présents dans les contraintes d'inclusions ne contiennent pas le mot vide (on dit alors qu'ils sont ε-free). On note $\models^1$ l'implication sur les données enracinées. Dans le cas de contraintes de la forme $p_i \preceq q_i$ où les p_i et les q_i sont ε-free, alors l'implication pour les données à plusieurs racines est équivalente à l'implication pour les données enracinées :

$\{a \preceq \varepsilon\} \not\models a \equiv aa$ $\qquad$ $\{\varepsilon \preceq a + b\} \not\models \varepsilon \preceq aa + bb$

FIG. 3.3 – Différences entre $\models$ et $\models^1$

Proposition 3.2 *Soient $\mathcal{C} = \{p_i \preceq q_i, 1 \leq i \leq n\}$ un ensemble de contraintes d'inclusions, p_0 et q_0 deux requêtes régulières telles que pour tout i plus grand que 1, p_i et q_i sont ε-free. On a alors :*

$$\mathcal{C} \models p_0 \preceq q_0 \text{ si et seulement si } \mathcal{C} \models^1 p_0 \preceq q_0$$

Preuve : Il est évident que $\mathcal{C} \models p_0 \preceq q_0$ implique que $\mathcal{C} \models^1 p_0 \preceq q_0$. Étudions la réciproque :

Supposons que $\mathcal{C} \models^1 p_0 \preceq q_0$ et montrons que $\mathcal{C} \models p_0 \preceq q_0$. Pour cela il faut montrer que si $D =< N, R, T >$ est un modèle de $\mathcal{C}$ alors D est un modèle de la contrainte $p_0 \preceq q_0$. Construisons le modèle enraciné $D_r =< N \cup \{r\}, \{r\}, T \cup \{(r, x, n) \mid \exists r' \in R \wedge (r', x, n) \in T\} >$ (dans la théorie des automates, cette construction s'appelle standardisation). On supprime dans D_r tous les noeuds qui sont devenus inaccessibles et on obtient un résultat bien connu :

$$\forall u \in A^+ \; D(u) = D_r(u) \tag{3.5}$$

Pour toute requête régulière q, q^ε désigne une requête régulière telle que $L(q^\varepsilon) = L(q) \setminus \{\varepsilon\}$. On déduit de (3.5) que pour tout couple de requêtes (p, q) $D \models p^\varepsilon \preceq q^\varepsilon$ si et seulement si $D_r \models p^\varepsilon \preceq q^\varepsilon$. Comme toutes les requêtes présentes dans $\mathcal{C}$ sont ε-free et que D est un modèle de $\mathcal{C}$, D_r est un modèle de $\mathcal{C}$. On a démontré que $D_r \models p_0 \preceq q_0$.

Comme ε est le seul mot atteignant le nœud r dans la donnée D_r, on a $D_r(p_0^\varepsilon) = D_r(p_0) \setminus \{r\}$ et $D_r(q_0^\varepsilon) = D_r(q_0) \setminus \{r\}$. On déduit de $D_r \models p_0 \preceq q_0$ que $D \models p_0^\varepsilon \preceq q_0^\varepsilon$ (cf. équation 3.5).

Si ε n'appartient pas à $L(p_0)$ alors $D \models p_0 \preceq p_0^\varepsilon$. De plus $D \models q_0^\varepsilon \preceq q_0$ étant toujours vrai, on peut conclure que $D \models p_0 \preceq q_0$.

Enfin si ε est un mot de $L(p_0)$ il faut montrer qu'il appartient aussi à $L(q_0)$. Comme r appartient à $D_r(p_0)$ et que $D_r \models p_0 \preceq q_0$, le noeud r est accessible par q_0. r n'étant accessible que par ε, ce mot appartient à $L(q_0)$. On sait déjà que $D(p_0^\varepsilon)$ est inclus dans $D(q_0^\varepsilon)$. On a donc $D(p_0^\varepsilon) \cup R$ est inclus dans $D(q_0^\varepsilon) \cup R$ ce qui signifie que $D \models p_0 \preceq q_0$. ◀

Remarque 3.2 *La proposition 3.2 devient fausse si l'une des contraintes contient le mot vide :*

Par exemple pour tout mot u, pour toute donnée enracinée D, $D \models u \preceq \varepsilon$ signifie soit que $D(u)$ est vide soit que $D(u)$ est la racine de la donnée. On en déduit que $\{a \preceq \varepsilon\} \models^1 a \equiv aa$ alors que $\{a \preceq \varepsilon\} \not\models a \equiv aa$ comme le prouve la donnée représentée par la figure 3.3.

Si ε apparaît du coté gauche de la contrainte d'inclusion, la proposition 3.2 est toujours fausse. En effet, $\{\varepsilon \preceq a + b\} \models^1 \varepsilon \preceq aa + bb$ car toute donnée enracinée D satisfaisant $\{\varepsilon \preceq a + b\}$ satisfait soit $\varepsilon \preceq a$ soit $\varepsilon \preceq b$. Par contre $\{\varepsilon \preceq a + b\} \not\models \varepsilon \preceq aa + bb$ (figure 3.3).

3.3.2 Modèles de contraintes

Dans la sous-section précédente, nous avons introduit la notion de contrainte d'inclusion et la notion de modèle d'un ensemble de contraintes d'inclusions. Bien sûr, tout ensemble de contraintes d'inclusions a un modèle : le modèle complet réduit à une racine qui satisfait toutes les contraintes (figure 3.4). Dans cette sous-section, nous définissons la notion de modèle exact d'un ensemble de contraintes d'inclusions et nous montrons que tout ensemble de contraintes d'inclusions a un modèle exact.

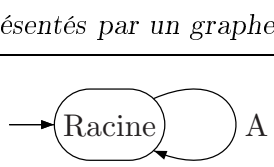


FIG. 3.4 – Tout ensemble de contraintes d'inclusions a un modèle

Définition 3.7 *Soit $\mathcal{C}$ un ensemble de contraintes d'inclusions. Une donnée $D_\mathcal{C}$ est un modèle exact de $\mathcal{C}$ si pour toutes requêtes régulières p et q*

$$D_\mathcal{C} \models p \preceq q \text{ si et seulement si } \mathcal{C} \models p \preceq q$$

Nous avons déjà montré que le problème de l'implication et le problème de l'équivalence sont équivalents (3.1) et donc :

Corollaire 3.1 *Soient $\mathcal{C}$ un ensemble de contraintes d'inclusions et $D_\mathcal{C}$ une donnée. $D_\mathcal{C}$ est un modèle exact de $\mathcal{C}$ si et seulement si*

$$D_\mathcal{C} \models p \equiv q \text{ si et seulement si } \mathcal{C} \models p \equiv q$$

Nous pouvons maintenant montrer que tout ensemble de contraintes d'inclusions a un modèle exact :

Proposition 3.3 *Tout ensemble $\mathcal{C}$ de contraintes d'inclusions a un modèle exact.*

Preuve : Soit $M_\mathcal{C}$ l'ensemble dénombrable de tous les modèles finis (à un isomorphisme près) de $\mathcal{C}$ i.e. $M_\mathcal{C} = \{D_i =< N_i, R_i, T_i > |\ D_i \models \mathcal{C} \wedge (|N_i| < \infty)\}$. Sans restreindre la généralité du problème, on impose que l'intersection $N_i \cap N_j$ est vide si i et j sont différents.

Nous prouvons maintenant que $D =< \underset{i\geq 0}{\cup} N_i, \underset{i\geq 0}{\cup} R_i, \underset{i\geq 0}{\cup} T_i >$ est un modèle exact de $\mathcal{C}$.

Il est évident que

$$D \models p \preceq q \text{ si et seulement si } \forall i \geq 0\ D_i \models p \preceq q$$

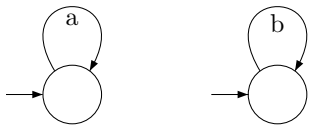
Donc D est un modèle de $\mathcal{C}$.

Soient p et q deux requêtes telles que $\mathcal{C} \not\models p \preceq q$. Nous savons déjà, d'après la proposition 3.1, qu'il existe un modèle fini D_f tel que $D_f \models \mathcal{C}$ et $D_f \not\models p \preceq q$. Par définition de $M_\mathcal{C}$, D_f appartient à $M_\mathcal{C}$ et donc D n'est pas un modèle de $p \preceq q$. ◄

Malheureusement, nous ne sommes pas capables de caractériser les ensembles qui ont un modèle exact fini.

Remarque 3.3 *Il est déjà connu de [Debarbieux et al., 2004] que l'ensemble $\{a \preceq \varepsilon, b \preceq \varepsilon\}$ n'a pas de modèle exact enraciné. Cette remarque met en évidence l'une des différences majeures entre les modèles enracinés et les modèles à plusieurs racines.*

Considérons les données D_a et D_b de la figure 3.5. Ces deux données satisfont $\mathcal{C}$ et nous allons les utiliser comment contre exemple. Soit $D_\mathcal{C}$ un modèle exact de $\mathcal{C}$. Comme $\mathcal{C} \models a \preceq \epsilon$ et $\mathcal{C} \models b \preceq \epsilon$ il n'y a que quatre valeurs possibles pour le couple $(D_\mathcal{C}(a), D_\mathcal{C}(b))$:



D_a D_b

FIG. 3.5 – Les données D_a et D_b.

$D_{\mathcal{C}}(a)$	$D_{\mathcal{C}}(b)$	*conséquence*
$\emptyset$	$\emptyset$	$D_{\mathcal{C}} \models a \equiv_{\mathcal{C}} b$ *mais* $\mathcal{C} \not\models a \equiv_{\mathcal{C}} b$ *(cf.* D_a*)*
$\emptyset$	$\{root\}$	$D_{\mathcal{C}} \models b \equiv_{\mathcal{C}} \epsilon$ *mais* $\mathcal{C} \not\models b \equiv_{\mathcal{C}} \epsilon$ *(cf.* D_a*)*
$\{root\}$	$\emptyset$	$D_{\mathcal{C}} \models a \equiv_{\mathcal{C}} \epsilon$ *mais* $\mathcal{C} \not\models a \equiv_{\mathcal{C}} \epsilon$ *(cf.* D_b*)*
$\{root\}$	$\{root\}$	$D_{\mathcal{C}} \models a \equiv_{\mathcal{C}} b \equiv_{\mathcal{C}} \epsilon$ *mais* $\mathcal{C} \not\models a \equiv_{\mathcal{C}} b \equiv_{\mathcal{C}} \epsilon$ *(cf.* D_b*)*

Il est donc impossible de trouver un modèle exact pour $\mathcal{C}$.

3.4 Les contraintes d'inclusions bornées

Dans cette section, on ne considère que le cas d'un ensemble fini de contraintes d'inclusions de la forme $p \preceq u$ où p est une requête régulière et u est un mot. De telles contraintes sont dites contraintes d'inclusions bornées. Elles ont été introduites par Y. André, F. Bossut et A.C. Caron dans [André et al., 1999]. Dans ce cas et en généralisant les travaux de S. Abiteboul et de V. Vianu [Abiteboul and Vianu, 1997] on associe à un ensemble de contraintes d'inclusions bornées $\mathcal{C}$ un système de réécriture préfixe tel que u se réécrit en v si et seulement si $\mathcal{C}$ implique la contrainte $u \preceq v$. Grâce à ce système de réécriture, il est possible de caractériser les ensembles de contraintes d'inclusions bornées qui ont un modèle exact fini. Cette technique donne aussi un moyen uniforme pour décider de l'implication de contraintes et pour trouver un équivalent fini d'une requête.

3.4.1 Contraintes d'inclusions et réécriture préfixe

Tout d'abord remarquons qu'un ensemble fini de contraintes d'inclusions bornées peut être vu comme une représentation finie d'une relation binaire sur A^* . Plus précisément, si $\mathcal{C} = \{p_i \preceq u_i, 1 \leq i \leq n\}$ est un ensemble de contraintes d'inclusions bornées et si u et v sont deux mots on dit que u est en relation avec v s'il existe un indice i tel que u appartienne au langage $L(p_i)$ et v soit égal au mot u_i.

Définition 3.8 *Soit* $\mathcal{C} = \{p_i \preceq u_i, 1 \leq i \leq n\}$ *un ensemble de contraintes d'inclusions bornées sur un alphabet* A*. La relation* $\underset{\mathcal{C}}{\longrightarrow}$ *est définie sur les mots par* $u \underset{\mathcal{C}}{\longrightarrow} v$ *s'il existe* i *tel que* u *appartienne à* $L(p_i)$ *et* $v = u_i$*. Par extension,* $\underset{\mathcal{C}}{\longrightarrow}$ *est aussi la congruence droite de cette relation. Enfin* $\overset{*}{\underset{\mathcal{C}}{\longrightarrow}}$ *est la clôture réflexive et transitive de* $\underset{\mathcal{C}}{\longrightarrow}$.

Cette relation est donc une relation de réécriture préfixe [Caucal, 1990] basée sur un système infini. Comme le prouve la proposition suivante, elle coïncide avec l'implication de contraintes de mots :

Proposition 3.4 *Soit $\mathcal{C}$ un ensemble de contraintes d'inclusions bornées. Pour tout couple de mots (u, v), $u \xrightarrow[\mathcal{C}]{*} v$ si et seulement si $\mathcal{C} \models u \preceq v$.*

La preuve de cette proposition utilise un modèle exact qui est proche de celui utilisé par S. Abiteboul et V. Vianu dans [Abiteboul and Vianu, 1997] :

Définition 3.9 *A tout ensemble $\mathcal{C}$ de contraintes d'inclusions bornées on associe la donnée infinie $I_{\mathcal{C}} =< N, Racines, T >$ définie par :*

- $N = \{u \mid u \in A^*\}$
- $Racines = \{u \mid u \xrightarrow[\mathcal{C}]{*} \varepsilon\}$
- $T = \{(u, x, v) \mid v \xrightarrow[\mathcal{C}]{*} ux\}$

Exemple 3.4.1 : Soit l'ensemble $\mathcal{C} = \{ab^* \preceq ba, b^+ \preceq a, a(aa)^*b \preceq a, b \preceq \varepsilon\}$ de contraintes d'inclusions bornées. La figure 3.6 présente une partie (finie) du modèle $I_{\mathcal{C}}$ correspondant.

Comme $b \xrightarrow[\mathcal{C}]{*} \varepsilon$, b est l'une des racines de $I_{\mathcal{C}}$. Comme $bb \xrightarrow[\mathcal{C}]{} a \xrightarrow[\mathcal{C}]{} ba \xrightarrow[\mathcal{C}]{} aa$ ($ba \xrightarrow[\mathcal{C}]{} aa$ car $b \xrightarrow[\mathcal{C}]{} a$ et par congruence droite), (a, a, bb) est une transition de ce modèle. De même $b \xrightarrow[\mathcal{C}]{*} ba$ entraîne que $(b, a, b) \in T$.

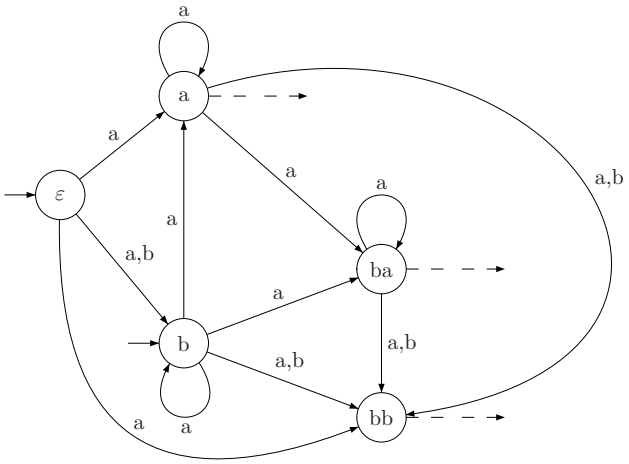


FIG. 3.6 – Une partie du modèle exact $I_{\mathcal{C}}$

Une propriété intéressante du graphe $I_{\mathcal{C}}$ est que l'ensemble des nœuds accessibles par un mot u coïncide avec les ancêtres de u pour la relation de réécriture.

Lemme 3.1 $I_{\mathcal{C}}(u) = \{v \mid v \xrightarrow[\mathcal{C}]{*} u\}$

Preuve : Raisonnons par récurrence sur la longueur de u
- *si u est le mot vide $I_{\mathcal{C}}(\varepsilon) = Racines = \{v \mid v \xrightarrow[\mathcal{C}]{*} \varepsilon\}$.*
- *si u est de la forme $u'.x$ alors un mot v appartient à $I_{\mathcal{C}}(u)$ s'il existe un mot v' de $I_{\mathcal{C}}(u')$ et une transition (v', x, v) dans $I_{\mathcal{C}}$. Par récurrence, on sait que $v' \xrightarrow[\mathcal{C}]{*} u'$ et donc $v'x \xrightarrow[\mathcal{C}]{*} u'x$. Une transition (v', x, v) dans $I_{\mathcal{C}}$ signifie que $v \xrightarrow[\mathcal{C}]{*} v'x$, on a donc montré que $v \xrightarrow[\mathcal{C}]{*} u$. Réciproquement, si $v \xrightarrow[\mathcal{C}]{*} u$ alors $v \xrightarrow[\mathcal{C}]{*} u'.x$ et donc (u', x, v) est une transition de $I_{\mathcal{C}}$. Comme $u' \xrightarrow[\mathcal{C}]{*} u'$, on sait par récurrence que $u' \in I_{\mathcal{C}}(u')$ i.e. v appartient à $I_{\mathcal{C}}(u)$.*

◄

On peut alors faire le lien entre les contraintes de mots satisfaites par $I_{\mathcal{C}}$ et le système de réécriture.

Lemme 3.2 *$I_{\mathcal{C}} \models u \preceq v$ si et seulement si $u \xrightarrow[\mathcal{C}]{*} v$.*

Preuve : Supposons que $I_{\mathcal{C}} \models u \preceq v$, i.e. $I_{\mathcal{C}}(u) \subseteq I_{\mathcal{C}}(v)$. D'après le lemme précédent, u appartient à $I_{\mathcal{C}}(u)$, et donc u appartient à $I_{\mathcal{C}}(v)$. En utilisant une fois de plus ce lemme on obtient $u \xrightarrow[\mathcal{C}]{*} v$.

Réciproquement, supposons que $u \xrightarrow[\mathcal{C}]{*} v$. Soit w un nœud de $I_{\mathcal{C}}(u)$, $w \xrightarrow[\mathcal{C}]{*} u$ et donc $w \xrightarrow[\mathcal{C}]{*} v$; D'après le lemme précédant $w \in I_{\mathcal{C}}(v)$, ce qui signifie que $I_{\mathcal{C}} \models u \preceq v$. ◄

Nous sommes maintenant en mesure de montrer que $I_{\mathcal{C}}$ est bien un modèle exact de $\mathcal{C}$.

Lemme 3.3 *$I_{\mathcal{C}} \models u \preceq v$ si et seulement si $\mathcal{C} \models u \preceq v$.*

Preuve : Premièrement, $I_{\mathcal{C}} \models \mathcal{C}$ car s'il existe i tel que u appartienne à $L(p_i)$, $v = u_i$ et $p_i \preceq u_i \in \mathcal{C}$, alors $u \xrightarrow[\mathcal{C}]{*} v$ i.e. $I_{\mathcal{C}}(u) \subseteq I_{\mathcal{C}}(v)$. Donc si $\mathcal{C} \models u \preceq v$ alors $I_{\mathcal{C}} \models u \preceq v$.

Réciproquement, $I_{\mathcal{C}} \models u \preceq v$ entraîne que $u \xrightarrow[\mathcal{C}]{*} v$. Soit $\preceq_{\mathcal{C}}$ la relation définie par $u \preceq_{\mathcal{C}} v$ si $\mathcal{C} \models u \preceq v$. La relation $\preceq_{\mathcal{C}}$ contient $\xrightarrow[\mathcal{C}]{}$, est transitive et close par congruence droite. Elle contient donc $\xrightarrow[\mathcal{C}]{*}$; donc $u \xrightarrow[\mathcal{C}]{*} v$ implique que $u \preceq_{\mathcal{C}} v$ i.e. $\mathcal{C} \models u \preceq v$. ◄

Ces deux derniers lemmes permettent de prouver la proposition 3.4. Le lemme suivant permet de faire le lien entre les contraintes de mots et les contraintes d'inclusions impliquées par un ensemble de contraintes bornées.

Lemme 3.4 *Si $I_{\mathcal{C}} \models u \preceq q$, il existe un mot v de $L(q)$ tel que $I_{\mathcal{C}} \models u \preceq v$.*

Preuve : En effet, si $I_{\mathcal{C}} \models u \preceq q$, $I_{\mathcal{C}}(u)$ est inclus dans $I_{\mathcal{C}}(q)$; en particulier, le nœud u appartient à $I_{\mathcal{C}}(q)$, i.e. il existe un mot v de $L(q)$ tel que u appartienne à $I_{\mathcal{C}}(v)$; donc, d'après le lemme 3.1, $u \xrightarrow[\mathcal{C}]{*} v$ i.e. (lemme 3.2) $I_{\mathcal{C}} \models u \preceq v$. ◄

La proposition suivante résume tous ces lemmes et elle n'est plus vraie si on considère des contraintes non bornées :

Proposition 3.5 *Soient $\mathcal{C}$ un ensemble de contraintes d'inclusions bornées et q une requête régulière. Les propriétés suivantes sont équivalentes :*

1. $\mathcal{C} \models u \preceq q$
2. *il existe un mot v de $L(q)$ tel que* $\mathcal{C} \models u \preceq v$
3. *il existe un mot v de $L(q)$ tel que* $u \xrightarrow[\mathcal{C}]{*} v$

Preuve : D'après les lemmes 3.2 et 3.3, les points 2 et 3 sont équivalents. De même le point 2 implique le point 1 d'après la définition de l'implication. Il reste à montrer que 1 implique 2.

Si $\mathcal{C} \models u \preceq q$, on déduit de $I_{\mathcal{C}} \models \mathcal{C}$ que $I_{\mathcal{C}} \models u \preceq q$. D'après le lemme 3.4, il existe donc un mot v de $L(q)$ tel que $I_{\mathcal{C}} \models u \preceq v$. Le lemme 3.3 permet de conclure que $\mathcal{C} \models u \preceq v$.

◄

Remarquons que, comme $\mathcal{C} \models p \preceq q$ si et seulement si pour tout mot u de $L(q)$ $\mathcal{C} \models u \preceq q$, $I_{\mathcal{C}}$ est un modèle exact de $\mathcal{C}$:

Corollaire 3.2 *$I_{\mathcal{C}} \models p \preceq q$ si et seulement si $\mathcal{C} \models p \preceq q$*

Dans le cas des contraintes d'inclusions bornées, la proposition 3.2 peut être étendue aux contraintes $p_i \preceq u_i$ avec les p_i qui ne sont pas nécessairement ε-free. En effet, la proposition 3.5 étend un résultat que nous avons montré dans [Debarbieux et al., 2003] : si $\mathcal{C}$ est un ensemble de contraintes d'inclusions bornées dans lequel aucun des u_i n'est le mot vide alors $\mathcal{C} \models^1 u \preceq v$ si et seulement si $u \xrightarrow[\mathcal{C}]{*} v$.

Corollaire 3.3 *Soit $\mathcal{C} = \{p_i \preceq u_i, 1 \leq i \leq n\}$ un ensemble de contraintes d'inclusions bornées dans lequel les u_i ne sont pas le mot vide. Soient p une requête régulière et u un mot. $\mathcal{C} \models p \preceq u$ si et seulement si $\mathcal{C} \models^1 p \preceq u$.*

L'exemple suivant montre que la proposition 3.5 (et les lemmes associés) n'est plus vraie si l'on considère des contraintes d'inclusions non bornées :

Exemple 3.4.2 : La donnée de la figure 3.7 satisfait la contrainte $a^+ \preceq (b+c)$ mais ne satisfait ni la contrainte $a \preceq b$ ni la contrainte $a \preceq c$. Par conséquent, $\{a^+ \preceq (b+c)\}$ n'impique ni $a \preceq b$ ni $a \preceq c$.

Théorie de la réécriture préfixe

Comme les contraintes d'inclusions bornées sont simulées par la réécriture préfixe, les propriétés concernant ces contraintes peuvent être exprimées avec des propriétés de la réécriture. Dans cette thèse nous ne considérons que la réécriture préfixe de mots, qui est un cas particulier de la réécriture close (i.e. les termes n'ont pas de variables). Büchi [Büchi, 1960] fût le premier à étudier des systèmes de réécriture pour les mots, qui réécrivent à partir de la fin

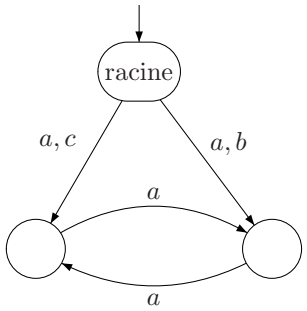


FIG. 3.7 – $\mathcal{C} \models a^+ \preceq (b + c)$ mais $\mathcal{C} \not\models a \preceq b$ et $\mathcal{C} \not\models a \preceq c$

du mot. Il a en particulier montré que le langage obtenu par toutes les dérivations d'un mot est reconnaissable. Plus récemment, les auteurs de [Dauchet and Tison, 1990] ont étudié les systèmes de réécriture clos ; ils ont introduit la notion de GTT (ground tree transducer) qui met en évidence le lien entre la réécriture préfixe et les automates.

Effectivement, l'idée générale est d'associer à un système logique une classe d'automates (automate de mots, automate de mots infinis, automate à pile, automate d'arbres, automate de graphes...) pour obtenir des algorithmes de décisions sur les propriétés du système. Le meilleur exemple est le théorème de Rabin qui prouve la décidabilité de la théorie monadique du second ordre en utilisant des automates d'arbres (sur des arbres infinis). Beaucoup d'autres résultats difficiles découlent de cette réduction.

Trois classes de théories décidables sont utilisées dans cette thèse :

1. La théorie du premier ordre
2. La théorie du deuxième ordre
3. Les relations reconnaissables

On peut définir une théorie dans laquelle les atomes sont (L est un langage reconnaissable, u et v sont deux mots et $<$ un ordre total sur les mots) :

$$u \in L \qquad \text{ou} \qquad u < v \qquad \text{ou} \qquad u \xrightarrow[\mathcal{C}]{1} v$$

A partir de ces formules atomiques, on construit usuellement les théories du premier et du deuxième ordre :

- Les formules du premier ordre sont les suivantes, et uniquement les suivantes (appelons E l'ensemble des formules et V_1 l'ensemble des variables) :
  - a où a est une formule atomique
  - $e_1 \wedge e_2$ si $e_1, e_2 \in E$
  - $e_1 \vee e_2$ si $e_1, e_2 \in E$
  - $e_1 \rightarrow e_2$ si $e_1, e_2 \in E$
  - $\neg e$ si $e \in E$
  - $\forall x(e)$ si $e \in E$ et $x \in V_1$
  - $\exists x(e)$ si $e \in E$ et $x \in V_1$

  Une variable x de V_1 s'interprète comme un mot du langage A^*
- Pour les formules monadiques du deuxième ordre (MSO), V_2 un deuxième ensemble de variables est introduit. On ajoute à la description précédente deux règles :

– $\forall X(e)$ si $e \in E$ et $X \in V_2$
– $\exists X(e)$ si $e \in E$ et $X \in V_2$

Une variable X de V_2 s'interprète comme un ensemble de mots de A^*. Classiquement, les variables appartenant à V_1 seront écrites en minuscule et celles appartenant à V_2 en majuscule.

Donnons quelques exemples de formules MSO :

– On peut tester si un ensemble X est inclus dans un ensemble Y :

$$X \subseteq Y =_{def} \forall x(x \in X \rightarrow x \in Y)$$

– On peut alors tester si deux ensembles sont égaux :

$$X = Y =_{def} X \subseteq Y \wedge Y \subseteq X$$

– On peut tester si un ensemble est vide :

$$X = \emptyset =_{def} \forall Y(Y \subseteq X \rightarrow X = Y)$$

– On peut tester si un ensemble est un singleton :

$$Sing(X) =_{def} X \neq \emptyset \wedge (\forall Y\ Y \subseteq X \Rightarrow (Y = X \vee Y = \emptyset))$$

– A partir de l'ordre total $<$ sur les mots (ordre alphabétique), on peut tester si un ensemble X est fini :

$$fini(X) =_{def} \exists y \forall x(x \in X \rightarrow x < y)$$

– La clôture transitive et réflexive de $\xrightarrow[\mathcal{C}]{1}$, notée $\xrightarrow[\mathcal{C}]{*}$, peut s'exprimer avec une formule MSO. $u \xrightarrow[\mathcal{C}]{*} v$ se définit avec la formule :

$$u \xrightarrow[\mathcal{C}]{*} v =_{def} \forall X((u \in X \wedge \forall x \forall y((x \in X \wedge x \xrightarrow[\mathcal{C}]{1} y) \rightarrow y \in X)) \rightarrow v \in X)$$

En utilisant des automates sur les arbres infinis et remarquant que le système $\xrightarrow[\mathcal{C}]{1}$ est un système infini comme défini dans [Caucal, 1990], on montre que la théorie monadique du second ordre (MSO) de la réécriture préfixe est décidable [Caucal, 1992].

L'idée des relations reconnaissables est de coder un n-uplet de mots sur un alphabet A par un mot de l'alphabet $A \cup \{\perp\} \times \ldots \times A \cup \{\perp\}$, où $\perp$ est un nouveau symbole. Ce codage est obtenu en superposant les n mots et en les alignant à partir de la fin. Par exemple, le triplet (ab, aaa, b) est codé par le mot $[\perp, a, \perp][a, a, \perp][b, a, b]$. Formellement, un morphisme π_j est défini sur $([A \cup \{\perp\}]^n)^*$ par $\pi_j[a_1, a_2, ...a_n] = a_j$, si a_j appartient à A, $\pi_j[a_1, a_2, ...a_n] = \epsilon$ si $a_j = \perp$. Maintenant notons Cor le langage reconnaissable de tous les mots $u_1...u_n$ -sur l'alphabet $[A \cup \{\perp\}]^n$- satisfaisant les deux propriétés suivantes :

– pour tout j, si $\pi_j(u_k)$ est une lettre de A alors $\pi_j(u_l)$ appartient aussi A pour tout $l > k$
– Il existe un indice j tel que $\pi_j(u_1)$ appartienne à A.

On définit alors la notion de relation reconnaissable par : une relation n-aire R de $A^\star \times \ldots \times A^\star$ est reconnaissable s'il existe un automate $\mathcal{M}$ sur l'alphabet $A \cup \{\perp\} \times \ldots \times A \cup \{\perp\}$ qui reconnaît le langage

$$L(\mathcal{M}) = \{u \in Cor \mid (\pi_1(u), \cdots, \pi_n(u)) \in R\}$$

Cette définition correspond à la définition de relation reconnaissable dans les arbres [Comon et al., 1997], si on considère un mot comme un arbre (un fil) dont la racine est associée avec la fin du mot.

Rec, l'ensemble des relations reconnaissables, hérite des bonnes propriétés des langages reconnaissables [Comon et al., 1997] :

– *Rec* est clos par union, intersection, complémentation, cylindrification et projection.[8]

– On sait décider si une relation est vide et si une relation est finie.

Les résultats présents dans [Comon et al., 1997] sont même beaucoup plus intéressants, puisqu'on obtient qu'une relation de réécriture préfixe dans laquelle tous les membres droits sont des relations reconnaissables et les membres gauches sont des mots, est reconnaissable. Ils ont en particulier montré que la relation binaire $\xrightarrow[\mathcal{C}]{*}$ est reconnaissable. Signalons enfin que l'automate qui reconnaît $\xrightarrow[\mathcal{C}]{*}$ peut se construire avec un algorithme PTIME.

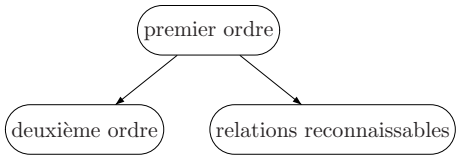


FIG. 3.8 – Expressivité des théories

On sait que ces trois théories sont décidables. Étudions leur expressivité (figure 3.8). La théorie du deuxième ordre est strictement plus expressive que la théorie du premier ordre pour la réécriture préfixe. De même, la théorie des relations reconnaissables est strictement plus expressive que la théorie du premier ordre pour la réécriture préfixe. Par contre MSO et les relations reconnaissables sont incomparables.

On peut maintenant utiliser les résultats montrés dans [Dauchet and Tison, 1990] concernant la décidabilité de la théorie du premier ordre, dans laquelle les constantes sont des termes clos et les prédicats sont des relations reconnaissables.

Lemme 3.5 *Soit $\mathcal{C} = \{p_i \preceq u_i, 1 \leq i \leq n\}$ un ensemble de contraintes d'inclusions bornées et deux requêtes régulières p et q.*

1. *Le problème de l'implication pour $\mathcal{C}$, p et q est décidable (PSPACE),*
2. *Le problème de l'équivalence pour $\mathcal{C}$, p et q est décidable (PSPACE),*

Preuve : Chacune de ces propriétés peut être exprimée avec une formule (monadique) de la théorie de la réécriture préfixe :

1. $\mathcal{C} \models p \preceq q =_{def} \forall u_p \in L(p)\ \exists u_q \in L(q) \mid u_p \xrightarrow[\mathcal{C}]{*} u_q$.
2. $\mathcal{C} \models p \equiv q =_{def} \mathcal{C} \models p \preceq q \wedge \mathcal{C} \models q \preceq p$.

Ces deux formules sont du premier ordre et se déduisent directement de la proposition 3.5.

◀

[8] une cylindrification (resp. projection) consiste à ajouter (resp. à supprimer) une composante dans un tuple.

La décidabilité de ces deux résultats a déjà été montrée (théorème 3.1 ; page 56). Par contre, comme les algorithmes canoniques de décision associés à ces formules sont PSPACE, on peut en déduire la complexité de l'implication dans le cas des contraintes bornées.

Pour l'instant, les formules utilisées sont du premier ordre. On obtient facilement, à partir de la théorie du second ordre, la décidabilité de nouveaux problèmes.

Définition 3.10 *Une requête régulière p a une borne finie (resp. a un équivalent fini) par rapport à un ensemble $\mathcal{C}$ de contraintes d'inclusions, s'il existe une requête f telle que $\mathcal{C} \models p \preceq f$ (resp. $\mathcal{C} \models p \equiv f$) et $L(f)$ est fini.*

Par exemple, considérons l'ensemble $\mathcal{C} = \{a^2 \preceq a\}$. Une borne finie de a^* est a alors que la requête ba^* n'a pas de borne finie.

Lemme 3.6 *Soient $\mathcal{C} = \{p_i \preceq u_i, 1 \leq i \leq n\}$ un ensemble de contraintes d'inclusions bornées et une requête régulière p.*

1. *Le problème de la borne finie pour $\mathcal{C}$ et p est décidable,*
2. *Le problème de l'équivalent fini pour $\mathcal{C}$ et p est décidable.*

Preuve : Chacune de ces propriétés peut être exprimée avec une formule (monadique) de la théorie de la réécriture préfixe.

1. $BorneFini(p) =_{def} \exists F \mid fini(F) \wedge \forall u_p \in L(p)\ \exists u_f \in F \mid u_p \xrightarrow[\mathcal{C}]{*} u_f$.
2. $EquivalentFini(p) =_{def} \exists F \mid fini(F) \wedge$
   $(\forall u_p \in L(p)\ \exists u_f \in F u_p \xrightarrow[\mathcal{C}]{*} u_f) \wedge (\forall u_f \in F\ \exists u_p \in L(p) u_f \xrightarrow[\mathcal{C}]{*} u_p)$.

Ces deux formules sont directement obtenues de la définition d'une requête qui a une borne finie. Elles sont du deuxième ordre. Il faut enfin remarquer que les algorithmes canoniques de décision associés à ces formules donnent un moyen effectif de calculer un tel F, quand la requête régulière p a une borne finie (a un équivalent fini). On peut enfin enrichir ces formules pour obtenir un F minimal au sens de l'inclusion. ◄

Le théorème suivant résume les résultats obtenus dans cette section en utilisant la théorie réécriture préfixe :

Théorème 3.2 *Soit $\mathcal{C} = \{p_i \preceq u_i, 1 \leq i \leq n\}$ un ensemble de contraintes d'inclusions bornées et deux requêtes régulières p et q.*

1. *Le problème de l'implication pour $\mathcal{C}$, p et q est décidable,*
2. *Le problème de l'équivalence pour $\mathcal{C}$, p et q est décidable,*
3. *Le problème de la borne finie pour $\mathcal{C}$ et p est décidable,*
4. *Le problème de l'équivalent fini pour $\mathcal{C}$ et p est décidable.*

La plupart des résultats obtenus sont déjà connus pour les données enracinées mais nous proposons un moyen uniforme de les prouver. Seule la décidabilité de l'équivalent fini est, à notre connaissance, nouvelle. En effet les points 1 et 2 sont prouvé dans [Abiteboul and Vianu, 1997] tandit que les auteurs de [André et al., 1999] ont montré le point 3.

3.4.2 Modèle de contraintes d'inclusions bornées

Dans la proposition 3.3 et dans le lemme 3.3, nous avons prouvé que tout ensemble de contraintes d'inclusions bornées a un modèle exact, mais le modèle que nous construisons est toujours infini.

Dans cette section, nous donnons une caractérisation des ensembles de contraintes d'inclusions bornées qui ont un modèle exact fini. Nous proposons alors un algorithme qui décide si un ensemble $\mathcal{C}$ a cette propriété.

Dans le but de caractériser les ensembles qui ont un modèle fini nous introduisons la relation d'équivalence suivante :

Définition 3.11 *Soit $\mathcal{C}$ un ensemble de contraintes d'inclusions. Nous appelons $\equiv_\mathcal{C}$ la relation sur $A^* \times A^*$ définie par $u \equiv_\mathcal{C} v$ si $\mathcal{C} \models u \equiv v$*

Il est évident que $\equiv_\mathcal{C}$ est une relation d'équivalence. Nous définissons alors la donnée $D_\mathcal{C} =< N, Racines, T >$ à partir des classes d'équivalence de $\equiv_\mathcal{C}$ qui sont notées $[u]_\mathcal{C}$:

- $N = \{[u]_\mathcal{C} \mid u \in A^*\}$
- $Racines = \{[u]_\mathcal{C} \mid \mathcal{C} \models u \preceq \varepsilon\}$
- $T = \{([u]_\mathcal{C}, x, [v]_\mathcal{C}) \mid \mathcal{C} \models v \preceq ux\}$

Remarquons que $D_\mathcal{C}$ est le quotient de $I_\mathcal{C}$ par la relation $\equiv_\mathcal{C}$. Le lemme suivant caractérise $D_\mathcal{C}(u)$ pour tout mot u :

Lemme 3.7 $\forall u \in A^*\ D_\mathcal{C}(u) = \{[v]_\mathcal{C} \mid \mathcal{C} \models v \preceq u\}$

Preuve : Raisonnons sur la longueur de u :
- *Si $u = \varepsilon$ alors $D_\mathcal{C}(\varepsilon) = \{[v]_\mathcal{C} \mid \mathcal{C} \models v \preceq \varepsilon\}$*
- *Si $u = vx$ alors $D_\mathcal{C}(u) = D_\mathcal{C}(D_\mathcal{C}(v), x)$. Si $[t]_\mathcal{C}$ appartient à $D_\mathcal{C}(u)$, il existe $[w]_\mathcal{C}$ dans $D_\mathcal{C}(v)$ et une transition $([w]_\mathcal{C}, x, [t]_\mathcal{C})$ dans $D_\mathcal{C}$. Par récurrence $\mathcal{C} \models w \preceq v$. Par construction de $D_\mathcal{C}$, $\mathcal{C} \models t \preceq wx$. On a donc $\mathcal{C} \models t \preceq wx$, $\mathcal{C} \models wx \preceq vx$ et $\mathcal{C} \models vx \preceq u$. Réciproquement si $\mathcal{C} \models t \preceq u$ alors la transition $([v]_\mathcal{C}, x, [t]_\mathcal{C})$ existe dans $D_\mathcal{C}$. Par récurrence $[v]_\mathcal{C}$ est accessible par v et $[t]_\mathcal{C}$ est accessible par u.*

◀

Nous pouvons maintenant prouver que :

Proposition 3.6 *Pour tout ensemble $\mathcal{C}$ de contraintes d'inclusions bornées, $D_\mathcal{C}$ est un modèle exact de $\mathcal{C}$.*

Preuve : Dans un premier temps, nous montrons que $D_\mathcal{C}$ est un modèle de $\mathcal{C}$: soient $(p \preceq u) \in \mathcal{C}$, v un mot de $L(p)$ et $[w]_\mathcal{C} \in D_\mathcal{C}(v)$. Il découle du lemme 3.7 que $\mathcal{C} \models w \preceq v$. Par transitivité, nous obtenons que $\mathcal{C} \models w \preceq u$ et, toujours d'après le lemme 3.7, on obtient que $[w]_\mathcal{C} \in D_\mathcal{C}(u)$ et donc $D_\mathcal{C} \models p \preceq u$.

Dans un deuxième temps il faut montrer que $D_\mathcal{C}$ est exact. Supposons que $D_\mathcal{C} \models p \preceq q$ pour deux requêtes régulières p et q. Soit u un mot de $L(p)$. Comme $[u]_\mathcal{C}$ appartient à $D_\mathcal{C}(u)$

et que $D_{\mathcal{C}} \models p \preceq q$, il existe un mot v de $L(q)$ tel que $[u]_{\mathcal{C}}$ appartient $D_{\mathcal{C}}(v)$. Il découle du lemme 3.7 que $\mathcal{C} \models u \preceq v$ et donc, pour conclure, $\mathcal{C} \models p \preceq q$. ◄

Il est évident que si $\equiv_{\mathcal{C}}$ est d'index fini alors, par construction, le modèle $D_{\mathcal{C}}$ est fini. Réciproquement, s'il existe un modèle exact fini de $\mathcal{C}$, alors $\equiv_{\mathcal{C}}$ est d'index fini. On a donc montré que :

Théorème 3.3 *Soit $\mathcal{C}$ un ensemble de contraintes d'inclusions bornées. $\mathcal{C}$ a un modèle exact fini si et seulement si $\equiv_{\mathcal{C}}$ est d'index fini.*

Corollaire 3.4 *Un ensemble de contraintes d'inclusions bornées a un modèle exact fini si et seulement si $D_{\mathcal{C}}$ est fini.*

La décidabilité du problème de l'index fini peut être établie en utilisant la décidabilité de la théorie du second ordre de la réécriture préfixe :

$$index_{fini} =_{def} \exists F fini(F) \wedge \forall u \; \exists u_f \in F \; u \xrightarrow[\mathcal{C}]{*} u_f \wedge u_f \xrightarrow[\mathcal{C}]{*} u$$

Remarque 3.4 *Remarquons que les deux propriétés suivantes ne sont pas équivalentes :*

1. *$\equiv_{\mathcal{C}}$ est d'index fini*
2. *A^* a un équivalent fini par rapport à $\mathcal{C}$*

En effet considérons l'alphabet $A = \{a\}$. $\{a^2 \preceq a\} \models a^* \equiv a + \varepsilon$ et pourtant a^n et a^m (n et m sont différents) n'appartiennent pas à la même classe et donc la relation n'est pas d'index fini.

L'ensemble $\{a^2 \preceq a\}$ est même très surprenant puisque toute requête régulière a un équivalent fini : si q est une requête régulière, u_q désigne le mot le plus court, qui n'est pas le mot vide, de $L(q)$. Il est alors facile de montrer que $\{a^2 \preceq a\} \models q \equiv \{u_q, \varepsilon\} \cap q$. Rappelons que cet ensemble de contraintes bornées n'a pas de modèle exact fini.

La propriété suivante étudie la complexité de cette caractérisation.

Proposition 3.7 *Soit $\mathcal{C}$ un ensemble de contraintes d'inclusions bornées. Décider si $\equiv_{\mathcal{C}}$ est d'index fini est EXPTIME dans la taille de $\mathcal{C}$.*

Preuve :

Pour étudier la complexité du problème on n'utilise pas la formule du deuxième ordre donnée ci-dessus mais on prouve un résultat plus général sur les relations reconnaissables. Rappelons en effet que si $\mathcal{C}$ est un ensemble de contraintes d'inclusions bornées alors $\xrightarrow[\mathcal{C}]{\star}$ est une relation reconnaissable. Nous pouvons décider si une relation d'équivalence reconnaissable est d'index fini.

En effet, soit r une relation d'équivalence reconnaissable. Nous allons construire une relation reconnaissable rep. $rep(u, v)$ signifie que v est le représentant de $[u]_r$. Pour représenter $[u]_r$, nous choisissons le mot le plus court (par longueur puis par ordre alphabétique) de la

classe $[u]_r$ car la relation plusCourt est évidemment une relation reconnaissable (on peut choisir un autre représentant associé à un autre ordre total reconnaissable). La relation rep, définie par $rep(u,v) = r(u,v) \wedge \forall w\ r(u,w) \Rightarrow plusCourt(v,w)$, est donc une relation reconnaissable et nous pouvons construire un automate pour la représenter. La relation monadique $\exists u \mid rep(u,v)$ est reconnaissable et peut effectivement être associée à un automate. Comme r est finie si et seulement si cette relation est finie, comme le test de finitude est décidable, on peut décider si r est d'index fini.[9]

D'après les résultats donnés dans la section 3.4.1.0 la relation equiv définie par $equiv(u,v) = (u \xrightarrow[\mathcal{C}]{\star} v) \wedge (v \xrightarrow[\mathcal{C}]{\star} u)$ est une relation reconnaissable et peut effectivement être associée avec un automate qui la représente. Il découle du théorème 3.3 que nous pouvons décider si $\mathcal{C}$ a un modèle exact.

Calculer $\{(u,v) \mid rep(u,v)\}$ en utilisant les algorithmes classiques de construction d'automate conduit à un algorithme exponentiel (le complémentaire de $equiv(u,v) \wedge plusCourt(u,v)$ doit être construit). Cette méthode permet aussi de construire un modèle exact puisque nous sommes capables de construire un automate pour l'ensemble des représentants de equiv et un automate pour $\{(u,v) \mid \mathcal{C} \models u \preceq v\}$. ◄

3.4.3 Le problème de la borne finie

Il est très difficile d'étudier la complexité d'un algorithme de décision associé à une formule du deuxième ordre. Dans cette section nous proposons donc un algorithme, dont nous pourrons évaluer la complexité, spécifique pour l'étude du problème de la borne finie (toujours avec des contraintes d'inclusions bornées). De plus, notre méthode permet de construire un transducteur qui construit, à partir de toute requête q, une borne de q. Cette borne est finie si et seulement si q a une borne finie. Enfin, notre algorithme propose un autre moyen de montrer que le problème de l'implication est PSPACE.

Le théorème suivant donne la dureté du problème de l'implication :

Théorème 3.4 *Soient $\mathcal{C} = \{p_1 \preceq u_1, \ldots, p_n \preceq u_n\}$ un ensemble fini de contraintes d'inclusions bornées, p et q deux requêtes régulières. Le problème de l'implication $\mathcal{C} \models p \preceq q$ est PSPACE-complet.*

Pour prouver ce théorème, définissons la notion d'ancêtre d'une requête par rapport au système de réécriture préfixe, puis montrons quelques lemmes concernant les ancêtres.

Soient $\mathcal{C} = \{p_i \preceq u_i, 1 \leq i \leq n\}$ un ensemble fini de contraintes d'inclusions bornées et q une requête régulière. On définit les ancêtres de q par $\text{ancêtre}_{\mathcal{C}}(q) = \{u \mid \exists\ w_q \in L(q),\ u \xrightarrow[\mathcal{C}]{*} w_q\}$.

On peut alors utiliser les ancêtres d'une requête pour résoudre le problème de l'implication :

[9] On peut remarquer que, dans ce cas, la relation est une union finie de produits cartésiens d'ensembles reconnaissables (i.e. une union finie d'ensembles de la forme $L \times L$ où L est reconnaissable). La relation est donc un sous-ensemble reconnaissable de $A^* \times A^*$ [Berstel, 1979, Comon et al., 1997].

Lemme 3.8 *Soient $\mathcal{C} = \{p_i \preceq u_i, 1 \leq i \leq n\}$ un ensemble fini de contraintes d'inclusions bornées, p et q deux requêtes régulières :*

$$\mathcal{C} \models p \preceq q \text{ si et seulement si } L(p) \subseteq \text{ancêtre}_{\mathcal{C}}(q)$$

Preuve : Ce lemme est une conséquence directe de la proposition 3.5 : $\mathcal{C} \models p \preceq q$ si et seulement si $\forall u_p \in L(p), \exists u_q \in L(q), u_p \xrightarrow[\mathcal{C}]{*} u_q$ i.e $L(p) \subseteq \text{ancêtre}_{\mathcal{C}}(q)$. ◄

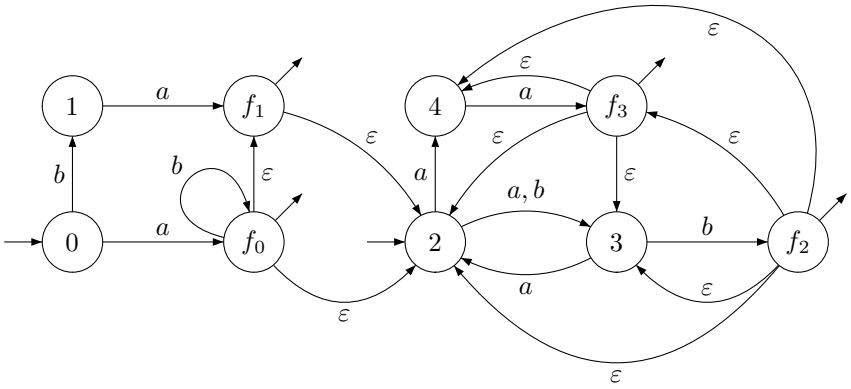
Il faut donc proposer un algorithme qui construise effectivement, à partir d'une requête régulière q et d'un ensemble de contraintes d'inclusions bornées $\mathcal{C}$, l'ensemble $\text{ancêtre}_{\mathcal{C}}(q)$. Dans un premier temps, on construit un automate $\mathcal{A}_{\mathcal{C}}$ (qui contient des ε-transitions) reconnaissant le langage $R_{\mathcal{C}} = \{v \in A^* \mid \exists i, v \xrightarrow[\mathcal{C}]{*} u_i\} = \text{ancêtre}_{\mathcal{C}}(u_1 + \ldots + u_n)$.

Il est déjà connu de [Büchi and Hosken, 1970], [Caucal, 1990] ou [Comon et al., 1997] que le langage $R_{\mathcal{C}}$ est un langage reconnaissable. Une nouvelle construction est proposée ici : pour tout i compris entre 1 et n on construit un automate $\mathcal{M}_i = (A, Q_i, I_i, F_i, \delta_i)$ qui reconnaît le langage $L(p_i + u_i)$. Sans restreindre la généralité du problème, on considère que deux ensembles Q_i et Q_j (i est différent de j) sont disjoints. On peut alors définir l'automate $\mathcal{A}_{\mathcal{C}} = (A, Q, I, F, \Delta)$ où $Q = \cup_{i=1}^{n} Q_i$, $I = \cup_{i=1}^{n} I_i$, $F = \cup_{i=1}^{n} F_i$ et $\Delta = \cup_{k \in \mathbb{N}} \Delta_k$. Δ_k est défini récursivement pour k dans $\mathbb{N}$ par

- $\Delta_0 = \cup_{i=1}^{n} \delta_i$
- si $k > 0$, $\Delta_k = \Delta_{k-1} \cup \{(q, \varepsilon, q') \mid q \neq q' \wedge \exists i \leq n, q \in F_i, q' \in {\mathcal{A}_{\mathcal{C}}}^{k-1}(u_i)\}$ où ${\mathcal{A}_{\mathcal{C}}}^{k-1}$ est l'automate $(A, Q, I, F, \Delta_{k-1})$

Comme Δ est inclus dans $Q \times (A \cup \{\varepsilon\}) \times Q$, il existe un entier K tel que $\Delta_K = \Delta_{K+1} = \Delta$. Comme K est plus petit que $|Q|^2$, l'automate $\mathcal{A}_{\mathcal{C}}$ peut être construit avec un algorithme polynomial dans la taille de $\mathcal{C}$.

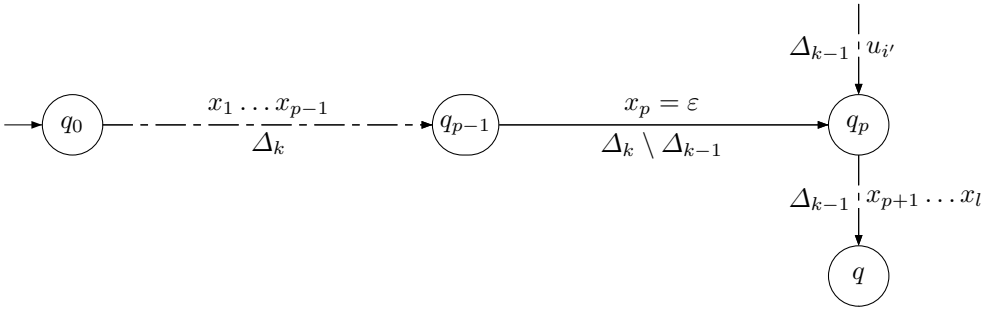
Exemple 3.4.3 : Soit $\mathcal{C} = \{ab^* \preceq ba, (aa + ba)^*(a + b)b \preceq aa\}$ un ensemble de contraintes d'inclusions bornées. L'automate $\mathcal{A}_{\mathcal{C}}$ est le suivant :



Il faut maintenant prouver que l'automate $\mathcal{A}_{\mathcal{C}}$ reconnaît $R_{\mathcal{C}}$.

Lemme 3.9 *Pour tout mot v dans A^*, pour tout entier i compris entre 1 et n si l'intersection entre $\mathcal{A}_{\mathcal{C}}(v)$ et F_i n'est pas vide alors $v \xrightarrow[\mathcal{C}]{*} u_i$.*

Preuve : Soient v un mot de A^* et q un état de $\mathcal{A}_\mathcal{C}$ tel qu'il existe i compris entre 1 et n pour lequel $q \in \mathcal{A}_\mathcal{C}(v) \cap F_i$. Il existe donc un entier k tel que $q \in {\mathcal{A}_\mathcal{C}}^k(v)$. On montre que $v \xrightarrow[\mathcal{C}]{*} u_i$ par récurrence sur k. Si $k = 0$, alors $v \in L(p_i + u_i)$ et $v \xrightarrow[\mathcal{C}]{} u_i$ ou $v = u_i$. Si $k > 0$, il existe $q_0, q_1, \ldots, q_l$, une suite d'états dans Q et $x_1, x_2, \ldots, x_l$, une suite d'éléments de l'ensemble $A \cup \{\varepsilon\}$ tels que $q = q_l$, q_0 appartienne à I, $v = x_1 x_2 \ldots x_l$ et pour tout j de $\{1, \ldots, l\}$, $(q_{j-1}, x_j, q_j) \in \Delta_k$. Soit m le nombre de transitions (q_{j-1}, x_j, q_j) qui appartiennent à $\Delta_k \setminus \Delta_{k-1}$. On raisonne maintenant par récurrence sur m. Si $m = 0$ on déduit des hypothèses faites sur k que $v \xrightarrow[\mathcal{C}]{*} u_i$. Si $m > 0$, soit p l'entier tel que $(q_{p-1}, x_p, q_p) \in \Delta_k \setminus \Delta_{k-1}$ et tel que pour tout j plus grand que p (q_{j-1}, x_j, q_j) appartient à Δ_{k-1}. Il existe alors un i' tel que $x_p = \varepsilon$, $q_{p-1} \in F_{i'}$ et $q_p \in {\mathcal{A}_\mathcal{C}}^{k-1}(u_{i'})$:



Par récurrence sur m, on obtient que $x_1 x_2 \ldots x_{p-1} \xrightarrow[\mathcal{C}]{*} u_{i'}$ et par induction sur k, on obtient $u_{i'} x_{p+1} \ldots x_l \xrightarrow[\mathcal{C}]{*} u_i$. On conclut par $v = x_1 x_2 \ldots x_l \xrightarrow[\mathcal{C}]{*} u_{i'} x_p x_{p+1} \ldots x_l \xrightarrow[\mathcal{C}]{*} u_i$. ◄

Il faut maintenant montrer la réciproque du lemme 3.9. On utilise pour cela le résultat suivant :

Lemme 3.10 *Soient v et w deux mots de A^*. Si $v \xrightarrow[\mathcal{C}]{*} w$, alors $\mathcal{A}_\mathcal{C}(w)$ est inclus dans $\mathcal{A}_\mathcal{C}(v)$.*

Preuve : Soit j la longueur de la dérivation $v \xrightarrow[\mathcal{C}]{*} w$. On prouve le lemme par récurrence sur j. Si $j = 0$ alors $v = w$ et $\mathcal{A}_\mathcal{C}(w) = \mathcal{A}_\mathcal{C}(v)$. Si $j > 0$, il existe alors un entier i dans $\{1, \ldots, n\}$ et deux mots v_1, v_2 tels que $v \xrightarrow[\mathcal{C}]{j-1} v_1 v_2$, $v_1 \in L(p_i)$ et $w = u_i v_2$. Par récurrence on a $\mathcal{A}_\mathcal{C}(v_1 v_2) \subseteq \mathcal{A}_\mathcal{C}(v)$. De plus, comme $v_1 \in L(p_i)$, il existe un état q dans $F_i \cap \mathcal{A}_\mathcal{C}(v_1)$. Soit q' un état de $\mathcal{A}_\mathcal{C}(u_i)$, on a alors $(q, \varepsilon, q') \in \Delta$ et $q' \in \mathcal{A}_\mathcal{C}(v_1)$. On déduit de $\mathcal{A}_\mathcal{C}(u_i) \subseteq \mathcal{A}_\mathcal{C}(v_1)$ que $\mathcal{A}_\mathcal{C}(w) = \mathcal{A}_\mathcal{C}(u_i v_2) \subseteq \mathcal{A}_\mathcal{C}(v_1 v_2) \subseteq \mathcal{A}_\mathcal{C}(v)$. ◄

On peut maintenant prouver que :

Proposition 3.8 *Pour tout mot v de A^*, pour tout i compris entre 1 et n $\mathcal{A}_\mathcal{C}(v) \cap F_i \neq \emptyset$ si et seulement si $v \xrightarrow[\mathcal{C}]{*} u_i$.*

Preuve : D'après le lemme 3.9, seule la condition nécessaire doit être prouvée. Considérons un mot $v \in A^*$ tel qu'il existe i et $v \xrightarrow[\mathcal{C}]{*} u_i$. Par définition $\mathcal{A}_\mathcal{C}(u_i) \cap F_i \neq \emptyset$, et d'après le lemme 3.10, $\mathcal{A}_\mathcal{C}(u_i) \subseteq \mathcal{A}_\mathcal{C}(v)$ et donc $\mathcal{A}_\mathcal{C}(v) \cap F_i \neq \emptyset$ ◄

Corollaire 3.5 *Soit $\mathcal{C}$ un ensemble de contraintes d'inclusions bornées, $\mathcal{A}_\mathcal{C}$ l'automate défini ci-dessus et $R_\mathcal{C}$ le langage régulier $\{v \in A^* \mid \exists i, v \xrightarrow[\mathcal{C}]{*} u_i\}$.*
L'automate $\mathcal{A}_\mathcal{C}$ reconnaît le langage $R_\mathcal{C}$

Pour utiliser le lemme 3.8 il faut pouvoir répondre à la question : p est-il inclus dans ancêtre$_\mathcal{C}(q)$?

Il est prouvé, dans le corollaire précédent, que $\mathcal{A}_\mathcal{C}$ reconnaît $R_\mathcal{C}$ et il est facile de construire, à partir de l'automate $\mathcal{A}_\mathcal{C}$ et d'un mot u_i, un automate qui reconnaît ancêtre$_\mathcal{C}(u_i)$. Nous n'avons en effet qu'à considérer l'automate ${\mathcal{A}_\mathcal{C}}^{u_i} = (A, Q, I, F_i, \Delta)$. De plus cette construction est polynomiale dans la taille de $\mathcal{C}$.

Pour y répondre, on représente les ancêtres de q par l'automate $\mathcal{A}^{\$_q}_{\mathcal{C}\cup\{q\preceq\$_q\}}$ ($\$_q$ est une nouvelle lettre). Dans [Aho et al., 1974] A. Aho, J. Hopcroft et J. Ullman donnent un algorithme de décision pour l'inclusion de deux langages représentés par leur automate. A partir de ce résultat on prouve que :

Lemme 3.11 *Pour tout ensemble $\mathcal{C} = \{p_i \preceq u_i, 1 \leq i \leq n\}$ de contraintes d'inclusions bornées, et pour toute requête régulière p et q, le problème de l'implication $\mathcal{C} \models p \preceq q$ est PSPACE.*

Preuve : L'algorithme proposé dans [Aho et al., 1974] pour tester l'inclusion entre deux langages est PSPACE. On peut de plus construire, en temps polynomial, un automate A_p qui reconnaît p [Gluskov, 1961] et on peut construire, en temps polynomial dans la taille de $|q| + |\mathcal{C}|$, un automate qui reconnaît ancêtre$_\mathcal{C}(q)$. ◄

Ce résultat n'est pas nouveau puisqu'il a déjà été montré dans cette thèse (lemme 3.5 ; page 65). Par contre, cette nouvelle preuve basée sur la notion d'ancêtre va nous permettre de calculer une borne finie d'une requête régulière q (quand la borne existe).

On est désormais capable de terminer la preuve du théorème 3.4 en étudiant la dureté du problème de l'implication. Pour cela, on montre que même si la requête régulière est réduite à un mot, le problème de l'implication $\mathcal{C} \models p \preceq u$ est PSPACE-complet.

Lemme 3.12 *Pour tout ensemble $\mathcal{C} = \{p_1 \preceq u_1, \ldots, p_n \preceq u_n\}$ de contraintes d'inclusions bornées, pour toute requête régulière p et pour tout mot u, le problème de l'implication $\mathcal{C} \models p \preceq u$ est PSPACE-dur.*

Preuve : Le problème de l'inclusion entre deux langages, représentés par leur expression est PSPACE-dur [Garey and Johnson, 1978]. Si on considère un ensemble de contraintes réduit à $\mathcal{C} = \{q \preceq \$\}$ ($\$$ n'apparaît pas dans q), ancêtre$_\mathcal{C}(\$) = L(q)$ *et $L(p) \subseteq L(q)$ est équivalent à $L(p) \subseteq$* ancêtre$_\mathcal{C}(\$)$. *Ce qui veut dire que décider si $L(p) \subseteq L(q)$ est équivalent à décider si $\mathcal{C} \models p \preceq \$$.* ◄

Néanmoins, pour le problème de l'implication d'une contrainte de mots $u \preceq q$, on a un algorithme polynomial : on doit simplement tester si u appartient à ancêtre$_\mathcal{C}(q)$.

Proposition 3.9 *Soient $\mathcal{C} = \{p_1 \preceq u_1, \ldots, p_n \preceq u_n\}$ un ensemble de contraintes d'inclusions bornées, u un mot et q une requête régulière. On peut décider du problème de l'implication $\mathcal{C} \models u \preceq q$ en PTIME.*

Preuve : $\mathcal{C} \models u \preceq q$ *si et seulement si* $u \in$ ancêtre$_{\mathcal{C}}(q)$. *On construit un automate* $\mathcal{A}_{\mathcal{C}}^{\$_q}$ *reconnaissant* ancêtre$_{\mathcal{C}}(q)$ *en PTIME et on teste l'appartenance de* u *à* ancêtre$_{\mathcal{C}}(q)$ *grâce à cet automate.* ◄

Le tableau suivant récapitule tous les résultats de cette section concernant le problème de l'implication (p et q sont deux requêtes régulières et u est un mot) :

problème	contraintes d'inclusions bornées	contraintes d'inclusions
donnée	multi-racines	enracinée
$\mathcal{C} \models p \preceq q$?	PSPACE (lemme 3.11)	EXPSPACE [Abiteboul and Vianu, 1997] EXPTIME [Alechina et al., 2003]
$\mathcal{C} \models p \preceq u$?	PSPACE-complet (lemme3.12)	
$\mathcal{C} \models u \preceq q$?	PTIME (proposition 3.9)	

Tous les éléments définis pour étudier le problème de l'implication permettent aussi d'étudier le problème de la borne finie. On va même au delà, car on va construire un transducteur $\tau_{\mathcal{C}}$ à partir de l'automate $\mathcal{A}_{\mathcal{C}}$ qui vérifie : pour toute requête régulière p, si p a une borne finie par rapport à $\mathcal{C}$, alors $C \models p \preceq \tau_{\mathcal{C}}(L(p))$ et $\tau_{\mathcal{C}}(L(p))$ est fini. Autrement dit, on peut décider si une requête a une borne finie, mais on peut aussi construire cette borne.

Définition 3.12 *Soient* $\mathcal{C}$ *un ensemble de contraintes d'inclusions bornées et* v *un mot de* $A^\star$. *Le plus petit suffixe de* v *qui ne peut pas être réécrit par le système* $\xrightarrow[\mathcal{C}]{\star}$, *noté* $f_{\mathcal{C}}(v)$, *est le grand suffixe de* v *qui satisfait pour tout mot* v_1 *de* $R_{\mathcal{C}} \cup \{\varepsilon\}$ *et pour tout mot* v_2 *de* $A^\star$, *si* $v = v_1 v_2$ *alors* $|v_2| \geq |f_{\mathcal{C}}(v)|$.

Le lemme suivant découle directement de la définition

Lemme 3.13 *Soient* $\mathcal{C} = \{p_1 \preceq u_1, \ldots, p_n \preceq u_n\}$ *un ensemble de contraintes d'inclusions bornées,* v *et* w *deux mots de* $A^\star$. *Alors* $v \xrightarrow[\mathcal{C}]{\star} w$ *si et seulement s'il existe* v_1 *et* w_1 *dans* $A^\star$ *tels que* $v = v_1 f_{\mathcal{C}}(v)$, $w = w_1 f_{\mathcal{C}}(v)$ *et* $v_1 \xrightarrow[\mathcal{C}]{\star} w_1$.

Preuve : La condition est clairement suffisante. Réciproquement, raisonnons par récurrence sur la longueur de la dérivation. Considérons la dérivation $v_1 f_{\mathcal{C}}(v) \xrightarrow[\mathcal{C}]{*} w$. *Si cette longueur est nulle la propriété est vraie. Sinon il existe un mot* v_1' *tel que*

$$v_1 f_{\mathcal{C}}(v) \xrightarrow[\mathcal{C}]{n} v_1' f_{\mathcal{C}}(v) \xrightarrow[\mathcal{C}]{1} w$$

On a donc $v_1' f_{\mathcal{C}}(v)$ *qui est de la forme* $w_1 w_2$ *avec* w_1 *un mot de* $R_{\mathcal{C}}$. *Par définition de* $f_{\mathcal{C}}(v)$, $|w_2| \geq |f_{\mathcal{C}}(v)|$. *Il existe donc un mot* w_2' *et un mot* u_i *tels que* $w = u_i w_2' f_{\mathcal{C}}(v)$ *et* $v_1 \xrightarrow[\mathcal{C}]{*} u_i w_2'$. ◄

Le plus petit suffixe d'un mot qui ne peut pas être réécrit par rapport à un système $\xrightarrow[\mathcal{C}]{\star}$ a déjà été utilisé pour tester si une requête a une borne finie.

Théorème 3.5 ([André et al., 1999]) *Soient* $\mathcal{C} = \{p_1 \preceq u_1, \ldots, p_n \preceq u_n\}$ *un ensemble de contraintes d'inclusions bornées et* p *une requête régulière.* p *a une borne finie par rapport à* $\mathcal{C}$ *si et seulement si l'ensemble* $F_{\mathcal{C}}(p) = \{f_{\mathcal{C}}(v) \mid v \in L(p)\}$ *est fini.*

Les auteurs de [André et al., 1999] étudient les contraintes d'inclusions dans des données enracinées. Comme le suggère la proposition 3.2, leur preuve s'étend directement au cadre de cette thèse.

Preuve : Pour tout mot w de $L(p)$, il existe un mot w' appartenant à $F = \{u_1, \ldots u_n, \varepsilon\}.F_{\mathcal{C}}(p)$ tel que $w \xrightarrow[\mathcal{C}]{*} w'$. Donc, si $F_{\mathcal{C}}(P)$ est fini, p admet F comme borne finie.

Réciproquement, supposons que p admette un certain f comme borne finie et posons $n = \max_{v \in L(f)} (|v|)$. Soit w un mot de $L(p)$, il existe alors $v \in L(f)$ tel que $w \xrightarrow[\mathcal{C}]{*} v$ et d'après le lemme 3.13 on a $|f_{\mathcal{C}}(w)| \leq n$ c'est-à-dire que $f_{\mathcal{C}}(p)$ est fini. ◄

Utilisons ce résultat pour construire un transducteur $\tau_{\mathcal{C}}$ qui ait la propriété suivante :

Pour tout mot v de $A^\star$, le mot w appartient à $\tau_{\mathcal{C}}(v)$ si et seulement si $w = v = f_{\mathcal{C}}(v)$ ou $v \xrightarrow[\mathcal{C}]{+} w \wedge w = u_i f_{\mathcal{C}}(v)$ pour un entier i de l'ensemble $\{1, \ldots, n\}$.

Soit $\mathcal{A}'_{\mathcal{C}}$ l'automate déterministe complet obtenu à partir de $\mathcal{A}_{\mathcal{C}}$ en appliquant les algorithmes classiques (suppression des transitions ε, déterminisation avec la construction des sous-ensembles, complétion). On associe alors à tout état q de l'automate $\mathcal{A}'_{\mathcal{C}} = (A, Q', q'_0, F', \Delta')$ l'ensemble $S(q) \subseteq \{1, \ldots, n\}$ tel que pour tout mot v, i appartienne à $S(\Delta'(q'_0, v))$ si et seulement si $v \xrightarrow[\mathcal{C}]{+} u_i$. En particulier, $F' = \{q \in Q' \mid S(q) \neq \emptyset\}$. Remarquons enfin que l'automate $\mathcal{A}'_{\mathcal{C}}$ peut être construit en $O(2^{|C|})$.

Le transducteur $\tau_{\mathcal{C}}$ est alors défini par : $\tau_{\mathcal{C}} = (A, Q' \cup \bar{Q}', \{q_0, \bar{q}_0\}, \bar{Q}', T)$ où A est l'alphabet de lecture et d'écriture, $Q = Q' \cup \bar{Q}'$ ($\bar{Q}'$ est une copie de Q') est l'ensemble des états, q_0 et $\bar{q}_0$ sont les états initiaux, $\bar{Q}'$ est l'ensemble des états finaux et $T \subseteq Q \times A \cup \{\varepsilon\} \times A^\star \times Q$ est l'ensemble des transitions de $\tau_{\mathcal{C}}$. T est défini par :

$$T = \{(q, x, \varepsilon, q') \mid q, q' \in Q', x \in A, (q, x, q') \in \Delta'\}$$
$$\cup \{q, \varepsilon, u_i, \bar{q}) \mid q \in Q', i \in S(q)\}$$
$$\cup \{(\bar{q}, x, x, \bar{q}') \mid q, q' \in Q', x \in A, S(q') = \emptyset, (q, x, q') \in \Delta'\}.$$

Proposition 3.10 *Pour tout mot v, w dans $A^\star$, le mot w appartient à $\tau_{\mathcal{C}}(v)$ si et seulement si $w = v = f_{\mathcal{C}}(v)$ ou $v \xrightarrow[\mathcal{C}]{+} w$ et $w = u_i f_{\mathcal{C}}(v)$ pour un i de l'ensemble $\{1, \ldots, n\}$.*

Preuve : Les chemins du transducteur $\tau_{\mathcal{C}}$ satisfont la propriété suivante : pour tout couple d'états $(q, \bar{q}')$ il n'existe aucun chemin qui permette de rejoindre q depuis $\bar{q}'$.

- *Soient $v, w \in A^\star$ tels que $w \in \tau_{\mathcal{C}}(v)$ il existe alors un chemin dans $\tau_{\mathcal{C}}$ d'un état initial vers un état final étiqueté par v. Si ce chemin part de l'état $\bar{q}_0$, alors $w = v$. De plus, comme pour tout $(\bar{q}, x, x, \bar{q}')$ dans T $S(q') = \emptyset$, il n'existe pas de préfixe de v, différent du mot vide, qui appartienne à $R_{\mathcal{C}}$. On en déduit que $f_{\mathcal{C}}(v) = v = w$. Supposons maintenant que le chemin étiqueté par v parte de l'état q_0. Il existe alors deux mots v_1 et v_2 tels que $v = v_1 v_2$, v_1 étiquette un chemin de q_0 vers un état q de Q'. Ce chemin vérifie : il existe un i dans $S(q)$ tel que $(q, \varepsilon, u_i, \bar{q})$ appartienne à T et tel que v_2 étiquette un chemin de $\bar{q}$ vers un état $\bar{q}'$ de $\bar{Q}'$. Dans ce cas, $v_1 \xrightarrow[\mathcal{C}]{+} u_i$, $w = u_i v_2$ et $v \xrightarrow[\mathcal{C}]{+} w$.*

Notons que pour tout préfixe v'_2 de v_2, différent du mot vide, nous sommes sûrs que $v_1 v'_2$ n'appartient pas à $R_{\mathcal{C}}$ car v'_2 étiquette un chemin de $\bar{q}$ vers un état $\bar{q''}$ pour lequel $S(q'') = \emptyset$. On en déduit que $v_2 = f_{\mathcal{C}}(v)$ et donc $w = u_i f_{\mathcal{C}}(v)$.

- *Réciproquement, s'il existe v et $w \in A^\star$ tels que $w = v = f_{\mathcal{C}}(v)$, il n'existe alors aucun préfixe de v, différent du mot vide, appartenant à $R_{\mathcal{C}}$. v étiquette donc un chemin de $\bar{q_0}$ vers un état $\bar{q'}$ de $\bar{Q'}$ et $v \in \tau_{\mathcal{C}}(v)$. Supposons maintenant que $v \xrightarrow[\mathcal{C}]{+} w$ et $w = u_i f_{\mathcal{C}}(v)$ pour un i de l'ensemble $\{1, \ldots, n\}$. D'après le lemme 3.13, $v = v_1 f_{\mathcal{C}}(v)$ et $v_1 \xrightarrow[\mathcal{C}]{+} u_i$. v_1 étiquette donc un chemin de q_0 vers un état q tel que $i \in S(q)$. De plus, $f_{\mathcal{C}}(v)$ étiquette un chemin de q vers un état de $\bar{q'}$ de $\bar{Q'}$. On a donc montré que $u_i f_{\mathcal{C}}(v) = w$ appartient à $\tau_{\mathcal{C}}(v)$.*

◀

Proposition 3.11 *Soient $\mathcal{C} = \{p_1 \preceq u_1, p_2 \preceq u_2, \ldots, p_n \preceq u_n\}$ un ensemble fini non vide de contraintes d'inclusions bornées et $\tau_{\mathcal{C}}$ le transducteur défini précédemment. Pour toute requête régulière p sur l'alphabet A :*

1. *$\mathcal{C} \models p \preceq q$ pour toute requête q telle que $\tau_{\mathcal{C}}(L(p))$ est inclus dans $L(q)$*
2. *p a une borne finie par rapport à $\mathcal{C}$ si et seulement si $\tau_{\mathcal{C}}(L(p))$ est fini.*

Preuve :

1. *Soient u_p un mot de $L(p)$ et v_q un mot de $\tau_{\mathcal{C}}(u_p)$ ($\tau_{\mathcal{C}}(u_p)$ n'est jamais vide). On déduit de la proposition 3.10 que soit $u_p \xrightarrow[\mathcal{C}]{+} v_q$ soit $u_p = v_q$ et donc $\mathcal{C} \models u_p \preceq v_q$. Comme v_q appartient à $L(q)$, la proposition 3.5 permet de conclure que $\mathcal{C} \models p \preceq q$.*
2. *Si $\tau_{\mathcal{C}}(L(p))$ est fini alors p a une borne finie par rapport à $\mathcal{C}$. Réciproquement, si p a une borne finie par rapport à $\mathcal{C}$ alors, d'après le théorème 3.5, $F_{\mathcal{C}}(p) = \{f_{\mathcal{C}}(w), w \in L(p)\}$ est un ensemble fini. On déduit de la proposition 3.10 que $\tau_{\mathcal{C}}(L(p)) \subseteq \{\varepsilon, u_1, ..., u_n\} F_{\mathcal{C}}(p)$ et donc $\tau_{\mathcal{C}}(L(p))$ est fini.*

◀

Remarque 3.5 *Nous proposons un algorithme qui, à partir d'un ensemble non vide de contraintes d'inclusions bornées $\mathcal{C}$ et d'une requête régulière q, construit une requête finie f telle que $\mathcal{C} \models p \preceq f$ si et seulement si p a une borne finie. Cet algorithme est $O(|p|^3.2^{|C|})$ puisqu'il calcule $\tau_{\mathcal{C}}(L(p))$ en faisant le produit cartésien entre un automate reconnaissant $L(p)$ et le transducteur $\tau_{\mathcal{C}}$ (dont la taille est $O(2^{|C|})$, suite à la déterminisation de $\mathcal{A}_{\mathcal{C}}$). Néanmoins notre algorithme est PSPACE.*

Montrons que le problème de la borne finie est PSPACE-dur : soient p et q deux expressions régulières sur un alphabet A. Considérons une nouvelle lettre \$ qui n'appartient pas à A. Définissons un ensemble de contraintes d'inclusions bornées $\mathcal{C}$ par : $\mathcal{C} = \{q\$^+ \preceq \$\}$. On peut maintenant montrer que la requête régulière $p\$^+$ a une borne finie par rapport à $\mathcal{C}$ si et seulement si $L(p)$ est inclus dans $L(q)$.

En effet si $L(p)$ est inclus dans $L(q)$, alors $\mathcal{C} \models p \preceq q$ et donc $\mathcal{C} \models p\$^+ \preceq \$$. Réciproquement si $p\$^+$ a une borne finie alors pour tout mot u_p de $L(p)$ pour tout entier n strictement positif

il existe une dérivation $u_p\$^n \xrightarrow[\mathcal{C}]{1} w$. *Comme* \$ *n'est pas une lettre de l'alphabet* A, u_p *est un mot de* $L(q)$.

On peut donc coder dans le problème de la borne finie le problème qui consiste à décider si deux expressions régulières sont incluses l'une dans l'autre. Il est connu [Aho et al., 1974] que ce problème est PSPACE-dur et donc que notre problème l'est aussi.

3.5 Les contraintes d'équivalences entre mots

Dans cette section, nous considérons le cas des égalités de mots de la forme $u \equiv v$ où u et v des mots. Ces contraintes étant un cas particulier des contraintes d'inclusions bornées, tous les algorithmes présentés dans la section 3.4 peuvent être utilisés. Mais, dans le cas des égalités de mots, l'implication devient symétrique (i.e. $\mathcal{C} \models u \preceq v$ implique que $\mathcal{C} \models v \preceq u$), il est possible d'améliorer certains de ces algorithmes. En particulier, il est possible de tester en temps polynomial, si un ensemble de contraintes d'égalité a un équivalent fini.

3.5.1 Un modèle exact

Dans la section 3.4, nous avons introduit $\equiv_{\mathcal{C}}$, une relation d'équivalence sur les contraintes d'inclusion bornées. Cette relation était définie pour deux mots u et v par $u \equiv_{\mathcal{C}} v$ si $\mathcal{C} \models u \equiv v$. Désormais, comme $\mathcal{C}$ est un ensemble d'égalités de mots (qui est une relation symétrique sur A^*), la relation $\equiv_{\mathcal{C}}$ satisfait la propriété suivante :

Lemme 3.14 *Soit* $\mathcal{C}$ *un ensemble d'égalités de mots.* $\equiv_{\mathcal{C}}$ *est la plus petite relation d'équivalence, close par congruence droite qui contient* $\mathcal{C}$.

Preuve : $\mathcal{C}$ *étant un ensemble de contraintes d'égalités de mots,* $\equiv_{\mathcal{C}}$ *contient* $\mathcal{C}$. *Si* (u, v) *appartient à* $\equiv_{\mathcal{C}}$ *alors* $u \xrightarrow[\mathcal{C}]{*} v$ *et* $v \xrightarrow[\mathcal{C}]{*} u$. *Par congruence droite et pour tout label* x, $ux \xrightarrow[\mathcal{C}]{*} vx$ *et* $vx \xrightarrow[\mathcal{C}]{*} ux$ *c'est à dire que* (ux, vx) *appartient à* $\equiv_{\mathcal{C}}$ *et donc la relation* $\equiv_{\mathcal{C}}$ *est close par congruence droite.*

On en déduit que si le couple (u, v) *appartient à* $\equiv_{\mathcal{C}}$ *alors il appartient à toute relation d'équivalence, close par congruence droite, qui contient* $\mathcal{C}$. ◄

Dans le cas particulier des égalités de mots, le modèle exact $D_{\mathcal{C}}$ associé à un ensemble $\mathcal{C}$ de contraintes bornées (introduit dans la section 3.4.2) est déterministe et complet. En effet on peut le définir par :

- $N = \{[u]_{\mathcal{C}} \mid u \in A^*\}$,
- $r = \{[\varepsilon]_{\mathcal{C}}\}$
- $T = \{([u]_{\mathcal{C}}, x, [ux]_{\mathcal{C}}) \mid u \in A^*, x \in A\}$.

Pour tout mot u de A^*, le lemme 3.7 nous garantit que $D_{\mathcal{C}}(u) = \{[u]_{\mathcal{C}}\}$. On en déduit la proposition suivante :

Proposition 3.12 *Pour tout ensemble* $\mathcal{C}$ *d'égalités de mots sur un alphabet* A, *les points suivants sont équivalents :*

1. $\mathcal{C} \models u \equiv v$.
2. $\mathcal{C} \models u \equiv v$ *sur la famille des documents enracinés.*
3. $\mathcal{C} \models u \equiv v$ *sur la famille des documents déterministes.*
4. $\mathcal{C} \models u \equiv v$ *sur la famille des documents déterministes et complets.*

Preuve : Il est évident qu'il suffit de montrer que 4 implique 1. Soient u et v deux chemins tels que $\mathcal{C} \models u \equiv v$ sur la famille des documents déterministes et complets. On a $D_\mathcal{C} \models u \equiv v$ car $D_\mathcal{C} \models \mathcal{C}$ et $D_\mathcal{C}$ est déterministe et complet. Il découle de la proposition 3.6, qui prouve que $D_\mathcal{C}$ est un modèle exact de $\mathcal{C}$, que $\mathcal{C} \models u \equiv v$. ◄

Généralement, le modèle $D_\mathcal{C}$ est un graphe infini. Néanmoins, quand $\mathcal{C}$ est un ensemble fini d'égalités de mots, il est possible de construire un sous-graphe fini de $D_\mathcal{C}$ tel que les informations présentes dans ce sous graphe soient suffisantes pour décider du problème de l'implication, de l'existence d'une requête équivalente finie ou de l'existence d'un modèle fini.

Une construction similaire est proposée par P. Buneman W. Fan et S. Weinstein dans [Buneman et al., 1999] ou dans [Buneman et al., 2000b]. Les auteurs de ces articles ont en effet étudié l'impact du déterminisme sur les contraintes d'inclusion.

Définition 3.13 *Soit $\mathcal{C}$ un ensemble de contraintes d'égalités de mots sur un alphabet A.*
- *Considérons W l'ensemble de tous les préfixes des mots de $\mathcal{C}$ i.e. les préfixes de $\{w \in A^* \mid \exists w' \in A^*, (w \equiv w') \in \mathcal{C}\}$.*
- *Pour tout mot w de W, $[w]$ est la classe d'équivalence de w*
- *Nous définissons $D_\mathcal{C}^f$, un graphe fini et déterministe, par $D_\mathcal{C}^f =< N', r', T' >$ où*
  - $N' = \{[w] \mid w \in W\}$,
  - $r' = \{[\varepsilon]\}$ *et*
  - $T' = \{([w], x, [wx]) \mid w \in W, wx \in W, x \in A\}$.

Considérons maintenant l'application $f_\mathcal{C}$, définie de A^* sur $N' \times A^*$ par : pour tout mot de A^* , $f_\mathcal{C}(u) = (D_\mathcal{C}^f(u_1), u_2)$ où $u = u_1u_2$ et u_1 est le plus long préfixe de u tel que $D_\mathcal{C}^f(u_1)$ n'est pas l'ensemble vide.

Exemple 3.5.1 : Soient $A = \{a, b, c, d, e, f\}$ un alphabet de six lettres et $\mathcal{C} = \{a \equiv bba$, $b \equiv c, cb \equiv dd, d \equiv e, fa \equiv aa, ed \equiv f, e \equiv f, aa \equiv bba\}$ un ensemble de contraintes d'égalités de mots. La figure 3.9 donne le graphe $D_\mathcal{C}^f$ pour cet ensemble de contraintes. Par exemple, $f_\mathcal{C}(a^3) = ([bba], \varepsilon)$, $f_\mathcal{C}(a^3c) = ([bba], c)$.

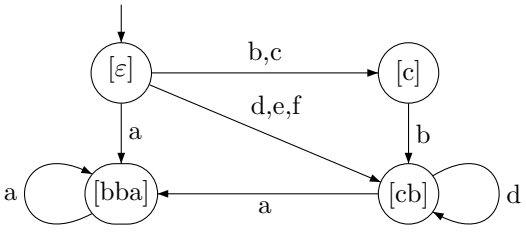


FIG. 3.9 – graphe $D_\mathcal{C}^f$

La fonction f_c est très utile pour étudier le problème de l'implication. En effet, $C \models u \equiv v$ si et seulement si $D_c(u) = D_c(v)$ si et seulement si $f_c(u) = f_c(v)$. On a donc montré que :

Proposition 3.13 *$C \models u \equiv v$ si et seulement si $f_c(u) = f_c(v)$.*

Si p est une requête régulière, $f_c(p)$ est l'ensemble $\bigcup_{u \in L(p)} f_c(u)$. On peut alors déduire de la proposition précédente, du lemme 3.14 et du fait que D_c est complet et déterministe que :

Corollaire 3.6 *Pour tout couple de requêtes régulières p et q, $\mathcal{C} \models p \equiv q$ si et seulement si $f_c(p) = f_c(q)$.*

On montre également que :

Corollaire 3.7 *Une requête régulière p a une borne finie par rapport à un ensemble fini de contraintes d'égalités de mots $\mathcal{C}$ si et seulement si $f_c(p)$ est fini.*

Preuve : Si une requête q a un équivalent fini par rapport à $\mathcal{C}$ il existe alors une requête q_f telle que $\mathcal{C} \models q \equiv q_f$ et $L(q_f)$ est fini. D'après le corollaire 3.6, $f_c(q) = f_c(q_f)$. Comme $L(q_f)$ est fini, $f_c(q_f)$ est une union finie de $f_c(u)$ (u appartient à $L(q_f)$) et il est donc fini.

Soit p une requête telle que $f_c(p)$ est fini. Soit $([v], v')$ un élément de $f_c(p)$ et notons $rep_{([v],v')}$ un mot de l'ensemble $\{\ u \in L(p) \mid f_c(u) = ([v], v')\ \}$. On considère alors le langage fini $q_f = \bigcup_{([v],v') \in f_c(p)} rep_{([v],v')}$. Par construction $f_c(q) = f_c(q_f)$ et donc $\mathcal{C} \models q \equiv q_f$. ◄

Concernant l'existence d'un modèle exact pour un ensemble $\mathcal{C}$ de contraintes d'égalités de mots, on peut énoncer :

Proposition 3.14 *Soit $\mathcal{C}$ un ensemble de contraintes d'égalités de mots. $\mathcal{C}$ a un modèle exact fini si et seulement si D_c^f est un graphe complet.*

Preuve : Si D_c^f est complet alors $\equiv_c$ est d'index fini et donc $\mathcal{C}$ a un modèle exact fini (théorème 3.3).

Réciproquement, si D_c^f n'est pas complet il existe un mot u tel que $D_c^f(u)$ est vide. Ce qui signifie que $f_c(u) = (D_c^f(u_1), u_2)$ avec u_2 qui n'est pas le mot vide. On a montré que pour tout label x et pour tout entier n, $f_c(ux^n) = (D_c^f(u_1), u_2x^n)$. Si ux^n et ux^m sont différents (i.e. n et m sont différents) alors $f_c(ux^n)$ est différent de $f_c(ux^m)$. Il découle de la proposition 3.13 que $\mathcal{C} \not\models ux^n \equiv ux^m$. Ces deux mots ne sont pas dans la même classe et donc $\equiv_c$ n'est pas d'index fini. ◄

Corollaire 3.8 *Pour tout ensemble $\mathcal{C}$ de contraintes d'égalités de mots, D_c est l'unique document déterministe (complet) qui satisfait : $D \models u \equiv v$ si et seulement $\mathcal{C} \models u \equiv v$.*

Corollaire 3.9 *Si $\equiv_c$ est d'index fini alors le nombre de classes est borné par la taille de $\mathcal{C}$.*

Dans le but de décider efficacement de ces différentes propriétés, il faut proposer un algorithme efficace qui calcule le graphe D_c^f. C'est le but de la prochaine section.

3.5.2 Modèle de contraintes de mots

La section précédente a mis en évidence l'importance du graphe $D_{\mathcal{C}}^f$ pour l'étude des contraintes de mots. Pour étudier la complexité des différents algorithmes de décision présentés, il faut être capable d'élaborer un algorithme efficace pour sa construction. Le but de cette section est donc de présenter un algorithme qui peut construire $D_{\mathcal{C}}^f$, en un temps quasi-linéaire dans la taille de l'ensemble de contraintes d'égalités de mots $\mathcal{C} = \{u_i \equiv v_i, 1 \leq i \leq n\}$.

Construire le graphe $D_{\mathcal{C}}^f$ consiste à mettre, pour toute contrainte $u \equiv v$ de $\mathcal{C}$, u et v dans la même classe d'équivalence et à itérer le processus par congruence droite. L'algorithme de table 3.1 propose une procédure **merge** qui a cette spécification.

procédure **merge**(in u,v : mots ; in out S :ensembleDeClasses)
% input : u et v deux mots tel que u ou v appartient à W, S un ensemble de classe d'équivalence
% output : l'ensemble S dans lequel on a fusionné les classes uw et vw (w est un mot)
% locales : x un label
retirer $[u]$ de S
retirer $[v]$ de S
ajouter $[u] \cup [v]$ dans S
pour tout $x \in A$ faire
  si ux ou vx appartient à W alors **merge**(ux,vx,S)
fin pour
fin

TAB. 3.1 – Calcul des classes de $\equiv_{\mathcal{C}}$

Pour implémenter efficacement la procédure **merge**, deux questions sont à étudier :

1. Comment gérer les classes d'équivalence afin de manipuler efficacement l'ensemble S ?
2. Comment représenter les mots de W afin de déduire de la classe d'un mot u la classe du mot ux ?

Nous utilisons le principe de l'*union-find* pour construire les classes de $\equiv_{\mathcal{C}}$ avec une idée similaire à celle de l'algorithme de Shostack [Shostak, 1978]. Le principe de l'*union-find* permet de calculer un ensemble de classes avec trois primitives : créer une nouvelle classe, calculer l'union de deux classes, et trouver (*find*) le représentant d'une classe. Dorénavant *find*(u) représente le représentant de la classe de u et *union* représente l'union de deux classes.

Pour répondre à la seconde question, on utilise une structure auxillaire G. G est un graphe orienté. Les nœuds de G sont les mots de W. Les arcs de G sont étiquetés par les labels de A. Notre but est de proposer un algorithme qui manipule G ayant l'invariant suivant : Soient u, v deux mots et x un label tels que ux et v appartiennent à W

Si v est le réprésentant de la classe $[u]$, alors il existe dans G un arc x de v vers le représentant de la classe $[ux]$.

Dans un premier temps, G représente l'arbre préfixe de W. Cette initialisation signifie que chaque nœud du graph (chaque mot de W) est une classe d'équivalence. On fusionne alors les nœuds équivalents comme le propose le principe de la table 3.1 et on obtient l'algorithme de la table 3.2. Cet algorithme utilise deux primitives dont nous donnons la spécification :

- **find**(u :nœud) :nœud ; retourne le représentant de la classe $[u]$
- **union**(in u :nœud ;in v :nœud ;in out S :ensembleDeClasses) ; retire $[u]$ et $[v]$ de S et crée une nouvelle classe $[u] \cup [v]$. Après un appel à **union**, **find**(u) est égale à **find**(v).

Nous ne donnons pas le code correspondant à ces deux fonctions. Néanmoins, il est possible de se référer à [Cormen et al., 1990] pour trouver toutes les structures de données et tous les algorithmes qui permettent d'implémenter **find** et **union** suivant le principe de l'*union-find*.

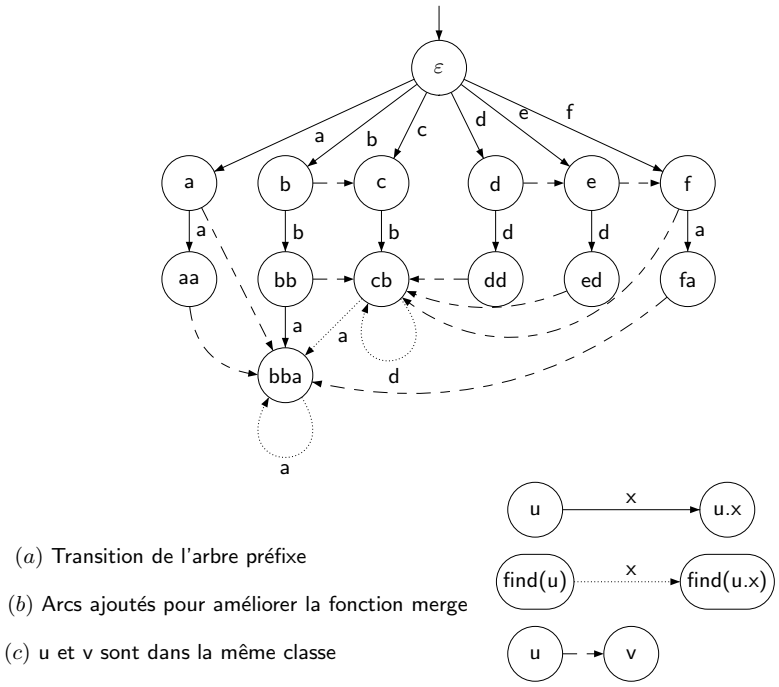
L'algorithme de calcul des classes d'équivalence devient alors (la procedure **merge** est donnée dans la table 3.2) :

pour toute les contraintes $(u \equiv v)$ de $\mathcal{C}$ faire
  merge(**find**(u),**find**(v))
fin pour

```
procédure merge(in out G :graphe ; in u,v : nœud de G ; in out S :ensembleDeClasses)
% input : G la structure auxiliaire, u et v deux nœuds de G, et
          S un ensemble de classes d'équivalence. u et v sont les représentants de leur classe.
% output : l'ensemble S dans lequel on a fusionné les classes uw et vw et G
% locales : x un label.
si u ≠ v alors
  union(u,v,S)
  pour tout x ∈ A faire
    si les transitions (u,x,w) et (v,x,w') existent dans G alors
      merge(w,w')
    fin si
    si la transition (u,x,w) existe dans G mais pas la transition (v,x,w') alors
      ajouter un nouvel arc, étiqueté par x, de v vers w dans G
    fin si
    si la transition (v,x,w') existe dans G mais pas la transition (u,x,w) alors
      ajouter un nouvel arc, étiqueté par x, de u vers w' dans G
    fin si
%     si ni (v,x,w') ni (u,x,w) n'existent dans G alors
%       pas de classes à fusionner
% fin si
  fin pour
fin si
fin
```

TAB. 3.2 – Calcul des classes de $\equiv_{\mathcal{C}}$ (algorithme quasi-linéaire)

Etudions la complexité de notre algorithme. Les auteurs de [Aho et al., 1974] ou R.E. Tarjan dans [Tarjan, 1975] ont montré qu'un algorithme utilisant $n-1$ unions et m *find*[10] est en $O(n + m.\alpha(n,m))$[11]. Dans le pire des cas, tous les nœuds appartiennent à la même classe d'équivalence. Dans ce cas, notre algorithme appelle exactement $|\text{nœuds}(G)| - 1$ fois la procédure **union** et $|\mathcal{C}|$ fois la fonction **find**[12]. Comme le nombre de nœud de G est égal à la taille de W, qui est elle même égale à la taille de $\mathcal{C}$, on déduit que notre algorithme, qui calcule les classes d'équivalence de $\equiv_{\mathcal{C}}$, est $O(|\mathcal{C}|.lg^*(|\mathcal{C}|))$. Il est de plus facile de montrer que le graphe $D_{\mathcal{C}}^f$ se définit par $< N, r, T >$ où $N = \{find(u) \mid u \in W\}$, $r = \{find(\varepsilon)\}$ et $T = \{(find(\mathrm{u}), x, find(ux)) \mid ux \in W\}$. On a donc prouvé la proposition suivante :



(*a*) Transition de l'arbre préfixe

(*b*) Arcs ajoutés pour améliorer la fonction merge

(*c*) u et v sont dans la même classe

FIG. 3.10 – Construction du graphe $D_{\mathcal{C}}^f$ avec l'algorithme de l'*union-find*

Proposition 3.15 *Le graphe $D_{\mathcal{C}}^f$ peut se construire avec un algorithme quasi-linéaire dans la taille de $\mathcal{C}$.*

Exemple 3.5.2 : (suite de l'exemple 3.5.1) Soit l'alphabet à six lettres $A = \{a, b, c, d, e, f\}$ et l'ensemble de contraintes $\mathcal{C} = \{a \equiv bba, b \equiv c, cb \equiv dd, d \equiv e, fa \equiv aa, ed \equiv f, e \equiv f, aa \equiv bba\}$. La figure 3.10 représente la structure de données G utilisée pour construire le graphe $D_{\mathcal{C}}^f$. On peut voir

– l'arbre préfixe (a) qui sert à initialiser G

[10] Chaque appel à **union** cache deux appels à **find**

[11] α la fonction inverse d'Ackermann. La fonction d'Ackermann A est définie par $A(1,j) = 2^j$ $j \geq 1$, $A(i,1) = A(i-1,2)$ si $i \geq 2$, $A(i,j) = A(i-1, A(i,j-1))$ si $i,j \geq 2$

[12] Dans le pire des cas, tous les mots de $\mathcal{C}$ sont de longueur 1.

- les nouveaux arcs (b) ajoutés par la procédure **merge**. Ils garantissent que pour tout mot u, si ux est un mot de W, alors (**find**(u),x,**find**(ux)) est une transition de G.
- Enfin les arcs qui simulent les classes d'équivalence (c) ne font pas partie de G. Ils permettent uniquement de suivre la progression de l'algorithme. Une implémentation naïve de **find** est de suivre tant que possible, à partir d'un nœud u, les arcs du type (c) et de retourner le dernier nœud atteint.

Comme ($b \equiv c$) appartient à $\mathcal{C}$, b et c sont dans la même classe. Il en découle que bb et cb sont dans la même classe. Comme $bba \in W$ et $cba \notin W$, l'algorithme ajoute un nouvel arc étiqueté par a entre la classe de cb et celle de bba. On déduit de $a \equiv bba \equiv cba \equiv dda \equiv eda \equiv fa$ que $G(a) = G(fa)$. De même comme $f \equiv ed \equiv fd$, on a $G(fd^*) = G(f)$. Finalement, après avoir fusionné les nœuds équivalents, on obtient le graphe $D_{\mathcal{C}}^f$ de la figure 3.9.

3.5.3 Le problème de l'implication

Dans cette section, on étudie comment la bonne complexité de l'algorithme qui construit le graphe $D_{\mathcal{C}}^f$ peut améliorer la complexité de quelques algorithmes pour le problème de l'implication, pour la recherche d'un équivalent fini ou pour l'existence d'un modèle fini exact.

Ce dernier problème peut être résolu facilement comme le montre cette propriété :

Proposition 3.16 *Soit $\mathcal{C}$ un ensemble de contraintes d'égalités de mots. Décider si $\mathcal{C}$ a un modèle exact fini peut se faire en temps quasi-linéaire dans la taille de $\mathcal{C}$.*

Preuve : Il est possible de construire le graphe $D_{\mathcal{C}}^f$ en temps quasi-linéaire. Décider si $D_{\mathcal{C}}^f$ est complet (proposition 3.14) peut se faire avec un algorithme qui est dans la taille de $D_{\mathcal{C}}^f$. ◀

En ce qui oncerne le problème de l'implication de contraintes, on peut améliorer l'algorithme de [Buneman et al., 1999] qui décide si un ensemble de contraintes d'inclusion (et d'inclusion inverse) de mots implique une égalité de mots. Malheureusement, il est aussi prouvé que le problème de l'implication entre deux requêtes régulières est toujours un problème PSPACE-dur. En ce qui conserne le problème de l'équivalent fini, il est possible de construire, en temps polynomial et si elle existe, à partir d'une requête q une requête finie q_f équivalente à q.

Pour tout mot u, si le graphe $D_{\mathcal{C}}^f$ est donné, le calcul de $f_{\mathcal{C}}(u)$ est linéaire dans la taille de u. Si on travaille avec une requête régulière on obtient :

Lemme 3.15 *Soit p et q deux requêtes régulières. Considérons que le graphe $D_{\mathcal{C}}^f$ est donné*

1. *Tester si $f_{\mathcal{C}}(p)$ est fini peut être fait en temps polynomial dans la somme de la taille de $D_{\mathcal{C}}^f$ et de la taille de p.*
2. *Tester si $f_{\mathcal{C}}(p)$ et $f_{\mathcal{C}}(q)$ sont égaux peut être fait avec un algorithme PSPACE dans la somme de la taille de $D_{\mathcal{C}}^f$, de la taille de p et de la taille de q.*

Preuve : Dans un premier temps, on prouve que, pour tout nœud n du graphe $D_{\mathcal{C}}^f$, il est possible de construire un automate $\mathcal{A}_{c,p}(n)$ qui reconnaît le langage $\{w \in A^* \mid (n, w) \in f_{\mathcal{C}}(p)\}$.

On prouve de plus que cette construction est PTIME dans la somme des tailles de D_c^f et de p. En effet, l'algorithme suivant calcule $\mathcal{A}_{c,p}(n)$ en cinq étapes :

1. *construire l'automate $\mathcal{A}_p$ qui reconnaît le langage décrit par la requête p. Tous les états de cet automate sont accessibles et co-accessibles. L'algorithme de Gluskov [Gluskov, 1961] permet une telle construction en $O(|p|^3)$.*
2. *compléter le graphe D_c^f avec un nœud puits $\perp$. On ajoute une transition $(n, x, \perp)$ pour tout nœud n et pour tout label x tel qu'il n'y ait pas, dans le graphe D_c^f, de transition étiquetée par x à partir du nœud n.*
3. *calculer le produit cartésien entre ce graphe complet et l'automate $\mathcal{A}_p$: dans ce nouveau graphe, les transitions sont de la forme $((n_1, s_1), x, (n_2, s_2))$ où n_1 et n_2 sont, soit des nœuds de D_c^f, soit le nœud $\perp$, s_1 et s_2 sont des états de l'automate $\mathcal{A}_p$ et x est un label.*
4. *enlever du graphe précédent toutes les transitions $((n_1, s_1), x, (n_2, s_2))$ dans lesquelles le nœud n_2 est un nœud de D_c^f (i.e. ce n'est pas le nœud $\perp$).*
5. *pour terminer, $\mathcal{A}_{c,p}(n)$ est obtenu à partir du graphe précédent : les états initiaux sont les nœuds de $\{n\} \times S$ où S est l'ensemble de nœuds de $\mathcal{A}_p$ et les états finaux sont les nœuds de $\{\perp\} \times F$ où F est l'ensemble de états finaux de $\mathcal{A}_p$.*

Toute cette construction peut se faire en PTIME dans la somme des tailles de $\mathcal{A}_p$ et de D_c^f.

Pour tester si $f_c(p)$ est fini, on teste si pour tout nœud n du graphe D_c^f le langage reconnu par l'automate est fini. Cet algorithme est PTIME dans la somme de la taille de D_c^f et de la taille de $\mathcal{A}_p$.

Pour terminer, on peut comparer $f_c(p)$ et $f_c(q)$ pour deux requêtes régulières p et q. Il suffit de comparer pour tous les nœuds n du graphe D_c^f le langage reconnu par l'automate $\mathcal{A}_{c,p}(n)$ et celui reconnu par $\mathcal{A}_{c,q}(n)$. On obtient alors un algorithme PSPACE dans la somme de la taille de $\mathcal{A}_{c,p}(n)$ et de la taille de $\mathcal{A}_{c,q}(n)$. ◄

Remarque 3.6 *On ne peut pas améliorer la complexité de l'algorithme qui compare $f_c(p)$ et $f_c(q)$. En effet, si on considère un ensemble vide de contraintes d'égalités de mots, $\mathcal{C} \models p \equiv q$ si et seulement si le langage décrit par p est égal au langage décrit par q. Il est connu de [Garey and Johnson, 1978] que ce problème est PSPACE-dur dans la somme des tailles des deux expressions régulières p et q. On en déduit que le problème de l'implication par un ensemble de contraintes d'égalités de mots est PSPACE-dur.*

Lemme 3.16 *Soient $\mathcal{C}$ un ensemble de contraintes d'égalité de mots, p une requête et u un mot. On peut décider en PTIME si $\mathcal{C} \models p \equiv u$.*

Preuve : Rappelons que $\mathcal{C} \models p \equiv v$ si et seulement si $f_c(u) = f_c(p)$. u étant un mot, $f_c(u)$ se réduit à l'ensemble $\{([u_1], u_2)\}$ avec $u = u_1 u_2$ et u_1 est le plus grand préfixe de u tel que $D_c^f(u_1)$ n'est pas vide. Pour décider si $f_c(p) = f_c(u)$ il faut donc :

- *Tester si l'ensemble $\{[u] \mid \exists v \in L(p), w \in A^* f_c(v) = ([u], w)\}$ se réduit au singleton $\{[u_1]\}$. En calculant le produit carthésien entre un automate reconnaissant $L(p)$ et D_c^f, ce test peut être éffectué en PTIME.*

– *Tester si l'automate $\mathcal{A}_{\mathcal{C},p}([u_1])$ (défini dans la preuve du lemme 3.15) reconnait le mot u_2. Cette seconde étape est aussi PTIME.*

◄

Le théorème suivant reprend tous les résultats de complexité démontrés dans cette section (proposition 3.13, proposition 3.15, corollaire 3.6, corollaire 3.7, lemme 3.15 et lemme 3.16) :

Théorème 3.6 *Pour tout ensemble $\mathcal{C}$ de contraintes de mots :*
- *Décider si $\mathcal{C} \models u \equiv v$ (u et v sont des mots) est quasi linéaire dans la somme de la taille de $\mathcal{C}$ et de la taille de $u \equiv v$.*
- *Décider si $\mathcal{C} \models p \equiv u$ (u est un mot ; q est une requête) est PTIME.*
- *Décider si $\mathcal{C} \models p \equiv q$ (p et q sont des requêtes régulières) est PSPACE-complet dans la somme de la taille de $\mathcal{C}$ et de la taille de la contrainte $p \equiv q$.*
- *décider si une requête régulière a un équivalent fini par rapport à l'ensemble $\mathcal{C}$ est PTIME dans la somme de la taille de $\mathcal{C}$ et la taille de p.*

On sait d'après le théorème 3.6 qu'il est possible de décider si une requête régulière a un équivalent fini par rapport à un ensemble $\mathcal{C}$ de contraintes d'égalités de mots. Le problème qui est maintenant étudié est celui de construire effectivement une requête finie f équivalente à q. Si u est un chemin, on construit f à l'aide de la fonction $f_{\mathcal{C}}$: si $f_{\mathcal{C}}(u) = ([u_1], u_2)$, alors $\mathcal{C} \models u \equiv \mathit{find}(u_1).u_2$. On généralise cette idée aux expressions régulières pour obtenir :

Proposition 3.17 *Soit $\mathcal{C}$ un ensemble non vide de contraintes d'égalités de mots. On construit, avec un algorithme quasi-linéaire, un transducteur $\tau_{\mathcal{C}}$ tel que, pour toute requête régulière p sur A :*

1. *$\mathcal{C} \models p \equiv \tau_{\mathcal{C}}(L(p))$,*
2. *$\tau_{\mathcal{C}}(L(p))$ est fini si et seulement si p a un équivalent fini par rapport à $\mathcal{C}$.*

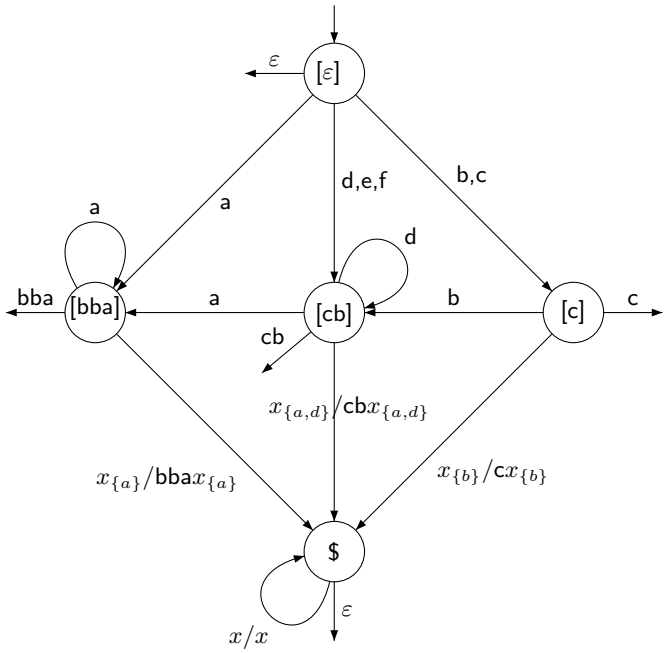
Preuve : Soit $\mathcal{C}$ un ensemble non vide de contraintes d'égalités de mots. Considérons le graphe $D^f_{\mathcal{C}} =< N, r, T_G >$ défini précédemment. Les noeuds de ce graphe sont les classes d'équivalence de la relation $\equiv_{\mathcal{C}}$. Pour toute classe $[u]$, $\mathit{find}(u)$ est le représentant de cette classe comme on l'a défini dans la section 3.5.2. On peut donc définir un transducteur $\tau_{\mathcal{C}} =< A, N \cup \{\$\}, r, N \cup \{\$\}, T, e >$ où A est l'alphabet de lecture et d'écriture, $N \cup \{\$\}$ ($\$$ n'appartient pas à N) est l'ensemble des états, r est l'état initial et tous les états sont finaux. L'ensemble T des transitions se définit par : $T = \{([u], x, \varepsilon, [ux]) \mid [ux] \in N\} \cup \{([u], x, \mathit{find}([u])x, \$) \mid [ux] \notin N\} \cup \{(\$, x, x, \$) \mid x \in A\}$. Enfin, e est la fonction d'écriture des états finaux vers A^* qui est définie par : $e([u]) = \mathit{find}([u])$ et $e(\$) = \varepsilon$. Il est facile de montrer que pour tout mot u, $f_{\mathcal{C}}(u) = ([u_1], u_2)$ si et seulement si $\tau_{\mathcal{C}}(u) = \mathit{find}([u_1])u_2$. Si q est une requête régulière, alors

$$\tau_{\mathcal{C}}(q) = \bigcup_{u \in L(q)} \tau_{\mathcal{C}}(u) = \bigcup_{\substack{u \in L(q) \\ f_{\mathcal{C}}(u) = ([u_1], u_2)}} \mathit{find}([u_1])u_2$$

Comme $f_{\mathcal{C}}(\tau_{\mathcal{C}}(q)) = f_{\mathcal{C}}(q)$, on déduit du corollaire 3.6 que pour tout q tel que $L(q) = \tau_{\mathcal{C}}(L(p))$, $\mathcal{C} \models p \equiv q$. Finalement, le corollaire 3.7 prouve que : $\tau_{\mathcal{C}}(L(p))$ est fini si et seulement si p a un équivalent borné par rapport à $\mathcal{C}$.

Remarquons que la construction du transducteur $\tau_{\mathcal{C}}$ est quasi-linéaire dans la taille de $\mathcal{C}$, puisqu'elle est basée sur la construction de $D^f_{\mathcal{C}}$. ◄

Exemple 3.5.3 : (suite de l'exemple 3.5.1) Le transducteur $\tau_{\mathcal{C}}$ est le suivant :



Pour simplifier la figure, les transitions de la forme $x \mid \varepsilon$ sont notées x. De plus, les transitions entre un noeud $[u]$ et le noeud \$ sont étiquetées par $x_s \mid ux_s$ avec $s \subseteq A$. Une telle transition correspond à l'ensemble de transitions $\{([u], x, \mathit{find}([u])x, \$) \mid x \notin s\}$. De même, l'ensemble de transitions $\{(\$, x, x, \$) \mid x \in A\}$ est représenté par $x \mid x$ (boucle sur le noeud \$).

- $\tau_{\mathcal{C}}(a^+b) = bbab$ car $f_{\mathcal{C}}(a^+b) = \{([bba], b)\}$. Comme $L(bbab)$ est fini, a^+b a un équivalent fini par rapport à $\mathcal{C}$. De plus $\mathcal{C} \models a^+b \equiv bbab$.
- $\tau_{\mathcal{C}}(f^+) = cbf^*$ car $f_{\mathcal{C}}(f^+) = \{([cb], f^n) \mid n \in \mathbb{N}\}$. Comme $L(cbf^*)$ n'est pas fini, f^+ n'a pas d'équivalent fini par rapport à $\mathcal{C}$. Néanmoins, $\mathcal{C} \models f^+ \equiv cbf^*$ est vrai.

Etant donné un transducteur $\tau_{\mathcal{C}}$ et une requête réguliére p, l'algorithme suivant calcule un automate reconnaissant le langage $\tau_{\mathcal{C}}(L(p))$:

1. Calculer le produit carthésien entre un automate reconnaissant $L(p)$ et l'automate A_τ construit à partir de $\tau'_{\mathcal{C}}$. $\tau'_{\mathcal{C}}$ est construit en ajoutant à $\tau_{\mathcal{C}}$ un nouvel état $\perp$ et l'ensemble de transitions $\{(n_\tau, \varepsilon \mid e(n_\tau), \perp) \mid n_\tau \in \mathit{etats}(\tau_{\mathcal{C}})\}$. A_τ a tous les états de $\tau'_{\mathcal{C}}$. Les états initiaux sont ceux de $\tau_{\mathcal{C}}$. L'état $\perp$ est final. Les transitions de A_τ sont la forme (n_τ, x, n'_τ) avec $(n_\tau, x \mid u, n'_\tau)$ une transition de $\tau'_{\mathcal{C}}$.
2. Remplacer toutes les transitions $((n_q, n_\tau), x, (n'_q, n'_\tau))$ du produit par les transitions $((n_q, n_\tau), u, (n'_q, n'_\tau))$ telles que $(n_\tau, x \mid u, n'_\tau)$ soit une transition de $\tau'_{\mathcal{C}}$.

3. Remplacer toutes les transitions étiquetées par u par un automate déterministe reconnaissant le mot u.

Toute cette construction est PTIME et on peut donc généraliser l'un des résultats du théorème 3.6 : il est possible de décider si une requête p a une borne finie en PTIME mais on peut aussi construire, si elle existe, une borne finie avec la même complexité.

3.6 Optimisation de requêtes : Qric

Pour valider ces résultats théoriques avec un point de vue *base de données*, nous avons développé QRIC (Query Rewriting with Inclusion Constraints [Debarbieux and Luchier, 2004]), un ensemble d'outils qui manipulent les données semi-structurées modélisées par des graphes.

Notre objectif est de répondre, expérimentalement, à la question suivante : étant donné une donnée D, un ensemble de contraintes $\mathcal{C}$ (satisfaites par D) et une requête q, est-il plus rapide de calculer $D(q)$ ou d'optimiser q (en cherchant une requête finie équivalente à q par rapport à l'ensemble $\mathcal{C}$) puis d'évaluer cette nouvelle requête sur D ?

Au vu des résultats théoriques présentés dans les sections précédentes, nous avons étudié une seule classe de contraintes : les égalités de mots. Plus précisément, nous avons implémenté l'algorithme qui construit le transducteur décrit dans la proposition 3.17, et nous avons comparé le temps nécessaire pour calculer $D(q)$ (ce temps est noté $t(q)$) et le temps nécessaire pour calculer $D(\tau_{\mathcal{C}}(q))$ (ce temps est noté $t(q_{opt})$).

Pour effectuer ces mesures, deux problèmes sont à résoudre : le premier problème consiste à trouver des documents (et des requêtes) pour effectuer nos tests, le second problème consiste à trouver des contraintes d'égalités de mots satisfaites par un document. Ce second problème sera résolu dans le chapitre consacré à l'indexation et plus particulièrement dans la section *dataguide*.

Nous avons trouvé trois sources de documents :
- Des benchmarks utilisés par la communauté internationale [Schmidt et al., 2001],
- Des documents XML construits à partir de la base IMDB.
- Des documents XML générés aléatoirement [Tox, 2002]

La table 3.1 présente trois cas caractéristiques :

Le premier document considéré a été construit à partir d'un ensemble de documents de la base IMDB (figure 4.1, page 91). A partir d'un ensemble de films, on construit un anneau entre les films (grâce aux transitions *suivant*) et on identifie dans quel(s) film(s) a joué chaque acteur.

Pour certains films, le champ auteur n'est pas spécifié et la donnée générée ci-dessus est fortement irrégulière. On a donc identifié les films ayant des acteurs et créé des documents plus réguliers (qui, par exemple, satisfont la contrainte `films.film.acteur.joueDans`$\equiv$`films.film`).

Les documents du troisième exemple sont issus du *XMark project* dont le but est de proposer un *benchmark* de documents XML (avec des références) pour permettre a chaque concepteur d'outils d'évaluer son travail. Les membres de *XMark project* ont aussi pour objectif de proposer des documents qui ressemblent à ceux du monde *réel*.

	$t(q) > t(q_{opt})$	$t(q) = t(q_{opt})$	$t(q) < t(q_{opt})$
Films (125k)	5	5	3
Films réguliers (125k - 1 482 arcs)	8	2	3
Benchmark (11M - 167 535 arcs)	1	21	3

TAB. 3.1 – Optimisation de requêtes

Quelles conclusions peut-on tirer de ces expériences ?

Comme le montrent les deux premières lignes du tableau, l'algorithme proposé pour optimiser une requête régulière est efficace et il permet un gain de temps lors de l'évaluation de nombreuses requêtes.

Il faut néanmoins nuancer ces résultats : en effet, pour qu'une majorité de requêtes soit optimisée, il faut beaucoup de contraintes d'équivalences (comparaison entre les deux premières lignes). Or les données semi-structurées sont irrégulières et contiennent peu de contraintes de ce type (à moins que la donnée ait été construite à partir d'une base relationnelle). La dernière ligne suggère de plus qu'il y a peu d'optimisations possibles dans des documents *réels*.

Comme le montre la dernière colonne du tableau 3.1, il est possible que, dans certains cas, l'utilisation de la requête réécrite entraîne une perte de temps par rapport à la requête d'origine. Une solution pour éviter ce problème (et qui améliorerait tous les résultats) pourrait être de modifier l'algorithme qui choisit le représentant dans une classe de $\equiv$ (on modifie alors le transducteur $\tau_{\mathcal{C}}$ et donc la réécriture). On choisit pour l'instant le mot le plus court et le premier dans l'ordre alphabétique, alors que l'on pourrait baser ce choix sur des caractéristiques du document. Les auteurs de [McHugh et al., 1998], ou de [McHugh and Widom, 1999] proposent, dans le cadre du projet Lore, des statistiques sur les chemins présents dans une donnée, qui pourraient guider ce choix.

3.7 Conclusion

Ce chapitre était consacré aux contraintes d'inclusions dans les données semi-structurées modélisées par des graphes étiquetés à plusieurs racines. Les problèmes abordés étaient : l'existence d'un modèle exact fini, le problème de l'implication de contraintes et le problème des équivalents finis (bornes finies).

Dans le cas le plus général, aucun outil n'est connu pour manipuler les contraintes et nous travaillons directement avec les graphes comme le font S. Abiteboul et V. Vianu dans [Abiteboul and Vianu, 1997].

Au contraire, dans le cas des contraintes d'inclusions bornées, les problèmes considérés se transforment en formule de la théorie de la réécriture préfixe. Les problèmes de l'implication et de l'équivalence s'expriment avec formules du premièr ordre alors que les problèmes de la borne finie, de l'équivalent fini et de l'index fini se définissent avec des formules du deuxième ordre. On a donc proposé une méthode commune pour étudier tous les problèmes.

Néanmoins, quelques algorithmes *ad hoc* sont présentés pour étudier plus finement les problèmes de complexité. Ces algorithmes utilisent la notion d'ancêtre dans le système de

réécriture. Ils ont permis de montrer que le problème de l'implication ou que le problème de la borne finie sont PSPACE-complet.

Dans le cas très particulier des contraintes d'égalités de mots, on construit (en temps quasi linéaire) un graphe fini qui contient toute l'information utile pour manipuler les contraintes d'inclusions. On en déduit, un algorithme quasi-linéaire qui décide si un ensemble de contraintes d'égalités de mots a un modèle fini, un algorithme quasi-linéaire pour le problème de l'implication entre deux mots et un algorithme PTIME qui construit, à partir d'une requête q et d'un ensemble de contraintes, une requête finie équivalente (si cette requête existe). Malheureusement le problème de l'implication entre deux requêtes reste PSPACE-complet. Pour valider ces résultats du point de vue de l'optimisation de requêtes, QRIC, un ensemble d'outils manipulant des données semi-structurées a été développé en Ocaml.

Trois pistes sont intéressantes à suivre dans le futur :

- La PSPACE-complétude du problème de l'implication est établie dans le cadre des contraintes d'inclusions bornées. Rapellons que, dans le cadre général, ce problème est un problème ouvert très intéressant d'un point du vue théorique. De même, une requête réguliére est représentée par son arbre syntaxique. Que deviennent les résultats de complexité si les requêtes sont codées par des DAGs (i.e. les sous structures communes sont partagées).
- La première consiste à optimiser des requêtes Xpath en utilisant des contraintes d'inclusions. L'idée naturelle est d'utiliser un fragment des requêtes graphes : on pourrait ainsi généraliser les travaux de [Amer-Yahia et al., 2001] ou de [Vagena et al., 2004] sur les twigs en considérant uniquement les axes *child*, *descandant* et *idref*.
- La seconde consiste à voir la notion de clefs dans les documents XML comme des contraintes d'inclusions (le type des contraintes dépendant du type de clefs). Dans ce cas, le problème est de savoir si on peut résoudre des problèmes d'implication entre clés avec les techniques que nous avons utilisées pour les contraintes d'inclusions[Fan and Simeon, 2000, Bouchou et al., 2003, Abrão et al., 2004].

Chapitre 4

Requêtes Régulières : indexation de chemins

4.1 Introduction

Les données semi-structurées sont modélisées par des graphes orientés étiquetés à plusieurs racines. Dans ce modèle, plusieurs informations sont donc explicites. On peut par exemple citer, la structure de graphe, l'alphabet des labels, les chemins... Rappelons que, et comme c'est le cas pour toute cette thèse, une donnée ne se conforme à aucun schéma prédéfini (ni DTD, ni XML schéma, ni grammaire de graphe...).

Si on reprend la figure 1.2 (page 4), une information implicite est que certains nœuds se ressemblent. Par exemple, les nœuds 1, 2, 3 et 5 ont tous un arc entrant étiqueté *auteur*. Notre étude peut encore être particularisée : en effet certains de ces nœuds ont un arc *co-auteur* entrant et d'autres n'en ont pas.

L'idée sous-jacente à l'exemple précédent est une notion de type. On souhaite typer les nœuds de la donnée afin d'avoir une (des) information(s) supplémentaire(s) sur le contenu de la donnée. Un type est un ensemble de nœuds d'une donnée qui se ressemblent. Typer les nœuds d'une donnée consiste à construire une relation *types* entre les nœuds de la donnée et un ensemble de types. On ne restreint pas la relation *types* c'est-à-dire qu'un nœud peut avoir plusieurs types et que plusieurs nœuds peuvent appartenir à un même type. Autrement dit, si $types(t)$ désigne l'ensemble de tous les nœuds d'une donnée qui ont le type t, un nœud n peut appartenir à deux ensembles $types(t)$ et $types(t')$ différents. Le choix des types et le mode de construction de la relation *types* dépendent de l'utilisation du typage et de l'information qu'il apporte aux utilisateurs.

La notion de type est très souvent associée à la notion de *guide* [Nestorov et al., 1997, Buneman et al., 1997, Nestorov et al., 1998, Abiteboul et al., 2000]. Le guide d'une donnée semi-structurée D est un graphe (comme D) dont les nœuds sont les types de D. Les transitions présentes dans le guide servent à modéliser les chemins de la donnée. Une contrainte très souvent imposée à un guide est d'avoir un langage de chemins qui soit un sur-ensemble du langage de chemins de la donnée. Cette contrainte permet d'avoir la même notion de requêtes sur les deux représentations.

Donnons un exemple de guide. Supposons que l'on veuille construire un guide qui représente les contraintes d'inclusions présentes dans une donnée. Le MAM (Miroir de l'Afer canonique du langage Miroir) [Caron et al., 2003] est un bon candidat. En effet, MAM_D est un graphe construit à partir de l'ensemble L_D de tous les chemins d'une donnée D et qui vérifie :

$$MAM_D \models p \preceq q \text{ si et seulement si } \exists \text{ une donnée } D' \mid L_{D'} = L_D \wedge D' \models p \preceq q$$

MAM_D vérifie donc des contraintes d'inclusions qui peuvent ne pas être vérifiées par D mais qui pourraient l'être et ce, sans modifier l'ensemble des chemins de la donnée. Dans ce cas, le typage a la sémantique suivante : deux nœuds n et n' ont le même type si pour toute requête q, $D(\{n\}, q)$ est égal à $D(\{n'\}, q)$.

Explicitons les principales fonctions d'un guide sur les nœuds d'une donnée :
- *Optimiser l'évaluation des requêtes :* connaître la structure d'une donnée permet de guider le processeur de requêtes et de travailler seulement dans une (petite) partie de la donnée.
- *Décrire, de façon à aider un utilisateur à formuler des requêtes, le contenu d'une donnée.*
- *Faciliter l'intégration de plusieurs sources de données :* si on connaît la structure de plusieurs données différentes, on peut chercher à trouver une structure commune et|ou à définir une conversion d'un type à l'autre.
- *Améliorer le stockage :* Au lieu de stocker naïvement la donnée, on peut regrouper les nœuds qui ont le même type. Ce regroupement permet de diminuer le nombre de pages chargées en mémoire et donc d'améliorer la performance du système.
- *Interdire certaines mises à jour :* C'est une utilisation classique des types dans le modèle relationnel qui est beaucoup moins courante dans notre modèle.

Indexer une donnée semi-structurée consiste à construire un guide couvrant c'est-à-dire un guide qui satisfasse une condition supplémentaire, non vérifiée par le MAM, sur les requêtes. Soit D une donnée et I_D un guide de D, on dit que I_D est un guide couvrant (un index) de D si pour toute requête régulière q :

$$D(q) = \bigcup_{t \in I_D(q)} \{n \in D \mid n \in types(t)\}$$

Autrement dit, on peut évaluer le résultat de toute requête régulière en utilisant uniquement l'index, sans jamais consulter la donnée d'origine.

Au vu des définitions précédentes, une donnée D est un index couvrant d'elle-même (la relation *types* est alors l'identité). Cette indexation n'apporte rien du point de vue de l'évaluation de requêtes. On définit donc trois critères pour évaluer les index.

1. Simplifier l'évaluation des requêtes. On souhaite que le temps nécessaire pour répondre à une requête sur l'index soit inférieur au temps nécessaire pour répondre à la même requête sur la donnée. Ce critère est souvent remplacé par une comparaison de **taille** : on souhaite avoir une taille de l'index qui soit plus petite que celle de la donnée initiale.
2. Outre le problème de la taille de l'index on souhaite aussi avoir un **algorithme efficace** pour calculer l'index. En effet un index de très petite taille peut être construit avec un algorithme très coûteux en temps.
3. La **préservation des contraintes d'inclusions**. Dans le chapitre précédent, on a mis en évidence l'utilité des contraintes d'inclusions dans l'évaluation des requêtes. On cherche donc à construire des index qui ont une structure *semblable* à celle de la donnée.

L'idée est, comme pour les requêtes, de pouvoir utiliser la structure de l'index sans consulter celle de la donnée.

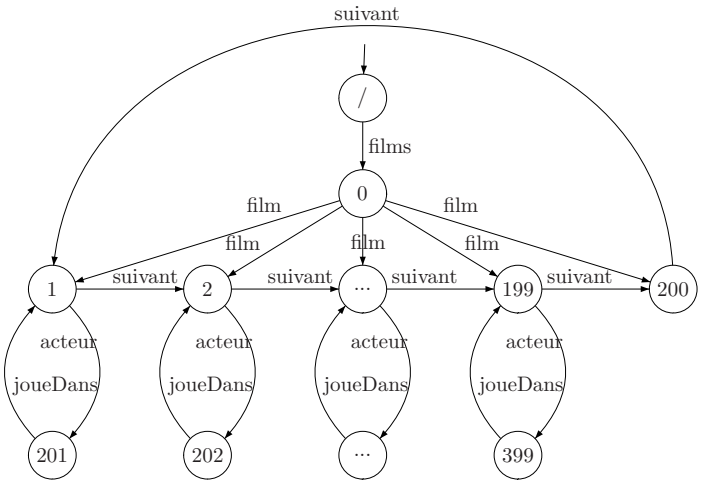


FIG. 4.1 – Exemple d'une donnée construite à partir de IMDB

Une autre propriété importante d'un index est d'avoir un algorithme de mise à jour par rapport à une modification d'une donnée qui ne recalcule pas tout l'index. Comme nous allons le voir, toutes les méthodes que nous présentons ont cette propriété.

Dans la littérature, les points 1 et 2 sont les plus étudiés. Historiquement, le dataguide [Goldman and Widom, 1997] fut, dans le cadre du projet Lore, le premier index de chemins proposé. Comme son nom l'indique, outre son rôle d'indexation, le data**guide** a un rôle de guide : guider l'utilisateur lors de son écriture de requêtes (le dataguide étant déterministe, il est facile d'assister l'écriture de requêtes en utilisant le dataguide comme un plan -*site map*- de la donnée). Dans le but de contrôler la taille du guide d'une donnée, S. Nestorov, S. Abiteboul et R. Motwani proposent de typer une donnée avec exactement K types [Nestorov et al., 1998]. Même si leur algorithme est basé sur des algorithmes de clustering qui ne sont pas des méthodes d'indexation (pas de notion de couverture), ils définissent le typage *perfect* qui peut se transformer en l'index *perfect*. Enfin, le 1-index [Milo and Suciu, 1999] (et ses méthodes d'indexation dérivées, le A(k)-index [Kaushik et al., 2002b] et le D(k)-index [Chen et al., 2003]) a pour but d'être optimal quant à la place qu'il occupe, quant à son temps de construction et quant au gain effectué en l'interrogeant plutôt que la donnée.

Par contre, à ma connaissance, le lien entre les contraintes d'inclusions et les index est nouveau (point 3). Il est l'apport principal de ce chapitre. Un premier résultat est que, comme l'index est couvrant, toute contrainte satisfaite par un index d'une donnée D et aussi satisfaite par D. On étudie donc le problème inverse qui est de trouver dans l'index les mêmes contraintes que celles satisfaites par la donnée. Autrement dit, si I_D est un index d'une donnée D, a-t-on $D \models p \preceq q$ implique $I_D \models p \preceq q$?

Dans un premier temps, nous présentons le dataguide [Goldman and Widom, 1997] (section 4.2). Le dataguide regroupe des nœuds tels que chaque chemin de la donnée apparaisse exactement une fois dans l'index. On crée ainsi un index déterministe pour lequel l'évaluation

est efficace. Par contre, le dataguide ne conserve pas les contraintes d'inclusions. Néanmoins, la relation de sous-typage induite de la construction du dataguide contient toute l'information concernant les inclusions de mots. On en déduit un algorithme de saturation du dataguide qui permet de construire un index satisfaisant les contraintes de mots de la donnée. Enfin, le principe de construction du dataguide permet d'extraire d'une donnée toutes ses contraintes : on peut construire un ensemble fini $\mathcal{C}(D)$ de contraintes finies tel que la donnée D soit un modèle exact de l'ensemble $\mathcal{C}(D)$.

Dans un deuxième temps (section 4.3) nous présentons des méthodes qui regroupent dans l'index, des nœuds d'une donnée qui sont indifférenciables du point de vue des requêtes. L'idée est de fusionner des nœuds qui ont le même langage de chemins entrants. Tester si deux nœuds sont équivalents étant très coûteux (c'est un problème PSPACE-complet), la condition d'équivalence est affaiblie pour devenir un test de bi-simulation sur les nœuds. L'index perfect [Nestorov et al., 1998], le 1-index [Milo and Suciu, 1999] et ses méthodes d'indexation dérivées le A(k)-index [Kaushik et al., 2002b] et D(k)-index [Chen et al., 2003] sont basés sur ce principe. Tout ces index sont compacts, faciles à calculer et satisfont les mêmes contraintes d'inclusions que la donnée initiale.

La dernière section de ce chapitre est consacrée à l'étude des ensembles de données. En effet, interroger un ensemble E de données avec une requête q revient à calculer $E(q)$ l'union de tous les $D(q)$ pour D une donnée de E (on suppose, sans restreindre la généralité du problème, que deux données différentes n'ont pas de nœuds en commun). Du point vue des requêtes, on ne fait donc pas la différence entre l'ensemble de données E et la donnée à plusieurs racines $D_E =< N_E, R_E, T_E >$ construite comme suit : N_E est l'union de tous les nœuds des données présentes dans E, R_E est l'union de toutes les racines et T_E est l'union de toutes les transitions. Il faut noter que cette donnée satisfait une contrainte si et seulement si elle est satisfaite par toutes les données de l'ensemble E.

Pour résumer, étudier les contraintes communes aux données de l'ensemble E revient à chercher les contraintes satisfaites par la donnée D_E. On peut donc utiliser tous les résultats précédents pour étudier les contraintes d'inclusions communes à plusieurs données. Néanmoins, la troisième section porte sur une différence : supposons que l'on ait un ensemble de données E et un algorithme d'indexation *index* et que quelqu'un veuille interroger non pas toutes les données de E mais uniquement un sous ensemble E' de E. Deux méthodes d'indexation sont possibles :

- Indexer chaque donnée séparément puis construire U-index(E) comme l'union de tous ces index.
- Indexer l'ensemble E en le considérant comme une unique donnée et construire index(E).

La troisième section est donc consacrée à la comparaison entre U-index(E) et index(E). Plus précisément nous prenons comme référence le 1-index et nous étudions expérimentalement le rapport entre le nombre de données présentes dans E et le nombre de données présentes dans E' et les forces|faiblesses du U-index(E) par rapport au 1-index(E).

Pour comparer les différents algorithmes d'indexation nous avons pris une donnée de référence qui nous sert de *fil rouge* tout au long de ce chapitre. Cette donnée contient 200 films de l'année 1966 et a été construite à partir de la base de données MovieDB (aussi appelée IMDB pour *The international movie database*). La figure 4.1 montre une (petite) sous-partie de cette donnée. Il manque en effet toutes les informations concernant les producteurs, les titres, les résumés...De plus, il y a plusieurs acteurs par film et un acteur peut jouer dans plusieurs

films. Cette partie permet de comprendre les mécanismes d'indexation. On remarque que le film correspondant au nœud 200 n'a pas d'acteur. Il s'agit de *Wolnamjeonseon isangeobtda*, un documentaire de guerre tourné en Corée du sud et pour lequel il n'y pas d'acteurs. On verra les conséquences de ce manque de régularité pour chacun des algorithmes.

4.2 Dataguide

La construction d'un dataguide ([Goldman and Widom, 1997], [Abiteboul et al., 2000]) a été la première méthode d'indexation de données semi-structurées. Le besoin d'une telle construction est apparu dans le cadre du projet Lore [Lore, 1997].

Le projet Lore a été développé à l'université de *Stanford* et il a pour objectif de gérer des données semi-structurées modélisées par des graphes. Le langage de requêtes utilisé est Lorel. C'est un langage basé sur les requêtes régulières mais qui permet en plus d'interroger le contenu des nœuds du graphe.

Il y a beaucoup d'alternatives possibles pour développer une base de données semi-structurées. Il est possible de partir d'un système déjà existant sur le modèle relationnel ou sur le modèle objet... Les concepteurs de Lore ont décidé de construire un système à "partir de rien" et de proposer, pour chaque problème, des solutions propres aux données semi-structurées.

Il est évident que l'un des problèmes majeurs à résoudre est l'absence de schéma qu'il faut compenser par des index ([McHugh et al., 1998], [McHugh and Widom, 1999], [Abiteboul et al., 2000]). Cinq types d'index sont utilisés dans Lore : le Vindex (*Value index* i.e. un index numérique), le Lindex (*Link index*) qui permet de suivre un lien (ou de le remonter), le Bindex qui, étant donné un label, retourne toutes les paires de nœuds liés par ce label, le Tindex (*full text index*) et le Pindex (*Path index*). Depuis, des index indexant simultanément le texte et les chemins ont été développés [Weigel et al., 2004, Gardarin and Yeh, 2005].

Dans cette section, seul l'un de ces index nous intéresse : le Pindex sur les chemins qui est communément appelé dataguide.

Définition 4.1 *Étant donné une donnée semi-structurée D on appelle dataguide de D toute donnée déterministe* DG_D *qui vérifie que :*

- *Chaque chemin de D a exactement une instance dans* DG_D*,*
- *Chaque chemin de* DG_D *est un chemin de D.*
- *Pour tout couple de mots (l,l') de $A^* \times A^*$,*

$$D(l) = D(l') \text{ si et seulement si } \mathrm{DG}_D(l) = \mathrm{DG}_D(l')$$

4.2.1 Algorithme de construction du dataguide

On donne maintenant une série de propriétés montrées dans [Caron et al., 2003], mais qui sont déjà évoquées dans [Milo and Suciu, 1999]. Pour cela on voit une donnée comme un automate : tous les nœuds du graphe sont des états finaux et tous les nœuds racines sont des états initiaux.

Proposition 4.1 *Le dataguide* DG_D *d'une donnée* D *est unique (à un isomorphisme près) et il est le résultat de l'application sur la donnée* D *de l'algorithme classique de déterminisation des automates finis de mots [Hopcroft and Ullman, 1979].*

Preuve :

1. *Montrons d'abord que* DG_D *est unique (à un isomorphisme près).*
   Supposons que pour une donnée D *il existe deux dataguides différents (*DG_D *et* $\mathrm{DG'}_D$*) et construisons un isomorphisme entre ces index. Soit* l *un chemin de* DG_D*, il est alors un chemin de* D *et donc de* $\mathrm{DG'}_D$ *(définition des dataguides).*
   - *Bijection entre les nœuds.*
     On définit l'application Φ *qui à un nœud* N_1 *de* DG_D *associe un nœud* N_2 *de* $\mathrm{DG'}_D$ *par le processus suivant. Soit* c_1 *un chemin de* DG_D *qui atteint* N_1*, on note alors* N_2 *le nœud associé par ce même chemin dans* $\mathrm{DG}_{D'}$*. S'il existe* c'_1 *un autre chemin qui atteint* N_1 *dans* DG_D*, alors* c_1 *et* c'_1 *ont le même résultat dans* DG_D *ils ont donc le même ensemble résultat dans* D *et le même résultat dans* $\mathrm{DG}_{D'}$ *(i.e. la construction de* N_2 *est indépendante du chemin choisi). Le fait que* DG_D *et* $\mathrm{DG}_{D'}$ *soient déterministes permet d'affirmer que* Φ *est effectivement une application.*
     Notons $|\mathrm{DG}_D|$ *le nombre de nœuds de* DG_D*.* $|\mathrm{DG}_D|$ *est égal au nombre de résultats différents que l'on peut atteindre avec des chemins de* D*. Par définition d'un dataguide,* $\mathrm{DG}_{D'}$ *a exactement la même taille.*
     Soit un nœud N_2 *de* $\mathrm{DG}_{D'}$*. On peut l'atteindre par un chemin* c_2 *dans* $\mathrm{DG}_{D'}$*. Ce chemin atteint* N_1 *dans* DG_D*. Il est ensuite rapide de vérifier que* N_1 *est un antécédent de* N_2 *par* Φ*. C'est à dire que* Φ *est surjective.*
     Φ *est donc une application surjective entre deux ensembles finis ayant le même cardinal. Elle est donc bijective.*
   - *Bijection entre les arcs.*
     On montre, par récurrence, que si un chemin c *atteint un nœud* N_c^1 *dans* DG_D *alors le chemin* c *atteint* $N_c^2 = \Phi(N_c^1)$ *dans* $\mathrm{DG}_{D'}$*.*
     • *si le chemin est vide le résultat est évident.*
     • *si le résultat est vrai pour un chemin* c *montrons qu'il est vrai pour le chemin* $c.l$ *(*l *est un label).*
     Dans $\mathrm{DG}_{D'}$ c *atteint le nœud* $\Phi(N_c^1)$*. L'arc* $(N_c^2, l, N_{c.l}^2)$ *existe (car* $\mathrm{DG}_{D'}$ *est déterministe et que tout chemin de* DG_D *est aussi chemin de* $\mathrm{DG}_{D'}$*) et donc* $c.l$ *atteint* $N_{c.l}^2 = \Phi(N_{c.l}^1)$*.*
   DG_D *et* $\mathrm{DG'}_D$ *sont donc isomorphes et la dataguide est unique.*
2. *Il est facile de montrer que le déterminisé* $det(D)$ *d'une donnée* D *vérifie les trois points de la définition du dataguide et que donc* $det(D)$ *est un dataguide. On sait de plus que le dataguide est unique (point 1) et donc* $det(D)$ *est le dataguide du donnée* D*.*

◄

C'est en utilisant ce résultat que R. Goldman et J. Widom ont présenté une méthode de mise à jour du dataguide dans [Goldman and Widom, 1999]. Une mise à jour consiste à recalculer le déterminisé de la donnée mais uniquement en suivant les chemins modifiés.

Maintenant que l'on a défini le dataguide il faut savoir comment on peut l'utiliser en tant qu'index. Pour construire le dataguide DG_D on a utilisé l'algorithme de déterminisation, il est donc facile de construire la relation *types* qui lie un nœud de DG_D à un ensemble de nœuds

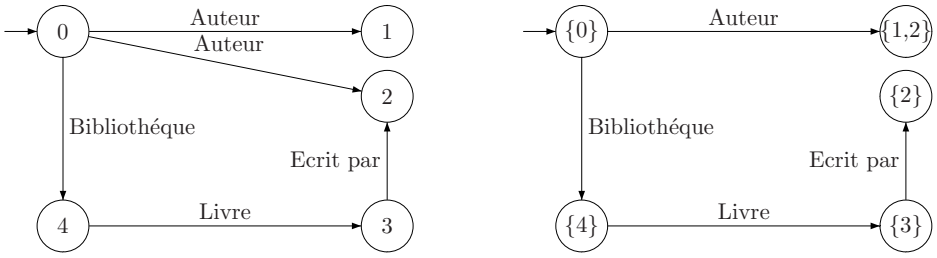


FIG. 4.2 – Une donnée et son dataguide

de D. En effet, tout nœud t du dataguide correspond à un ensemble e de nœuds de la donnée. On définit donc *types*(t) comme l'ensemble e. La construction du dataguide par la méthode des sous-ensembles permet de faire le lien entre un type t et les mots u qui atteignent t dans l'index :

Lemme 4.1 *Soient D une donnée semi-structurée, DG_D son dataguide et u un mot. Notons t_u le nœud (le type) atteint par u dans DG_D et $types(t_u)$ l'ensemble des nœuds de D ayant le type t_u. On a alors :*

$$types(t_u) = D(u)$$

Preuve : Raisonnons par récurrence sur la longueur de u. Soit $D =< N, Racines, T >$ une donnée semi-structurée.

- *Si $u = \varepsilon$ alors $D(\varepsilon) = Racines = t_\varepsilon$*
- *Si $u = vx$ alors $D(u) = D(D(v), x) = D(types(t_v), x) = types(t_u)$ d'après l'algorithme de déterminisation.*

◀

Les auteurs de [Goldman and Widom, 1997] déduisent de ces définitions que le dataguide est un guide couvrant (i.e. un index). En effet si q est une requête et que $\mathrm{DG}_D(q)$ est le résultat de q sur DG_D on a alors $D(q)$ le résultat de q sur D qui se calcule facilement par :

$$D(q) = \bigcup_{t \in DG_D(q)} (types(t)) \tag{4.1}$$

Un corollaire de ce lemme fait le lien entre contraintes d'inclusions et sous-typage

Corollaire 4.1 *Soit D une donnée semi-structurée, DG_D son dataguide, u et v deux mots Notons t_u (reps. t_v) le nœud (le type) atteint par u (resp. v) dans DG_D. On a alors :*

$$D \models u \preceq v \text{ si et seulement si } types(t_u) \subseteq types(t_v)$$

Exemple 4.2.1 : La figure 4.2 représente une donnée et son dataguide. On construit la relation *types* entre les nœuds du dataguide et ceux de la donnée. Par exemple *types* ($\{1,2\}$) =$\{1,2\}$ ou bien *types* ($\{4\}$) =$\{4\}$. On déduit de $\mathrm{DG}_D(Auteur + Bibliotheque) = \{\{1,2\},\{4\}\}$ que $D(Auteur + Bibliotheque) = \{1,2,4\}$.

De plus, le dataguide étant par définition déterministe, l'algorithme qui permet de calculer $\mathrm{DG}_D(q)$ est plus efficace que celui qui calcule $D(q)$. Mais cela cache une mauvaise propriété des dataguides qui est leur taille par rapport à la donnée initiale :

Proposition 4.2 *La taille de* DG_D *peut être exponentielle par rapport à la taille de* D.

Pour prouver cette proposition il suffit de dire qu'elle est le corollaire de la proposition 4.1.

Pour réduire la taille du dataguide R. Goldman et J. Widom proposent dans [Goldman and Widom, 1999] plusieurs méthodes permettant de regrouper des nœuds du dataguide. Malheureusement, ces nouveaux guides ne conservent pas forcément l'ensemble des chemins et ils ne sont donc pas couvrants.

En ce qui concerne les contraintes d'inclusions, la donnée représentée dans la figure 4.2 nous montre que les contraintes d'inclusions présentes dans D ne le sont pas forcément dans DG_D. En effet D est un modèle de $\mathit{Bibliothèque.Livre.EcritPar} \preceq \mathit{Auteur}$ alors que le dataguide ne l'est pas. L'information concernant les contraintes d'inclusions n'est pas tout à fait perdue puisque l'ensemble $\mathit{types}(\mathrm{DG}_D(\mathit{Bibliothèque.Livre.EcritPar}))$ est inclus dans l'ensemble types $(\mathrm{DG}_D(\mathrm{Auteur}))$.

4.2.2 Saturation du dataguide

Nous proposons donc un algorithme de saturation, inspiré de la construction des automates résiduels [Denis et al., 2001], qui ajoute des transitions au dataguide pour conserver les contraintes d'inclusions de mots. On propose d'ajouter les transitions (t', x, t) dans DG_D s'il existe une transition (t', x, t'') dans DG_D telle que $\mathit{types}(t)$ est inclus dans $\mathit{types}(t'')$. En effet, comme le montre l'équation 4.1, si $\mathit{types}(t)$ est inclus dans $\mathit{types}(t'')$ alors tout mot atteignant t'' dans DG_D, atteint tous les nœuds de $\mathit{types}(t)$ dans D. On obtient alors la définition suivante :

Définition 4.2 *Soit* D *une donnée et* $\mathrm{DG}_D =< N, \{r\}, T >$ *son dataguide. On définit le saturé* DGS_D *du dataguide* DG_D *comme étant le graphe* $< N, R', T' >$
- $R' = \{t \in N \mid \mathit{types}(t) \subseteq \mathit{types}(r)\}$,
- $T' = \{(t', x, t) \mid \exists t'' \in N,\ (t', x, t'') \in T \wedge \mathit{types}(t) \subseteq \mathit{types}(t'')\}$.

En considérant le cas pour lequel t'' et t sont égaux, on prouve que DGS_D contient toutes les transitions de DG_D. On peut aussi avoir une vision 'automate' du problème et ajouter au dataguide des transitions ε entre un type t'' et un type t si $\mathit{types}(t)$ est inclus dans $\mathit{types}(t'')$.

Exemple 4.2.2 : La figure 4.3 représente une donnée et son dataguide saturé. Comme $\mathit{types}(\{4\})$ est inclus dans $\mathit{types}(\{3,4\})$ on ajoute (en pointillés) au nœud $\{4\}$ des transitions correspondant aux transitions entrantes dans le nœud $\{3,4\}$.

Comme $\mathit{types}(\{1\})$ est inclus dans $\mathit{types}(\{1,2\})$ et que $(\{0\}, b, \{1,2\})$ est une transition, on ajoute la transition $(\{0\}, b, \{1\})$ et c'est ainsi que la contrainte $a \preceq b$ se transmet de D vers DGS_D.

On va pouvoir énoncer le résultat concernant les contraintes d'inclusions.

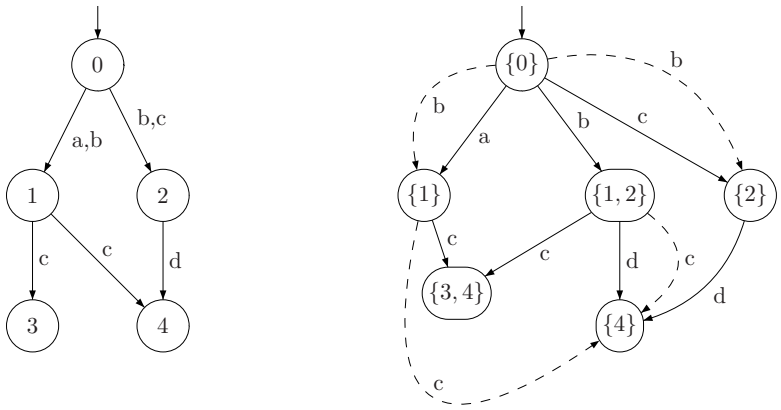


FIG. 4.3 – Représentation du dataguide saturé

Proposition 4.3 ([Andre et al., 2005]) *Soient D une donnée, DGS_D le dataguide saturé associé à D, u et v deux mots.*

D est un modèle de la contrainte $u \preceq v$ si et seulement si DGS_D est un modèle de $u \preceq v$.

La preuve se base sur le lemme suivant.

Lemme 4.2 *Soit t un nœud du dataguide (un type de la donnée D), u un mot et t_u le nœud atteint par u dans le dataguide :*

$$types(t) \subseteq types(t_u) \text{ si et seulement si } t \in \mathrm{DGS}_D(u)$$

Preuve :

On montre les deux sens de l'équivalence en raisonnant par récurrence sur la longueur de u.

- *Si u est le mot vide alors t appartient à $\mathrm{DGS}_D(\varepsilon)$ si et seulement si $types(t)$ est inclus dans $D(\varepsilon)$ i.e. $types(t_\varepsilon)$.*
- *Si $u = vx$. Notons t_u le nœud $\mathrm{DG}_D(u)$ et t_v le nœud $\mathrm{DG}_D(v)$.*
  Si $types(t)$ est inclus dans $types(t_u)$ alors, par le principe de saturation, la transition (t_v, x, t_u) appartenant à DG_D on ajoute la transition (t_v, x, t) dans DGS_D et donc t est accessible par u dans DGS_D.
  Réciproquement, supposons que t soit accessible par u dans DGS_D. Soit w un mot et t_w le noeud accessible par w dans DG_D tel que t_w appartient à $\mathrm{DGS}_D(v)$ et la transition (t_w, x, t) existe dans DGS_D (tous les états du dataguide étant accessibles, le mot w existe). Cette transition existe car le mot wx atteint le nœud $t_{wx} = \mathrm{DG}_D(wx)$ et que (t_w, x, t_{wx}) est une transition de DG_D et que $types(t)$ est inclus dans $types(t_{wx})$. Par récurrence, on sait que $types(t_w)$ est inclus dans $types(t_v)$. D'après le corollaire 4.1, $D \models w \preceq v$ et donc $D \models wx \preceq vx$. Ce même corollaire permet de dire que $types(t_{wx})$ est inclus dans $types(t_u)$. On conclut en utilisant la transitivité : $types(t) \subseteq types(t_{wx}) \subseteq types(t_u)$

◀

Terminons maintenant la preuve de la proposition 4.3. Le lemme précédent signifie que $\mathrm{DGS}_D(u)$ est l'ensemble $\{t \mid types(t) \subseteq types(t_u)\}$. On a alors $D \models u \preceq v$ si et seulement si $types(t_u)$ est inclus dans $types(t_v)$ si et seulement si $\mathrm{DGS}_D(u)$ est inclus dans $\mathrm{DGS}_D(v)$ car pour tout mot v, t_v appartient à $\mathrm{DGS}_D(v)$.

On peut donc conclure que D et DGS_D satisfont le même ensemble de contraintes de mots, et, du fait que DG_D est un guide couvrant de D, un corollaire de cette proposition est que DGS_D est aussi un index de D. En effet $D(q) = \bigcup_{t \in DGS_D(q)} types(t)$.

Malheureusement cette preuve ne s'étend pas à toutes les contraintes d'inclusions. En effet la donnée présentée dans la figure 4.3 est un modèle de $b \preceq a + c$ alors que son dataguide saturé ne vérifie pas cette contrainte. Une fois de plus le mode de construction du dataguide va nous aider. En effet bien que la contrainte semble perdue, on a quand même l'information que $types(\mathrm{DG}_D(b))$ est inclus dans l'union entre $types(\mathrm{DG}_D(a))$ et $types(\mathrm{DG}_D(c))$. Cette dernière remarque nous permettra, dans la section suivante, de proposer un algorithme qui extrait les contraintes satisfaites par une donnée D de son dataguide.

On peut donc résumer le positionnement des dataguides vis-à-vis de nos objectifs dans le tableau suivant :

	Taille de l'index	Préservation des contraintes
Dataguide	Exponentielle (au pire)	Non
Dataguide saturé	Exponentielle (au pire)	Oui (contraintes sur les mots)

Il faut remarquer que le dataguide saturé est meilleur que le dataguide en ce qui concerne les contraintes d'inclusions mais pour atteindre cet objectif nous avons construit un index non déterministe de taille exponentielle (dans le pire des cas) par rapport à la taille de la donnée. Du point de vue de l'optimisation de requêtes cela n'est pas satisfaisant. Par contre, on peut extraire du dataguide (saturé) de D un ensemble fini de contraintes finies qui implique les contraintes vérifiées par D.

4.2.3 Extraction des contraintes satisfaites par une donnée

Dans cette section, on exploite le lien entre le dataguide et les contraintes de mots (mis en évidence dans la section précédente) pour étudier un problème lié à la notion de modèle exact. On répond à la question "est-ce que toute donnée semi-structurée est un modèle exact d'un ensemble de contraintes d'inclusions ?". La réponse est *oui*. De plus, si D est finie il existe alors un ensemble $\mathcal{C}(D)$ fini de contraintes *finies* tel que D soit un modèle exact de $\mathcal{C}(D)$.

Proposition 4.4 *Pour toute donnée finie D, il existe un ensemble fini de contraintes finies d'inclusions $\mathcal{C}(D)$ tel que $\mathcal{C}(D) \models p \preceq q$ si et seulement si $D \models p \preceq q$.*

Preuve : Soit $D =< N, R, T >$ une donnée et $S(D) = \{D(u) \mid u \in A^*\}$ l'ensemble des nœuds de son dataguide.

La preuve utilise le corollaire 3.1 qui stipule que D est un modèle exact de $\mathcal{C}$ si

$$D \models p \equiv q \text{ si et seulement si } \mathcal{C} \models p \equiv q.$$

Pour tout ensemble s de $S(D)$, $\mathrm{lex}(s)$ est un représentant de l'ensemble $\{u \mid D(u) = s\}$. On peut, par exemple choisir le plus petit et le premier dans l'ordre alphabétique. La seule restriction est que $\mathrm{lex}(R)$ doit être ε.

On construit alors l'ensemble $\mathcal{C}(D)$ comme suit :

$$\begin{array}{ll} \mathcal{C}(D) = & \{\mathrm{lex}(s)x \equiv \mathrm{lex}(s') \mid s' = \{n' \mid \exists n \in s \wedge (n, x, n') \in T\}\} \\ \cup & \{\mathrm{lex}(s_1) + \ldots + \mathrm{lex}(s_n) \equiv \mathrm{lex}(s'_1) + \ldots + \mathrm{lex}(s'_k) \mid \underset{1 \le i \le n}{\cup} s_i = \underset{1 \le i \le k}{\cup} s'_i\} \end{array}$$

Par construction, D est un modèle de $\mathcal{C}(D)$ et donc $\mathcal{C}(D) \models p \equiv q$ implique que $D \models p \equiv q$.

Il reste à prouver que D est exact.

Montrons par récurrence sur la longueur de u que $\mathcal{C}(D) \models u \equiv \mathrm{lex}(D(u))$. Si $u = \varepsilon$ alors $D(u) = R$ et $\mathrm{lex}(R) = \varepsilon$ permet de conclure. Si u est de la forme vx alors, par hypothèse de récurrence, $\mathcal{C}(D) \models v \equiv \mathrm{lex}(D(v))$. De plus, la contrainte $(\mathrm{lex}(D(v)).x \equiv \mathrm{lex}(D(u)))$ appartient à $\mathcal{C}(D)$ et on peut donc conclure par $\mathcal{C} \models u \equiv v.x$, $\mathcal{C} \models v.x \equiv \mathrm{lex}(D(v))x$ et enfin $\mathcal{C} \models \mathrm{lex}(D(v))x \equiv \mathrm{lex}(D(u))$.

Ce résultat s'étend, par union infinie, au cas des requêtes régulières :

$$\mathcal{C}(D) \models q \equiv \underset{u \in L(q)}{+} \mathrm{lex}(D(u))$$

.

Si $D \models p \equiv q$ (p et q sont deux requêtes régulières), on déduit de

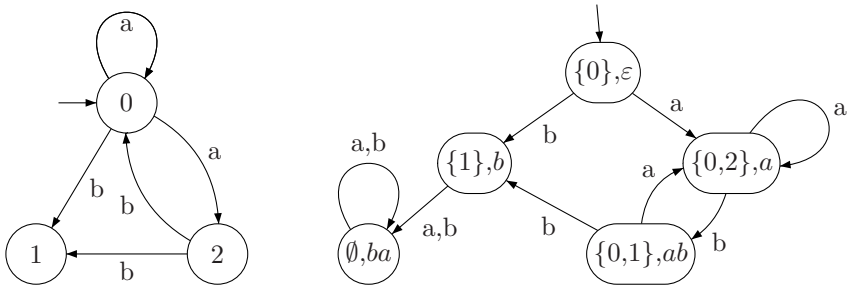
$$\underset{u \in L(p)}{\cup}(D(u)) = \underset{u \in L(q)}{\cup}(D(u))$$

que

$$\underset{u \in L(p)}{\cup}(\mathrm{lex}(D(u))) \equiv \underset{u \in L(q)}{\cup}(\mathrm{lex}(D(u)))$$

appartient à $\mathcal{C}(D)$ et donc $\mathcal{C}(D) \models p \equiv q$. ◄

Exemple 4.2.3 : Soit la donnée suivante construite sur l'alphabet $A = \{a, b\}$:



L'ensemble de nœuds du dataguide est $S(D) = \{\{0\}, \{0, 2\}, \{0, 1\}, \{1\}, \emptyset\}$,
Chaque nœud a pour représentant le mot le plus court (et le premier dans l'ordre alphabétique) qui l'atteint : $\mathrm{lex}(\{0\}) = \varepsilon$, $\mathrm{lex}(\{0, 2\}) = a$, $\mathrm{lex}(\{0, 1\}) = ab$, $\mathrm{lex}(\{1\}) = b$, $\mathrm{lex}(\emptyset) = ba$.
On extrait enfin un ensemble fini de contraintes finies (dont on donne une sous-partie)

$$\begin{aligned}\mathcal{C}(D) = \quad & \{a \equiv a, a \equiv aa, aba \equiv a, abb \equiv b, bb \equiv ba,\\ & ba + \varepsilon \equiv \varepsilon, ba + a \equiv a, ba + ab \equiv ab, ba + b \equiv b\\ & \varepsilon + a \equiv a,\ \varepsilon + ab \equiv ab, b + ab \equiv ab, b + \varepsilon \equiv ab\\ & baa \equiv ba, bab \equiv ba\\ & \varepsilon + b + a + ab \equiv \varepsilon + b + a + ab + ba \ldots\ \}.\end{aligned}$$

L'ensemble $S(D)$ qui sert de base à la construction de $\mathcal{C}(D)$ pouvant être exponentiellement plus grand que D, l'ensemble $\mathcal{C}(D)$ peut être doublement exponentiel par rapport à D. Le but de la proposition suivante est de réduire cette taille maximale (en bornant la taille des contraintes présentes dans $\mathcal{C}(D)$) et de donner une borne supérieure pouvant être atteinte.

Comme le montre l'exemple, des contraintes contenues dans $\mathcal{C}(D)$ peuvent être impliquées par d'autres contraintes. En effet la contrainte $c = \varepsilon+b+a+ab \equiv \varepsilon+b+a+ab+ba$ est présente dans l'ensemble car $\{0\} \cup \{1\} \cup \{0,2\} \cup \{0,1\} = \{0,1,2\} = \{0\} \cup \{1\} \cup \{0,2\} \cup \{0,1\} \cup \emptyset$. Mais l'ensemble $\{0,1,2\}$ peut être obtenu à partir de l'union entre $\{0,1\}$ et $\{0,2\}$ i.e. les contraintes $\varepsilon + b + a + ab \equiv \mathbf{a} + \mathbf{ab}$ et $\varepsilon + b + a + ab + ba \equiv \mathbf{a} + \mathbf{ab}$ sont aussi présentes dans $\mathcal{C}(D)$, impliquent la contrainte c et ont une taille inférieure à c.

Proposition 4.5 *Soit D une donnée semi-structurée finie. Il existe un ensemble $\mathcal{C}'(D)$ de contraintes d'inclusions tel que :*
- *$\mathcal{C}'(D) \models p \preceq q$ si et seulement si $D \models p \preceq q$*
- *dans le pire des cas, $\mathcal{C}'(D)$ est exponentiellement plus grand que D*

Preuve : Considérons l'ensemble $\mathcal{C}(D)$ défini dans la preuve de la proposition 4.4 et supprimons des contraintes qui sont impliquées par d'autres.

Une contrainte de la forme $\mathrm{lex}(s_1) + \ldots + \mathrm{lex}(s_n) \equiv \mathrm{lex}(s'_1) + \ldots + \mathrm{lex}(s'_k)$ correspond en fait à deux contraintes : $\mathrm{lex}(s_1) + \ldots + \mathrm{lex}(s_n) \preceq \mathrm{lex}(s'_1) + \ldots + \mathrm{lex}(s'_k)$ et $\mathrm{lex}(s'_1) + \ldots + \mathrm{lex}(s'_k) \preceq \mathrm{lex}(s_1) + \ldots + \mathrm{lex}(s_n)$. Rappelons aussi que la contrainte $\mathrm{lex}(s_1) + \ldots + \mathrm{lex}(s_n) \preceq \mathrm{lex}(s'_1) + \ldots + \mathrm{lex}(s'_k)$ équivaut à n contraintes de la forme $\mathrm{lex}(s_i) \preceq \mathrm{lex}(s'_1) + \ldots + \mathrm{lex}(s'_k)$.

Pour réduire la taille de $\mathcal{C}(D)$ on propose de considérer uniquement des contraintes dont le membre droit est de longueur inférieure au nombre de nœuds de D. L'idée est de trouver un recouvrement de l'ensemble $\bigcup_{1\le i\le k} s'_i$ et de compléter la contrainte grâce au résultat suivant : soient p, q et r trois requêtes régulières

$$p \preceq q \Rightarrow p \preceq q + r \tag{4.2}$$

Soit
$$\begin{aligned}\mathcal{C}'(D) = \quad & \{\mathrm{lex}(s)x \equiv \mathrm{lex}(s') \mid s' = \{n' \mid \exists n \in s \wedge (n, x, n') \in T\}\} \cup\\ & \{\mathrm{lex}(s) \preceq \mathrm{lex}(s'_1) + \ldots + \mathrm{lex}(s'_k) \mid s \subseteq \bigcup_{1\le i\le k\le |noeuds(D)|} s'_i\}\end{aligned}$$

Montrons que toute contrainte appartenant à $\mathcal{C}(D)$ est impliquée par $\mathcal{C}'(D)$. Le problème réside dans les contraintes de la forme $\mathrm{lex}(s) \preceq \mathrm{lex}(s'_1) + \ldots + \mathrm{lex}(s'_k) \mid s_i \subseteq \bigcup_{1\le i\le k} s'_i$ avec k strictement plus grand que le nombre de nœuds de D.

Définissons la suite $\{i_n\}_{1\le n\le m}$ ($m \le k$) par $i_1 = 1$ et $i_{n+1} = min\{i_n \le j \le k \mid s'_j \not\subseteq \bigcup_{l<j} s'_l\}$ (m est le plus grand indice tel que i_{m+1} n'existe pas). Deux remarques sont à faire sur cette suite :

– *m est plus petit que le nombre de nœuds D. En effet la suite $|\bigcup_n s'_{i_n}|$ est positive, strictement croissante et bornée par $|noeuds(D)|$.*
– *$\bigcup_{1\leq i\leq k} s'_i = \bigcup_{1\leq n\leq m} s'_{i_n}$ par construction.*

On déduit donc que la contrainte $\text{lex}(s_1) \preceq \text{lex}(s'_{i_1}) + \ldots + \text{lex}(s'_{i_m})$ appartient à $\mathcal{C}(D')$ et on conclut avec la relation 4.2.

Pour être complet, il faut donner une borne supérieure de la taille de $\mathcal{C}'(D)$. Il y a un nombre exponentiel de contraintes de la forme $\text{lex}(s)x \equiv \text{lex}(s')$ et la taille de chaque contrainte est en $O(|noeuds(DG_D)|)$. De plus, le nombre de contraintes de la forme $\text{lex}(s_1) \preceq \text{lex}(s'_1) + \ldots + \text{lex}(s'_k)$ est aussi exponentiel (car k est plus petit que $|noeuds(DG_D)|$) et chaque contrainte a une taille quadratique dans le nombre de nœuds de DG_D. Comme le dataguide de D a un nombre, dans le pire des cas exponentiel de nœuds par rapport au nombre de nœud de D, que la taille maximale d'une contrainte est en $O(|D|^2)$ la taille de $\mathcal{C}'(D)$ est, dans le pire des cas, exponentiellement plus grande que D. ◀

Remarque 4.1 – *L'ensemble $\mathcal{C}'(D)$ contient encore des redondances.*
– *Malgré ces redondances, il existe des données D pour lesquelles le nombre de contraintes non redondantes dans $\mathcal{C}'(D)$ est exponentiel. Il suffit de prendre une donnée qui possède un dataguide exponentiellement plus grand qu'elle.*
– *Par contre, on n'a pas encore de résultat concernant la "dureté" du problème ce qui signifie que l'on ne sait pas s'il est possible d'extraire un ensemble de contraintes correspondant à nos critères qui n'ait pas une taille exponentiellement plus grande que D.*

Nous avons vu dans le chapitre 3 que la complexité des problèmes liés aux contraintes d'inclusions dépend de la forme des contraintes. Il est donc naturel de se demander à quelles conditions les contraintes de l'ensemble $\mathcal{C}'(D)$ extraites du dataguide de D sont bornées.

Proposition 4.6 *Soit D une donnée semi-structurée finie. Nous pouvons décider en EXPTIME (dans la taille de D) s'il existe un ensemble de contraintes bornées dont D est le modèle exact.*

Preuve :

La preuve se fait en deux temps. On donne d'abord une caractérisation des ensembles $\mathcal{C}'(D)$ de contraintes d'inclusions qui sont équivalents à un ensemble de contraintes bornées. On en déduit un algorithme EXPTIME.

Soit $\mathcal{C}'(D)$ l'ensemble de contraintes extraites de D. Il existe $\mathcal{C}_b(D)$ un ensemble de contraintes d'inclusions bornées équivalent à $\mathcal{C}'(D)$ si et seulement si

$$\forall(\text{lex}(s) \preceq \text{lex}(s'_1) + \ldots + \text{lex}(s'_k)) \in \mathcal{C}'(D))\ \exists 1 \leq j \leq k \mid \mathcal{C}'(D) \models \text{lex}(s) \preceq \text{lex}(s'_j) \in \mathcal{C}'(D) \tag{4.3}$$

En effet

1. *S'il existe $\mathcal{C}_b(D)$ équivalent à $\mathcal{C}'(D)$ alors*
   $\mathcal{C}'(D) \models \text{lex}(s_1) \preceq \text{lex}(s'_1) + \ldots + \text{lex}(s'_k)$ si et seulement si

$\mathcal{C}_b(D) \models \text{lex}(s_1) \preceq \text{lex}(s'_1) + \ldots + \text{lex}(s'_k)$ *si et seulement si*
$\exists j \mid \mathcal{C}_b(D) \models \text{lex}(s_1) \preceq \text{lex}(s'_j)$ *si et seulement si (proposition 3.5)*
$\exists j \mid \mathcal{C}'(D) \models \text{lex}(s_1) \preceq \text{lex}(s'_j)$ *si et seulement si*
$\exists j \mid D(\text{lex}(s_1)) \subseteq D(\text{lex}(s'_j))$ *si et seulement si*
$\exists j \mid (\text{lex}(s_1) \preceq \text{lex}(s'_j) \in \mathcal{C}'(D)$ *par définition de* $\mathcal{C}'(D)$.

2. *Si* $\forall(\text{lex}(s_1) \preceq (\text{lex}(s'_1) + \ldots + \text{lex}(s'_k)) \in \mathcal{C}'(D)) \; \exists 1 \leq j \leq k \mid (\text{lex}(s_1) \preceq \text{lex}(s'_j) \in \mathcal{C}'(D)$.
Soit $\mathcal{C}_b(D) = \{\text{lex}(s) \preceq \text{lex}(s') \mid (\text{lex}(s) \preceq \text{lex}(s')) \in \mathcal{C}'(D)\}$ *un ensemble de contraintes d'inclusions bornées.*
Nous devons prouver que $\mathcal{C}_b(D) \models p \preceq q$ *si et seulement si* $\mathcal{C}'(D) \models p \preceq q$.
$\forall(\text{lex}(s) \preceq \text{lex}(s')) \in \mathcal{C}_b(D) \; \mathcal{C}'(D) \models \text{lex}(s) \preceq \text{lex}(s')$ *est évident.*
Si $(\text{lex}(s_1) \preceq (\text{lex}(s'_1) + \ldots + \text{lex}(s'_k))) \in \mathcal{C}'(D)$ *alors*
$\exists 1 \leq j \leq k \mid (\text{lex}(s_1) \preceq \text{lex}(s'_j) \in \mathcal{C}'(D)$ *(par hypothèse) i.e.*
$\exists 1 \leq j \leq k \mid (\text{lex}(s_1) \preceq \text{lex}(s'_j) \in \mathcal{C}_b(D)$ *i.e.* $\mathcal{C}_b(D) \models \text{lex}(s_1) \preceq \text{lex}(s'_j)$ *i.e.*
$\mathcal{C}_b(D) \models \text{lex}(s_1) \preceq (\text{lex}(s'_1) + \ldots + \text{lex}(s'_k))$ *(équation 4.2).*

L'algorithme suivant est une traduction directe de cette caractérisation :

```
a_un_équivalent_borné(C)
% Entrée : un ensemble de contraintes C = {u_i ≼ v_1 + ... + v_{n_i} | 1 ≤ i ≤ m}
% Sortie : vrai si et seulement si C satisfait la condition 4.3
% Locales : est_borné et bool deux booléens
est_borné = vrai
pour tout u_i ≼ v_1 + ... + v_{n_i} ∈ C faire
   bool = faux
   pour j allant de 1 à n_i faire
      si (u_i ≼ v_j ∈ C) alors bool = vrai
   fin pour
   est_borné = est_borné ∧ bool
fin pour
retourner est_borné
```

La boucle externe est effectuée autant de fois qu'il y a de contraintes dans $\mathcal{C}$ *et le nombre d'exécutions de la boucle interne dépend de la taille des contraintes dans* $\mathcal{C}$. *Si on se ramène à notre problème, il y a un nombre exponentiel de contraintes (dans la taille de D) et la taille des contraintes est exponentielle dans la taille de D. L'algorithme, dans sa globalité, est donc EXPTIME dans la taille de D.* ◄

Dans la section suivante nous allons présenter d'autres méthodes d'indexations. Ces méthodes vont fusionner deux nœuds équivalents et ainsi avoir une taille qui soit toujours inférieure (ou égale) à la taille de la donnée d'origine. Par contre la notion de déterminisme n'apparaîtra plus.

4.3 Fusionner des nœuds indifférenciables par les requêtes

Dans cette section nous allons présenter l'index *perfect* introduit par S. Nestorov, S. Abiteboul et R. Motwani dans [Nestorov et al., 1998] et le 1-index qui a été introduit par T. Milo et

D. Suciu dans [Milo and Suciu, 1999]. Tous ces auteurs sont partis du constat que le dataguide était trop volumineux par rapport à la donnée initiale.

L'idée (principalement évoquée par T. Milo et D. Suciu) est d'utiliser une relation d'équivalence $\equiv$ sur les nœuds de la donnée initiale définie par $n \equiv n'$ si : un mot u atteint le nœud n si et seulement s'il atteint le nœud n'. Plus précisément on définit, pour un nœud n de D, le langage rationnel $L_n(D)$ de tous les mots qui atteignent n depuis l'une des racines de D. On définit alors une relation d'équivalence sur les nœuds par :

$$n \equiv n' \text{ si } L_n(D) = L_{n'}(D)$$

Dans ce cas, indexer une donnée D consiste à construire le quotient de D par une relation d'équivalence. L'algorithme d'indexation doit alors calculer, pour tout nœud n d'une donnée, sa classe d'équivalence $[n]$. En effet, le quotient $D_{/\equiv}$ de D par une relation d'équivalence $\equiv$ se construit comme suit :

Définition 4.3 *Soit $D =< N, R, T >$ une donnée. Son quotient par $\equiv$ est la donnée $D_{/\equiv} =< N_q, R_q, T_q >$ définie par :*
- $N_q =< [n] \mid n \in N >$
- $R_q =< [n] \mid \exists r \in R,\ r \equiv n >=< [r] \mid r \in R >$
- $T_q = \{([n], x, [n']) \mid \exists n_1, n'_1,\ n_1 \equiv n \wedge n'_1 \equiv n' \wedge (n_1, x, n'_1) \in T\}$.

Construire un index par quotient permet de construire simultanément la relation *types* . En effet à tout nœud $[n]$ de l'index on associe l'ensemble des nœuds de D qui sont dans la classe de $[n]$ i.e.

$$types([n]) = \{n' \in D \mid n' \in [n]\} = [n]$$

Le problème de la relation $\equiv$ est que le calcul des classes d'équivalence est très coûteux. En effet tester si deux nœuds n et n' sont équivalents revient à tester si les langages $L_n(D)$ et $L_{n'}(D)$ sont égaux. Il est connu de [Stockmeyer and Meyer, 1973] que tester l'égalité entre deux langages rationnels est PSPACE-complet.

L'idée est donc de ne pas prendre $\equiv$ mais une relation $\equiv_b$ qui en soit un raffinement (i.e. si $n \equiv_b n'$ alors $n \equiv n'$) mais dont le calcul des classes d'équivalence est plus rapide. De plus, le quotient de D par un raffinement de $\equiv$ a les trois propriétés suivantes :

Proposition 4.7 *[Milo and Suciu, 1999] Soient D une donnée, $\equiv$ la relation d'équivalence sur les nœuds de D définie précédemment et $\equiv_b$ un raffinement de $\equiv$ alors*

1. $L_n(D) = L_{[n]}(D_{/\equiv_b})$
2. *$D_{/\equiv_b}$ est un index couvrant de D.*
3. *D et $D_{/\equiv_b}$ ont le même dataguide (i.e. l'automate déterministe construit avec l'algorithme des sous ensembles appliqué à $D_{/\equiv_b}$ coïncide avec le dataguide de D).*

Le lien entre nos objectifs et les deux premiers items est clair. Le troisième point nous garantit que $D_{/\equiv_b}$ et D satisfont exactement les même contraintes d'inclusions. Il faut en effet se souvenir que l'on peut extraire du dataguide de D un ensemble de contraintes $\mathcal{C}(D)$ qui

implique une contrainte $p \preceq q$ si et seulement si elle est satisfaite par D. Comme D et $D_{/\equiv_b}$ ont le même dataguide, les ensembles $\mathcal{C}(D)$ et $\mathcal{C}(D_{/\equiv_b})$ sont égaux et donc D et $D_{/\equiv_b}$ satisfont les mêmes contraintes [Andre et al., 2005].

Utiliser un quotient comme index nous garantit de bons résultats sur la taille. En effet la taille du quotient est toujours plus petite que la taille de la donnée d'origine. Plus généralement, si $\sim$ est un raffinement de $\sim'$, alors la taille de $D_{/\sim'}$ est plus petite que la taille de $D_{/\sim}$.

Pour résumer, quel que soit $\equiv_b$ un raffinement de $\equiv$, le quotient $D_{/\equiv_b}$ est un index candidat qui satisfait tous nos critères sauf un : en effet, nous n'avons pas encore pris en compte le temps de construction de l'index. Il nous faut donc trouver des raffinements de $\equiv$ pour lesquels le calcul de l'index (i.e. le calcul des classes d'équivalences) soit efficace. La littérature propose deux index, basés sur des bi-simulations [Park, 1981].

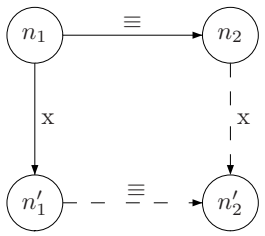
Trois définitions de bi-simulation existent suivant que l'on considère les arcs entrants dans un noeud, les arcs sortants ou les deux.

Définition 4.4 *Soient D une donnée et N l'ensemble de ses nœuds. On dit qu'une relation $\sim$ sur $N \times N$ est une Forward bi-simulation de D si $\sim$:*

- *préserve les racines : si n_1 est une racine de D et $n_1 \sim n_2$ alors n_2 est une racine de D (et vice-versa),*
- *préserve les arcs sortants : si $n_1 \sim n_2$ alors pour tout arc (n_1, x, n_1') dans D il existe un arc (n_2, x, n_2') dans D tel que $n_1' \sim n_2'$ (et vice versa).*

On dit alors que deux nœuds n_1 et n_2 sont Forward bi-similaires ($n_1 \equiv_{Fbs} n_2$) s'il existe une Forward bi-simulation $\sim$ qui contienne le couple (n_1, n_2). Cette relation est réflexive, symétrique et transitive. C'est donc une relation d'équivalence. Elle est de plus la plus grossière des Forward bi-simulations. Dans la suite, elle est notée F-bi-simulation.

On représente souvent la F-bi-simulation par le diagramme suivant :



Définition 4.5 *Soient D une donnée et N l'ensemble de ses nœuds. On dit qu'une relation $\sim$ sur $N \times N$ est une Backward bi-simulation de D si $\sim$:*

- *préserve les racines : si n_1 est une racine de D et $n_1 \sim n_2$ alors n_2 est une racine de D (et vice-versa),*
- *préserve les arcs entrants : si $n_1 \sim n_2$ alors pour tout arc (n_1', x, n_1) dans D il existe un arc (n_2', x, n_2) dans D tel que $n_1' \sim n_2'$ (et vice versa).*

On dit alors que deux nœuds n_1 et n_2 sont Backward bi-similaires ($n_1 \equiv_{Bbs} n_2$) s'il existe une Backward bi-simulation $\sim$ qui contienne le couple (n_1, n_2). Cette relation est réflexive,

symétrique et transitive. C'est donc une relation d'équivalence. Elle est de plus la plus grossière des Backward bi-simulations. Dans la suite, elle est notée B-bi-simulation.

Une autre relation de bi-simulation est utilisée dans ce chapitre :

Définition 4.6 *Soient D une donnée et N l'ensemble de ses nœuds. On dit qu'une relation $\sim$ sur $N \times N$ est une Forward and Backward bi-simulation de D si $\sim$:*
- *préserve les racines : si n_1 est une racine de D et $n_1 \sim n_2$ alors n_2 est une racine de D (et vice-versa),*
- *préserve les arcs sortants : si $n_1 \sim n_2$ alors pour tout arc (n_1, x, n'_1) dans D il existe un arc (n_2, x, n'_2) dans D tel que $n'_1 \sim n'_2$ (et vice versa).*
- *préserve les arcs entrants : si $n_1 \sim n_2$ alors pour tout arc (n'_1, x, n_1) dans D il existe un arc (n'_2, x, n_2) dans D tel que $n'_1 \sim n'_2$ (et vice versa).*

Dans ce cas on dit que deux nœuds n_1 et n_2 sont *Forward and Backward* bi-similaires (que l'on note $n_1 \equiv_{FBbs} n_2$) s'il existe une *Forward and Backward* bi-simulation $\sim$ entre n_1 et n_2. Elle est de plus la plus grossière des *Forward and Backward* bi-simulations. Dans la suite, elle est notée FB-bi-simulation.

4.3.1 Algorithme de Paige et Tarjan

Le calcul d'un index basé sur une relation de bi-simulation demande donc de calculer des classes d'équivalence d'une bi-simulation. L'algorithme de Paige et Tarjan [Paige and Tarjan, 1987] est, à ma connaissance, le plus efficace pour cette tâche.

L'objectif de Paige et Tarjan est de résoudre le problème suivant :

Définition 4.7 *Le problème du raffinement d'une partition consiste à calculer*
- *A partir d'un ensemble fini E*
- *A partir d'une partition initiale p_I de E,*
- *A partir d'un ensemble fini $\{R_i\}_{1 \leq i \leq n}$ de relations binaires sur E.*

le raffinement le plus grossier de p_I qui soit stable par rapport à la F-bi-simulation.

Afin de bien spécifier l'algorithme de Paige et Tarjan, rappelons quelques définitions :

Définition 4.8
- *Raffinement : une partition K est un raffinement de la partition L si chaque élément de K est inclus dans un élément de L.*
- *Stable par rapport à la F-bi-simulation : si ρ est une partition, $n \equiv_\rho n'$ signifie que n et n' sont dans un même ensemble de ρ. Une partition ρ est stable par rapport à la F-bi-simulation si quel que soit le triplet (n, n', n_1) d'éléments de E $n \equiv_\rho n'$ et nR_in_1 implique qu'il existe n'_1 dans E tel que $n_1 \equiv_\rho n'_1$ et $n'R_in'_1$.*
- *Grossier : p est le raffinement le plus grossier de p_I qui soit stable par rapport à la F-bi-simulation si tout raffinement r de p_I qui soit stable par rapport à la F-bi-simulation a plus d'éléments que p.*

La version initiale de l'algorithme de Paige et Tarjan permet de calculer la F-bi-simulation dans des graphes non étiquetés [Paige and Tarjan, 1987, Alur and Henzinger, 2004] : dans ce cas E désigne l'ensemble des nœuds du graphe et l'ensemble $\{R_i\}_{1\leq i\leq n}$ est réduit à un singleton. Il contient l'unique relation représentant les arcs du graphe.

Néanmoins, les auteurs de [Fernandez, 1990, Kerbrat, 1994] proposent une extension de cet algorithme au cas des graphes étiquetés. Pour tous les labels l de la donnée, R_l est la relation binaire contenant tous les couples de nœuds (n, n') tels qu'il existe une transition (n, l, n') dans le graphe. On raffine alors par rapport à l'ensemble de relations $\{R_l\}_{l\in\Sigma}$.

Pour terminer on peut citer [Buneman et al., 1997, Milo and Suciu, 1999] et dire qu'il est possible de calculer la B-bi-simulation sur les nœuds d'une donnée semi-structurée en utilisant l'algorithme de Paige et Tarjan. Il suffit en effet de raffiner la partition initiale des nœuds d'une donnée par rapport aux relations de l'ensemble $\{R_l^{-1}\}_{l\in\Sigma}$. Si on considère l'ensemble $\{R_l\}_{l\in\Sigma} \cup \{R_l^{-1}\}_{l\in\Sigma}$ on obtient un algorithme pour calculer la FB-bi-simulation.

Dans tous les articles cités ci-dessus et dans la suite de cette thèse, nous considérons que la partition initiale sépare les racines des autres nœuds i.e. pour une donnée $D =< N, R, T >$, p_I sera toujours l'ensemble $\{R, N \setminus R\}$.

On donne l'algorithme correspondant au calcul de la F-bi-simulation en deux temps : on donne tout d'abord un algorithme dont le coût est quadratique puis on montre comment en déduire la réduction de R. Paige et R. E. Tarjan.

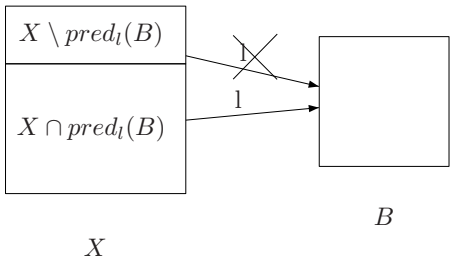


FIG. 4.4 – Partition d'un ensemble de nœuds

L'algorithme est un algorithme de raffinements successifs : à partir de la partition initiale p_I, on partitionne les classes par rapport à la F-bi-simulation. De manière générale, si σ et τ sont deux ensembles de nœuds d'une donnée D, alors partitionner σ par rapport à τ revient à calculer :

$$\text{partitionne}(\sigma, \tau) = \begin{cases} \{\sigma\} & \text{si } \sigma \subseteq \tau \text{ ou } \sigma \cap \tau = \emptyset \\ \{\sigma \cap \tau, \sigma \setminus \tau\} & \text{sinon} \end{cases}$$

Si ρ est un ensemble d'ensembles de nœuds, alors partitionne(ρ, τ) partitionne tous les ensembles de ρ par rapport à τ :

$$\text{partitionne}(\rho, \tau) = \bigcup_{\sigma\in\rho} \text{partitionne}(\sigma, \tau)$$

Pour calculer la F-bi-simulation, on utilise donc la fonction *partitionne* pour raffiner une partition des nœuds d'une donnée. Il reste donc à identifier les classes d'une partition qui ne

sont pas stables par rapport à la F-bi-simulation. Comme le montre la figure 4.4, une classe X n'est pas stable s'il existe une classe B et un label l tels que les deux propositions suivantes ne soient pas équivalentes :

1. il existe un noeud n_X de X et un noeud n_B de B tel que (n_X, x, n_B) est un arc de la donnée D.
2. pour tout noeud n_X de X il existe un noeud n_B de B tel que (n_X, x, n_B) est un arc de la donnée D.

Tous les éléments sont définis pour présenter l'algorithme 4.1 qui, à partir de deux classes B et X, partitionne X par rapport aux labels entrants dans B.

```
Partitionne_Σ(X, B)
% Entrée : Deux ensembles X et B de nœuds d'une donnée D
% Sortie : Une partition de X telle que
%          chaque élément de la partition soit stable par rapport à B
% Locale : La partition P de X
P = {X}
pour tous les labels l de Σ faire
   pour toutes les classes C de P faire
   pred_l(B) = {n_c ∈ C | ∃n_b ∈ B ∧ (n_c, l, n_b) est une transition de D}
      remplacer C par partitionne(C,pred_l(B)) dans P
   fin pour
fin pour
retourner P
fin
```

TAB. 4.1 – Algorithme de partition

Il est possible d'implémenter la fonction Partitionne$_\Sigma(X, B)$ telle que sa complexité soit linéaire par rapport au nombre d'arcs entrants dans B. On peut alors obtenir un algorithme quadratique en (|nœuds(D)|.|arc(D)|) pour le calcul de la F-bi-simulation. En effet, dans le pire des cas, chaque nœud de la donnée appartient à une classe différente et il y a donc |nœuds(D)| appels à *Partitionne*$_\Sigma$. De plus, toujours dans le pire des cas, chaque appel à cette fonction a un coût linéaire par rapport au nombre de transitions de la donné semi-structurée.

Exemple 4.3.1 : La figure 4.5 représente une donnée dont on veut calculer les classes d'équivalence par rapport à la F -bi-simulation. On montre les partitions successives construites grâce à la procédure *Partitionne*$_\Sigma$ présentée ci-dessus (seuls les appels raffinant la partition sont visibles).

Partition initiale :	{0}, {1,2,3,4,5,6,7,8}
Partitionne$_\Sigma(\{1,2,3,4,5,6,7,8\},\{1,2,3,4,5,6,7,8\})$:	{0}, {5,6,7}, {1,2}, {3,4,8}
Partitionne$_\Sigma(\{3,4,8\},\{5,6,7\})$:	{0}, {5,6,7}, {1,2}, {3}, {4,8}
Partitionne$_\Sigma(\{1,2\},\{3\})$:	{0}, {5,6,7}, {1}, {2}, {3}, {4,8}

La partition { {0}, {5,6,7}, {1}, {2}, {3}, {4,8} } est stable.

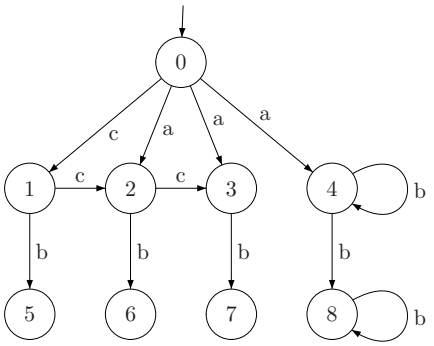


FIG. 4.5 – Exemple de calcul de la F-bi-simulation

R. Paige et R. E. Tarjan proposent de généraliser l'algorithme de J. E. Hopcroft [Aho et al., 1974] qui minimise le nombre d'états d'un automate déterministe pour améliorer le calcul de la F-bi-simulation. Intuitivement l'idée est de garder trace de la manière dont une classe a été partitionnée en sous classes lors d'un raffinement. En effet, si $\{B_1, B_2\}$ est obtenu par raffinement du bloc $B = B_1 \cup B_2$, on peut calculer $partitionne_\Sigma(X, B)$ à partir de $partitionne_\Sigma(X, B_1)$ et de $partitionne_\Sigma(X, B_2)$. Retenir, comment une classe B a été partitionnée en $B_1 \cup B_2$, va permettre de stocker toute l'information nécessaire pour déduire un algorithme pour partitionner $B = B_1 \cup B_2$ en utilisant uniquement l'ensemble B_1. La fonction *partitionne* est donc en $O(min(|B_1|, |B_2|))$, au lieu de l'algorithme en $O(|B_1| + |B_2|)$ utilisé jusqu'à maintenant.

% Entrée : une donnée semi-structurée $D =< N, R, T >$
% Sortie : les classes de la F-bi-simulation.
% Locales : deux partitions p_1 et p_2
$p_1 = p_I = \{R, N \setminus R\}$
$p_2 = \{N\}$
tant que $p_1 \subsetneq p_2$ faire
% Invariant : p_1 est stable par rapport à toutes les régions de p_2
  Choisir B dans $p_2 \setminus p_1$
  Choisir B_1 dans p_1 tel que $B_1 \subseteq B$ et $|B_1| \leq |B|/2$
  % Autrement dit $B = B_1 \cup B_2$ et $|B_1| \leq |B_2|$
  $p_1 = partitionne_\Sigma(p_1, B_1, B \setminus B_1)$
  $p_2 = (p_2 \setminus \{B\}) \cup \{B_1, B \setminus B_1\}$
fin tq
retourner p_1
fin

TAB. 4.2 – Algorithme de Paige et Tarjan

Comment calculer $partitionne_\Sigma(X, B_1, B_2)$ **efficacement ?**

L'optimisation est basée sur la proposition suivante. Dans cette proposition, si ρ et ρ' sont deux ensembles d'ensembles de nœuds, alors $\rho \sqcap \rho'$ désigne l'ensemble $\{B \cap B' \mid B \in \rho, B' \in \rho'\}$:

Proposition 4.8 *Soient ρ une partition d'un ensemble N, X une classe de ρ et B un ensemble inclus dans N tel que X est stable par rapport à B. Si B se décompose en $B_1 \cup B_2$, alors* $partitionne_\Sigma(X, B) = partitionne_\Sigma(X, B_1, B_2) =$
$partitionne_\Sigma(X, B_1) \sqcap partitionne_\Sigma(X, B_2) = \bigsqcap_{l \in \Sigma} \{X_1^l, X_2^l, X_3^l\}$ *avec* $X_1^l = (X \cap \mathrm{pred}_l(B_1)) \setminus \mathrm{pred}_l(B_2)$
$X_2^l = (X \cap \mathrm{pred}_l(B_2)) \setminus \mathrm{pred}_l(B_1)$
$X_3^l = (X \cap \mathrm{pred}_l(B_1)) \cap \mathrm{pred}_l(B_2)$
chacun des X_i^l pouvant être vide.

On définit la fonction nb_succ$_B(n,l)$ qui retourne le nombre de successeurs étiquetés par l du nœud n dans B : si $succ_l(n)$ désigne l'ensemble des nœuds qui ont un arc étiqueté par l à partir n, alors $nb\_succ_B(n,l) = |succ_l(n) \cap B|$ et on construit les X_i^l en itérant les trois règles suivantes sur tous les nœuds n de X :

$$\frac{nb\_succ_{B_1}(n,l) = nb\_succ_B(n,l)}{X_1^l = X_1^l \cup \{n\}} \quad \text{(R1)}$$

$$\frac{nb\_succ_{B_1}(n,l) = 0}{X_2^l = X_2^l \cup \{n\}} \quad \text{(R2)}$$

$$\frac{0 < nb\_succ_{B_1}(n,l) < nb\_succ_B(n,l)}{X_3^l = X_3^l \cup \{n\}} \quad \text{(R3)}$$

Comment obtient-on un algorithme en $O(|D|.lg(|D|))$ **?**

La complexité de l'algorithme qui découle de ces trois règles dépend de B, B_1 et de la fonction nb_succ. Il suffit de choisir B_1 comme étant la plus petite des deux classes de B pour évaluer $nb\_pred_{B_1}(n,l)$. Il reste néanmoins un gros problème de complexité : la fonction nb_succ$_B$. Il est évident que le pré-calcul des valeurs de cette fonction est exponentiel. On va donc définir une structure de données qui permette à la fois d'avoir des partitionneurs de la forme $B_1 \cup B_2$ et les valeurs de nb_succ associées. Cette structure s'appelle arbre de raffinement et permet de conserver des informations sur les raffinements déjà effectués.

Définition 4.9 *L'arbre de raffinement d'une donnée D est un arbre binaire A tel que :*
- *Chaque nœud de A est une classe X de 2^N. On note ce noeud X.*
- *Un noeud X de fils X_1 et X_2 est tel qu'il existe une classe Y de 2^N, un label l tel que $(X_1, X_2) = partitionne_l(X, Y)$.*
- *La racine est la partition initiale (i.e. { N }).*
- *Les feuilles forment une partition de N.*

Le lecteur peut se référer à [Paige and Tarjan, 1987, Fernandez, 1990, Kerbrat, 1994] pour avoir tous les algorithmes (construction et mises à jour de l'arbre A) qui garantissent que le calcul des X_i^Σ, et donc celui de partitionne$_\Sigma(X, B_1, B_2)$, peut se faire en $O(min(|B_1|, |B_2|))$.

Considérons l'algorithme de Paige et Tarjan présenté dans la table 4.2. Soit $B^i = B_1^i \cup B_2^i$ ($|B_1^i| \leq |B_2^i|$) l'ensemble choisi pour la partition lors de la i-ème itération de la boucle *tant que*. Si un noeud n appartient à deux ensembles B_1^i et B_1^j ($i < j$) alors $|B_1^j| \leq |B_1^j|/2$. Comme la taille des B^i est bornée par le nombre de nœuds présents dans la donnée, il existe au plus $lg(|\text{nœuds}(D)|)$ indice i pour lesquels n appartienne à B_1^i. Le noeud n est choisi donc au plus $lg(|\text{nœuds}(D)|)$ fois pour partitionner un ensemble. Autrement dit, pour tout nœud n de D les arcs sortants de n sont utilisés au plus $lg(|\text{nœuds}(D)|)$ fois i.e. l'algorithme est en $O(|\text{arc}(D)|.lg(|\text{nœuds}(D)|)$.

4.3.2 Index Perfect : fusionner des nœuds FB-bi-similaires

Dans [Nestorov et al., 1998], S. Nestorov, S. Abiteboul et R. Motwani proposent déjà de typer les nœuds d'une donnée en utilisant la notion d'index sur les chemins. Partant du principe qu'une donnée semi-structurée est irrégulière mais contient des informations similaires (les pages d'un annuaire d'un laboratoire contiennent beaucoup d'informations similaires - nom, téléphone, émail, ... mais certaines de ces pages peuvent avoir un champ manquant), les auteurs de [Nestorov et al., 1998] proposent de regrouper dans un même type des nœuds qui "se ressemblent". L'une de leurs méthodes consiste à regrouper des nœuds qui sont FB-bi-similaires dans un même type. Cette méthode de typage s'appelle *perfect*. Nous proposons de construire l'index *perfect* à partir de ces types.

Pour construire cet index il faut regrouper dans la donnée initiale des nœuds qui ont les mêmes chemins entrants et sortants. Pour cela on dit que deux nœuds sont équivalents s'ils sont en FB-bi-simulation.

Il existe plusieurs algorithmes permettant de calculer cet index. Comme le suggèrent les auteurs de [Nestorov et al., 1998], l'index *perfect* peut se calculer avec l'algorithme de Paige et Tarjan en $O(|D|.lg(|D|))$ (cf. section précédente).

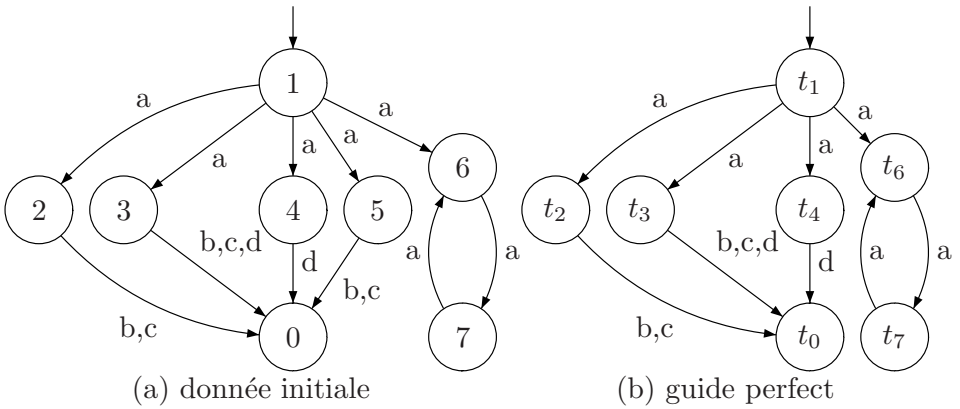
A partir de la relation $\equiv_{FBbs}$, on peut définir un graphe perfect(D) appelé index perfect de D en faisant le quotient de D par $\equiv_{FBbs}$. Les nœuds sont des classes d'équivalence de $\equiv_{FBbs}$ et les arcs sont tels que : (T, x, T') est un arc de perfect(D) si et seulement s'il existe un nœud n dans T, un nœud n' dans T' tels que soit (n, x, n') un arc de D.

Remarque 4.2 *Deux nœuds ayant le même type ont les mêmes chemins entrants et sortants. La réciproque n'est pas toujours vraie. Par exemple, bien que les nœuds 4 et 5 de la donnée représentée par la figure 4.7 aient les même chemins entrants et sortants, ils n'ont pas le même type. En effet les noeuds 1 et 2 ne sont pas FB-bi-similaires.*

Comme perfect(D) est le quotient de D par la relation $\equiv_{FBbs}$, on a la proposition suivante (qui est un corollaire de la proposition 4.7) :

Proposition 4.9 *Soient D une donnée,* perfect(D) *l'index perfect associé, n un nœud de D dont la classe d'équivalence est $[n]$:*

$$n \in D(u) \text{ si et seulement si } [n] \in \text{perfect}(D)(u)$$



(a) donnée initiale (b) guide perfect

FIG. 4.6 – Guide perfect.

Exemple 4.3.2 : La figure 4.6 montre une donnée initiale et le guide perfect associé. Dans cet exemple on a pu fusionner les nœuds 2 et 5 dans le nœud t_2. D'après la définition de nos classes d'équivalence on a $types(t_2)$ qui est égal à l'ensemble {2,5}. Remarquons enfin que ces deux nœuds ont exactement le même ensemble de chemins entrants $\{a\}$ et exactement le même ensemble de chemins sortants $\{b, c\}$.

Tous les éléments permettant de définir et de construire l'index *perfect* sont désormais connus. On peut donc l'évaluer vis-à-vis de nos objectifs :

Proposition 4.10

- *On peut borner la taille de l'index perfect par la taille de la donnée initiale.*
- *Les contraintes d'inclusions sont préservées.*
- *L'index perfect peut se construire avec un algorithme en* $O(|D|.lg(|D|))$.

Preuve :

Les deux premiers points sont des conséquences des résultats généraux montrés dans 4.3. La complexité a été étudiée dans la section consacrée à l'algorithme de Paige et Tarjan. ◄

Exemple 4.3.3 : Comme dans l'étude des dataguides, la donnée extraite de la base moviesBD (figure 4.1 ; page 91) va nous donner un exemple d'index de taille maximale. Le nœud 1 et le nœud 200 ne peuvent pas être fusionnés : `films.film.acteur.joueDans` est un chemin entrant pour le nœud 1 mais ne l'est pas pour le nœud 200 (c'est un chemin distinguant). En itérant ce procédé sur le label `suivant`, on montre qu'aucune fusion n'est possible et que l'index perfect de cette donnée est égal à la donnée d'origine.

Ces résultats correspondent aux objectifs fixés mais il faut les nuancer : l'index perfect n'est pas, dans la majorité des cas, beaucoup plus petit que la donnée initiale. En effet toute irrégularité dans la donnée va être source de nouveaux chemins et donc de nouveaux nœuds dans le guide. Le guide perfect va donc avoir un bon comportement sur les données régulières qui sont plus du domaine des bases de données classiques que des données semi-structurées.

Dans [Nestorov et al., 1998], le type perfect est en réalité utilisé pour initialiser un algorithme de K-clustering qui va typer la donnée semi-structurée avec exactement K types. Le guide ainsi obtenu est une approximation de l'index perfect et ne préserve plus les chemins.

On peut donc résumer le positionnement des index perfect vis-à-vis de nos objectifs :

- La taille de l'index *perfect* est bornée par la taille de la donnée
- Une contrainte est satisfaite par la donnée si et seulement si elle l'est par l'index *perfect*.

4.3.3 1-index : fusionner des nœuds B-bi-similaires

T. Milo et D. Suciu ont constaté que l'index perfect était trop 'semblable' à la donnée initiale et qu'il n'était pas utile de prendre en compte les arcs sortants d'un nœud. Ils définissent donc un 1-index [Milo and Suciu, 1999] par :

Définition 4.10 *Soit D une donnée semi-structurée. On appelle 1-index de D, noté 1-index(D), le quotient de D par la relation de Backward bi-simulation.*

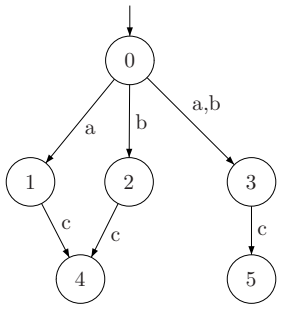


FIG. 4.7 – Donnée dans la quelle il n'y pas de nœuds B-bi-similaires

$\equiv_B$ étant un raffinement de $\equiv$, deux nœuds fusionnés ont le même langage d'arcs entrants. La figure 4.7 montre que, comme dans le cas de l'index *perfect*, la réciproque est fausse. En effet L_4 et L_5 sont égaux et valent $ac+bc$ mais les nœuds 4 et 5 se sont pas en B-bi-simulation : le nœud 3 n'est B-bi-similaire à aucun autre nœud. En effet c'est le seul nœud de la donnée qui soit accessible par a et par b.

Deux nœuds qui sont en FB-bi-simulation (fusionnés dans l'index *perfect*) sont aussi B-bi-similaires (i.e. fusionnés dans le 1-index). On déduit donc que la taille du 1-index est bornée par celle de l'index *perfect*, et donc (par transitivité) la taille du 1-index est bornée par la taille de D.

Tous les cas intermédiaires sont possibles. Une donnée peut avoir un 1-index qui se réduise à un nœud. A l'opposé, la donnée construite sur la base de IMDB (figure 4.1 - page 91) montre une donnée dans la quelle deux nœuds différents ont toujours deux langages de chemins entrants différents. Dans ce cas, le 1-index est égal au guide perfect qui est lui même égal à la donnée. Une étude expérimentale a été menée par R. Kaushik et al [Kaushik et al., 2002b] :

ils ont montré qu'en moyenne la taille d'un 1-index était environ de 45% la taille de la donnée d'origine.

Exemple 4.3.4 : La figure 4.8 montre une donnée dont l'index *perfect* et le 1-index sont différents. Les nœuds 1 et 2 sont B-bi-similaires et FB-bi-similaires. Par contre les nœuds 2 et 3 sont B-bi-similaires mais ne sont pas FB-bi-similaires (le nœud 2 a un arc b sortant que le nœud 3 n'a pas). On a expliqué pourquoi le 1-index est plus petit que l'index perfect.

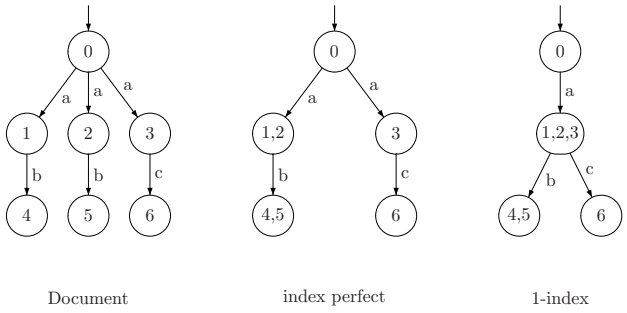


FIG. 4.8 – Différence entre l'index perfect et le 1-index

Pour terminer notre étude sur le 1-index, il faut rappeler que le 1-index se calcule avec l'algorithme de Paige et Tarjan [Paige and Tarjan, 1987]. Autrement dit, on peut le construire en $O(|D|.lg(|D|))$.

Le 1-index respecte nos trois critères. On lui reproche parfois d'être trop ressemblant à la donnée d'origine (par exemple le 1-index de la donnée construite à partir de la base IMDB - figure 4.1 - est égale à la donnée d'origine). C'est pourquoi d'autres méthodes d'indexation ont été proposées sur la base des 1-index : les A(k)-index ([Kaushik et al., 2002b],[Yi et al., 2004]) et les D(k)-index ([Chen et al., 2003]). Ces deux algorithmes se basent sur la même constatation : il existe très peu de requêtes qui matchent un long chemin de la donnée. Typiquement les auteurs de [Kaushik et al., 2002b] considèrent qu'une requête matche au maximum un chemin de longueur 8 de la donnée. Pour un nœud n on ne considère plus le langage L_n mais le langage L_n^k composé de tous les mots de longueur inférieure ou égale à k qui atteignent le nœud n. On va alors regrouper deux nœuds n et n' si L_n^k est égal à $L_{n'}^k$. Pour cela on utilise une k-B-bi-simulation. Si une requête matche un chemin de longueur plus grande que k, l'index fournit un sur-ensemble de la réponse. Il faut alors vérifier sur la donnée initiale quels sont les nœuds qui sont réellement accessibles par la requête. Le A(k)-index n'est donc plus un guide couvrant. Par contre, des études expérimentales effectuées par les auteurs de [Kaushik et al., 2002b] montrent qu'utiliser un A(k)-index permet, en moyenne, un gain de temps de 50% par rapport à un 1-index.

Ce tableau résume tous les résultats démontrés ou énoncés dans cette section. Pour être complet il faut rappeler que l'on sait extraire du dataguide d'une donnée D un ensemble fini $\mathcal{C}(D)$ de contraintes finies telles que D soit le modèle exact de $\mathcal{C}(D)$.

	Taille maximum de l'index	préservation des contraintes
Dataguide	Exponentielle	Non
Dataguide saturé	Exponentielle	Oui (contraintes de mots)
Index perfect	Taille de la donnée	Oui
1-index	Taille de l'index perfect	Oui

4.4 Ensemble de données

Le but de cette section est d'étudier les contraintes communes à un ensemble de données en se basant sur des résultats montrés dans [Andre et al., 2005]. En effet, un ensemble E de données peut se représenter par une unique donnée (à partir d'un ensemble E de données, on construit la donnée D_E comme étant l'union de toutes les données) et toutes les techniques et tous les résultats décrits dans ce chapitre peuvent donc être utilisés.

Définition 4.11 *Soit $E = \{D_i =< N_i, R_i, T_i >\}_{1 \leq i \leq n}$ un ensemble de données. La donnée $D_E =< N_E, R_E, T_E >$ est définie comme suit :*

- $N_E = \bigcup_{1 \leq i \leq n} N_i$
- $R_E = \bigcup_{1 \leq i \leq n} R_i$
- $T_E = \bigcup_{1 \leq i \leq n} T_i$

Représenter l'ensemble E par la donnée D_E convient à nos objectifs. On peut en effet montrer que D_E satisfait une contrainte d'inclusion si et seulement si elle est satisfaite par toutes les données de E.

L'extraction des contraintes communes à toutes les données d'un ensemble E est possible. Le dataguide d'une donnée D permet de construire un ensemble $\mathcal{C}(D)$ fini de contraintes d'inclusions finies qui implique une contrainte si et seulement si elle est satisfaite par D. Si on applique ce résultat à la donnée D_E on obtient :

$$\mathcal{C}(D_E) \models p \preceq q \text{ si et seulement si } \forall D \in E \; D \models p \preceq q$$

Cette méthode a toujours le même défaut : dans le pire des cas, la taille $\mathcal{C}(D_E)$ est exponentiellement plus grande que la somme de la taille des données présentes dans E.

De même, construire un index couvrant de E qui vérifie les contraintes d'inclusions satisfaites par toutes les données est un problème résolu puisque le 1-index de D_E convient. On peut néanmoins se demander si du point de vue des requêtes, il n'y a pas une meilleure utilisation du 1-index. Il y a en effet deux algorithmes, basés sur le 1-index, qui indexent un ensemble E de données :

1. Soit on indexe les données séparément et on considère l'union de tous les index. Dans ce cas on considère le U-index(E)= $\bigcup_{D \in E}$ 1-index(D).
2. Soit on indexe E en le considérant comme une unique donnée et dans ce cas on construit le 1-index(E) = 1-index(D_E).

La proposition 4.7 certifiant que le 1-index d'une donnée D satisfait exactement les mêmes contraintes d'inclusions de D, les trois points suivants sont équivalents :

1. $\forall D \in E\ D \models p \preceq q$
2. 1-index$(E) \models p \preceq q$
3. U-index$(E) \models p \preceq q$

On va donc comparer le 1-index de E et son U-index sur deux critères : leur taille et le temps nécessaire pour répondre à une requête. Si la donnée est beaucoup plus grande que la requête, évaluer une requête q sur une donnée D est $O(|D|.|q|^2)$ (cf. page 53). A requête fixe, les deux critères sont donc équivalents. C'est pourquoi, le second critère est évalué non pas lorsque l'on interroge toutes les données de E, mais lorsque l'on interroge un sous ensemble E' de données.

Du point de vue de la taille, la position de la taille du 1-index par rapport à celle du U-index n'est pas très difficile à établir : en effet deux nœuds fusionnés dans le U-index le sont forcément dans le 1-index. On a donc montré que la taille du 1-index est toujours plus petite que la taille du U-index. Étant donné que l'on considère des données qui appartiennent au même ensemble, on peut imaginer qu'elles sont similaires et que des nœuds de données différentes sont Backward bi-similaires. Dans ce cas, la taille du 1-index est significativement plus petite que celle du U-index. Cette remarque est confirmée par les expériences menées sur la base des films (figure 4.9) dont le résultat est présenté dans la figure 4.10. L'axe des abscisses correspond au nombre de données présentes dans l'ensemble alors que l'axe des ordonnées représente la taille de l'index par rapport à la taille des données d'origine. Rappelons que les auteurs de [Kaushik et al., 2002b] ont estimé (expérimentalement) que la taille du 1-index est de 45% celle de la donnée d'origine.

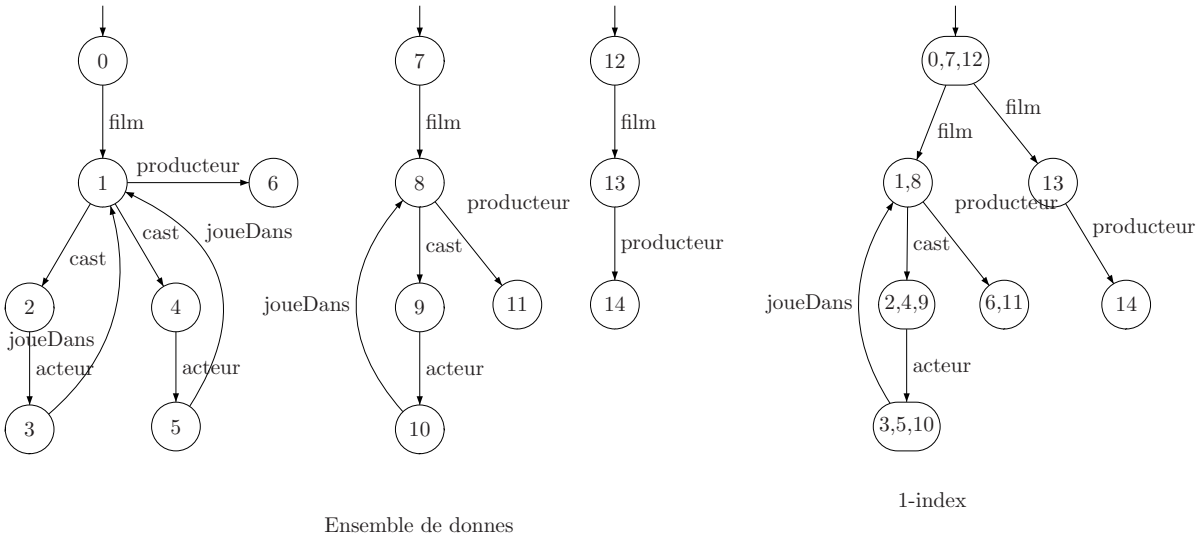


FIG. 4.9 – Base de données construite à partir de IMDB

Du point de vue du temps, on interroge un sous ensemble E' avec deux méthodes différentes. On peut, soit interroger 1-index(E), puis projeter le résultat sur les nœuds de E', soit questionner une sous partie de U-index(E) : U-index(E') et avoir directement le résultat.

Dans une implantation naïve, le temps utilisé pour répondre à une requête q sur un sous ensemble E' de E en utilisant 1-index(E) est indépendant de E'. En effet, calculer l'ensemble

1-index(E)(q) ne dépend pas de E'. La projection, quant à elle, parcourt tous les nœuds de cet ensemble et le temps consacré à cette opération est indépendant de E'. Une solution pour contourner ce problème, utilisée dans les expériences suivantes, est d'associer aux nœuds d'une même donnée des identifiants contigus et de simplifier la projection.

L'algorithme qui utilise U-index(E) dépend de E' : si E' ne contient qu'un élément, il est alors toujours plus rapide d'utiliser U-index(E'). A l'opposé, si E' est égal à E, alors l'utilisation du 1-index(E) est plus rapide. En plus du degré de ressemblance entre les données de E, la réponse à la question dépend donc de la fraction : $\frac{\text{taille de E'}}{\text{taille de E}}$.

On a donc mis en place l'expérience suivante : pour un ensemble de requêtes Q et pour un ensemble de données $E = \{D_i, 1 \leq i \leq n\}$, on construit tous les sous-ensembles $E_j = \{D_i,\ 1 \leq i \leq j \leq n\}$ et on calcule $E_j(q)$ pour tout j et pour toute requête q de Q.

L'une des mesures effectuées consiste à déterminer pour quel type de sous-ensemble le 1-index est plus adapté que le U-index et inversement. Pour cela on peut calculer le plus petit indice l pour lequel l'utilisation du 1-index est toujours plus rapide que celle du U-index et le plus grand indice l' pour lequel l'utilisation du U-index est toujours plus rapide que celle du 1-index. Le résultat de cette mesure est présenté dans la figure 4.11. Les expériences qui ont permis de construire la figure 4.11 ont été faites avec des bases de films. Le principe consiste à évaluer la valeur des limites l et l' en fonction du nombre de données. L'axe des abscisses représente donc le nombre de données présentes dans l'ensemble étudié et l'axe des ordonnées correspond à la taille relative (en pourcentage) d'un sous-ensemble E_j de E. Pour tout sous-ensemble E_j de E dont la taille relative est plus petite que l', il est plus intéressant d'utiliser le U-index. Si la taille relative est plus importante que l il est plus intéressant d'utiliser le 1-index. Les autres cas (entre l' et l) correspondent à la transition entre les deux méthodes d'indexation.

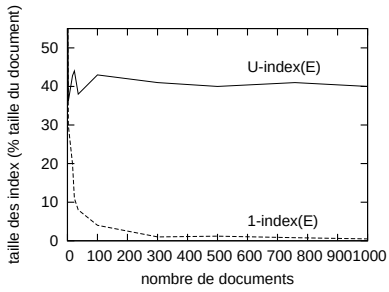


FIG. 4.10 – Expériences effectuées avec la base des films

Une autre expérience consiste à estimer le temps gagné (ou perdu) en utilisant le 1-index plutôt que le U-index. Pour cela on compare le temps moyen utilisé par le 1-index pour interroger tous les ensembles E_j avec toutes les requêtes de Q et le temps moyen utilisé par le U-index pour interroger les mêmes données avec les même requêtes. Les résultats de cette expérience sont présentés dans les figures 4.12 et 4.14. L'axe des abscisses correspond aux nombres de données dans l'ensemble E. L'axe des ordonnées de la figure 4.12 correspond au gain de temps effectué en utilisant le 1-index plutôt que le U-index (par exemple, pour 100 données, si le temps utilisé par le U-index est 100 alors le temps utilisé par le 1-index est 80,

ce qui donne un gain de 20). L'axe des ordonnées de la figure 4.14 correspond au temps utilisé par chaque index pour évaluer toutes les requêtes sur toutes les données.

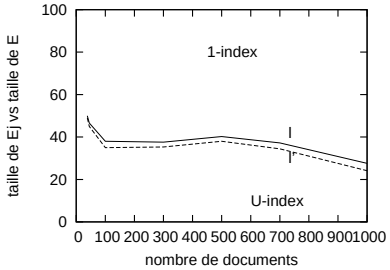


FIG. 4.11 – 1-index ou U-index ? (films)

On va maintenant présenter les trois jeux de données que nous avons utilisés pour obtenir ces résultats. Toutes les expériences ont été effectuées avec QRIC [Debarbieux and Luchier, 2004].

Ensemble 1 *On considère un ensemble de documents XML (avec les références). Ces données font référence aux films produits en 1966 (la figure 4.9). Elles ont été construites à partir de "The International Movie Database". Les exemples présentés dans la figure 4.9 montrent bien que les données se ressemblent mais ne sont pas identiques. On note en particulier que le nombre d'acteurs est différent d'une donnée à l'autre.*

Nos expériences, dont les résultats sont présentés dans la figure 4.10 montrent que le 1-index d'une donnée (et de l'ensemble) est d'environ 41% de la taille de la donnée d'origine alors que la taille du U-index est beaucoup plus grande. Il apparaît de plus que si l'ensemble considéré est assez gros, alors le 1-index devient constant. En effet, quand tous les cas particuliers sont stockés dans le 1-index, il ne grossit plus. On peut dire qu'il a atteint une taille limite.

La figure 4.12 (ligne films) confirme cette analyse. Elle montre que l'on peut indexer ensemble toutes les données de E sans perte d'efficacité même si on interroge un sous-ensemble de E.

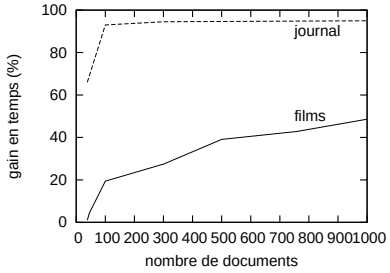


FIG. 4.12 – Gain de temps en utilisant 1-index(E)

Ensemble 2 *On a aussi fait des expériences avec un ensemble de données qui représentent des journaux (figure 1.2 ;page 4). On utilise ToXgene [Tox, 2002] pour construire des données aléatoirement. On a obtenu des ensembles tels que deux données différentes ont exactement le même (à isomorphisme près) 1-index i.e. on a la relation suivante (rappelons que D_E est la donnée qui représente l'ensemble E) :*

$$\forall D \in E \ \texttt{1-index}(D) = \texttt{1-index}(D_E)$$

Dans ce cas l'utilisation du 1-index est bien meilleure que celle du U-index comme le montre la figure 4.12 (ligne journal).

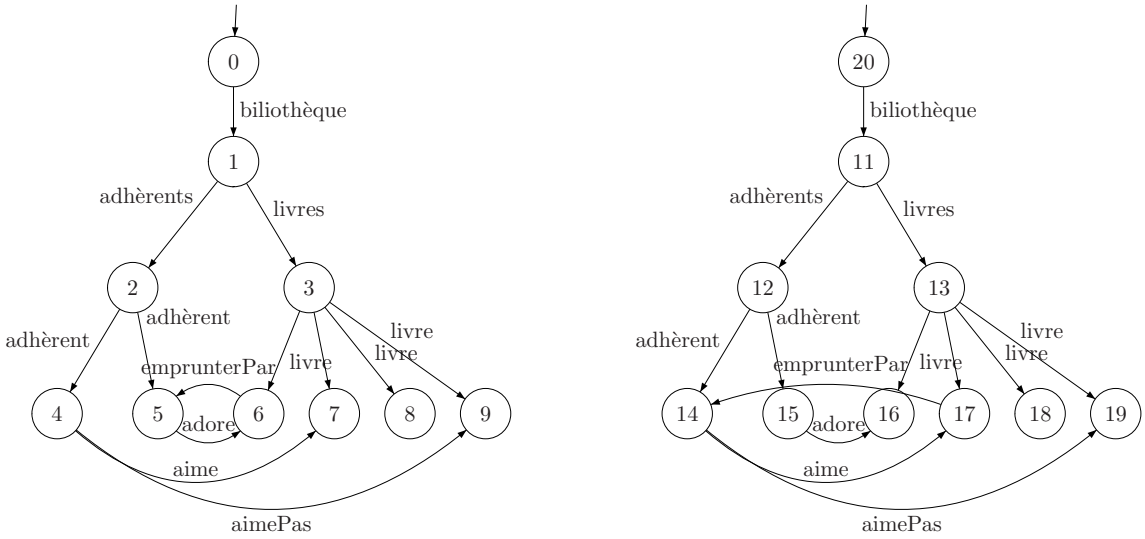


FIG. 4.13 – Ensemble de données décrivant des bibliothèques

Ensemble 3 *Le troisième type de données que nous avons considéré est représenté dans la figure 4.13. Une donnée représente une bibliothèque dans laquelle il y a des adhérents et des livres. Les adhérents peuvent donner leur opinion (*`adore`, `aime`, `aimePas`, `nonTerminé`*) sur les livres qu'ils ont lus. Les livres ont trois états : soit ils sont empruntés ou réservés par un adhérent, soit ils sont libres. Comme dans l'expérience précédente on utilise ToXgene [Tox, 2002] pour générer aléatoirement des ensembles de données.*

Nos expériences montrent que le 1-index a une taille qui est environ 58% celle de la donnée initiale. Malheureusement, cette taille n'est pas beaucoup plus petite que celle du U-index. En effet, nous avons établi, par des expériences que ces deux index ont des tailles très similaires (figure 4.14). Ce résultat n'est pas très surprenant : une donnée décrite ci-dessus peut contenir beaucoup de cycles différents. En effet, pour tout mot du langage ((`adore+ aime+ aimePas+ NonFini`*)(*`emprunté`*+ `réservé`*))* on peut construire une donnée contenant un cycle étiqueté par ce mot. Par exemple, l'une des données de la figure 4.13 a un cycle basé sur le chemin* `emprunté.adore` *alors que la deuxième donnée a un cycle basé sur le chemin* `emprunté.aime`*. Les nœuds 6 et 17 n'ont donc pas le même ensemble de chemins entrants et ils ne peuvent pas être fusionnés dans le 1-index.*

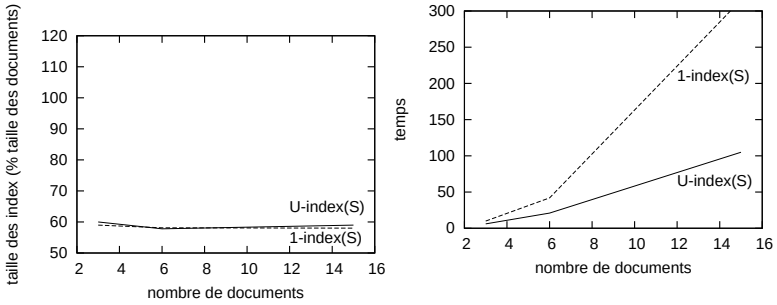


FIG. 4.14 – Expériences sur les données de type *bibliothèque*

Nos expériences ont montré que pour tout E_j, sous ensemble strict de E, la taille de U-index(E_j) est plus grande que la taille de 1-index(E). Il est donc plus rapide d'interroger E_j en indexant chaque donnée séparément que d'indexer toutes les données de E ensemble (cf. figure 4.14).

4.5 Conclusion

On a prouvé que la taille du 1-index est toujours plus petite que celle du U-index. Nos expériences ont montré que l'on peut faire un gain en espace et en temps très intéressant si le 1-index atteint une taille constante. Même si on considère un ensemble dans lequel toutes les données se ressemblent, cette taille limite peut ne pas exister. Dans ce cas le 1-index et le U-index se ressemblent, ont une taille similaire et le U-index est plus adapté pour interroger un sous-ensemble de l'ensemble considéré.

Le but de cette section était d'établir le lien entre la notion d'index et les contraintes d'inclusions satisfaites par un ensemble de données. Rechercher les contraintes communes à plusieurs données étant équivalent à rechercher les contraintes satisfaites par une donnée, nous avons utilisé deux types d'index existants : le dataguide et le 1-index. Le dataguide nous a permis d'extraire les contraintes satisfaites par une donnée (par un ensemble de données). Le 1-index permet d'indexer les chemins d'une donnée (d'un ensemble de données) tout en préservant toutes les contraintes satisfaites par cette (ces) donnée(s). Il est une représentation compacte de la donnée initiale.

Nous avons constaté, expérimentalement, qu'indexer un ensemble de données en le considérant comme une seule donnée est à double tranchant. On peut effet considérer un ensemble de données très ressemblantes (i.e. ayant des 1-index similaires) et ainsi avoir une représentation très compacte de l'ensemble. Par contre, on peut aussi construire des ensembles pour lesquels ce phénomène ne se produit pas. Dans ce cas, il est plus efficace d'indexer (avec un 1-index) chaque donnée séparément que d'indexer tout l'ensemble (ce point est particulièrement mis en valeur si on interroge non pas tout l'ensemble mais uniquement une sous-partie).

Trois grandes pistes peuvent être étudiées dans le futur.

1. Il serait très intéressant de caractériser des ensembles de données pour lesquels le 1-index est plus adapté que le U-index. Ce problème semble difficile à résoudre dans sa globalité. Un objectif intéressant serait donc de trouver des heuristiques pour guider ce choix.

2. Une autre piste serait de trouver des algorithmes efficaces de mise à jour d'un index lorsqu'une donnée est ajoutée. On pourrait utiliser un tel algorithme pour définir un indice de similarité d'une donnée par rapport à un ensemble de données. Une donnée D serait similaire aux données présentes dans E si l'index de l'ensemble $E \cup \{D\}$ est 'similaire' à l'index de E.
3. Enfin, on peut essayer d'utiliser le 1-index (ou le U-index) comme guide commun à plusieurs données. Pour l'instant, le point de vue des requêtes est dominant. On pourrait définir des mesures de similarité entre plusieurs données en utilisant des index. L'idée naturelle est de se baser sur le rapport entre la taille du 1-index (ou du U-index) et la somme des tailles des données présentes dans l'ensemble.

Chapitre 5

Conclusion

L'objectif de cette thèse était d'exploiter la structure de graphe pour modéliser et interroger des données semi-structurées. Deux grandes familles de documents ont été étudiées : les documents XML coloré et les graphes orientés étiquetés à plusieurs racines.

Les arbres XML sont un premier exemple de données semi-structurées. On peut enrichir les arbres XML en considérant les références entre deux noeuds de l'arbre. Dans ce cas, chaque arc du graphe porte soit l'étiquette *arbre* soit l'étiquette *référence*. L'étape suivante consiste à considérer plus que deux étiquettes pour les arcs et à colorier les documents XML.

XML n'est pas le seul standard pour les données semi-structurées. On peut donc utiliser un modèle basé sur les graphes orientés étiquetés à plusieurs racines dans lequel il n'y a pas de structure d'arbre sous-jacente. On peut, par exemple, penser à des données issues des bases de données orientées objets ou aux sites Web. Plusieurs sous-catégories sont possibles : on peut enraciner la donnée, imposer que le graphe soit déterministe....

Deux points sont communs à toutes les données utilisées. Premièrement, seule la structure du document est exploitée : on ne modélise que les nœuds, les arcs et leurs étiquettes. En particulier, on ne tire pas profit du contenu des noeuds (i.e. en XML, le texte qui se trouve entre deux balises). Deuxièmement, les données ne se réfèrent à aucun schéma explicite (i.e. on a aucune information sur le type des noeuds, on ne connaît pas la DTD des documents XML, on ne connaît pas les clefs vérifiées par un document ...). Cette approche ne permet pas d'utiliser les techniques classiques du modèle relationnel, dans lequel la connaissance préalable d'un schéma est primordiale.

Le fil rouge de cette thèse est la notion de requêtes. Les requêtes régulières permettent de parcourir les nœuds d'un graphe en suivant ses chemins alors qu'une requête graphe se propose de naviguer dans un document XML coloré en utilisant des axes de navigation dans les arbres ordonnés.

Rappelons maintenant quelques résultats importants montrés dans les pages précédentes :

- Les requêtes graphes ont été introduites et ont été utilisées dans le cadre des documents XML coloré. Par rapport au langage existant, cette nouvelle classe de requête permet d'exprimer l'égalité entre deux noeuds. On a montré que, dans ce cadre, l'évaluation d'une requête est NP-complète.

- On peut essayer d'optimiser une requête régulière en utilisant des contraintes d'inclusion. Dans cette optique, le problème de l'équivalent fini a été introduit : peut-on transformer une requête q en une requête fini q_f équivalente à q par rapport à un ensemble de contraintes ? La décidabilité du problème a été montrée pour des contraintes bornées grâce à une réduction du problème de l'implication de contraintes à un problème de réécriture. Dans le cas particulier des contraintes de mots, nous avons montré que le problème est PTIME. Enfin, un outil de réécriture de requêtes a été développé pour valider ce résultat avec un point de vue optimisation de requêtes.
- Cet outil a aussi permis de comparer différents algorithmes d'indexation de chemins. Dans le but de trouver des contraintes d'inclusion satisfaites par plusieurs documents, on s'est intéressé au lien qui existe entre la structure d'une donnée et celle de son index. On a notamment prouvé, par l'intermédiaire des dataguides, que toute donnée est un modèle exact fini d'un ensemble fini de contraintes finies et que le 1-index est une représentation compacte des contraintes satisfaites par une donnée. Pour terminer, ces résultats ont été étendus à l'étude des contraintes d'inclusions satisfaites par un ensemble de données semi-structurées.

Perspectives

Quelques travaux peuvent être envisagés pour compléter les résultats présentés dans ce mémoire. La première piste est d'utiliser une forme de contraintes d'inclusion pour optimiser les requêtes graphes. Dans cette optique, deux questions me semblent intéressantes. La première est d'identifier un fragment des requêtes graphes compatibles avec la notion de chemins utilisée par les contraintes d'inclusion. Une idée simple est de restreindre les axes à *child*, *descendant*, *idref*. Deuxièmement, il faut préciser la notion de contraintes d'inclusion dans le contexte d'XML. L'idée qui semble le mieux correspondre est celle de *clef*.

Une autre perspective naturelle concerne les vues. On peut imaginer réécrire une requête par rapport à un ensemble de vues en utilisant des techniques semblables à celles que nous utilisons pour les contraintes d'inclusion. Le but serait de trouver la meilleure réécriture possible i.e. celle qui maximise la taille de la (des) sous-requête(s) transformée(s) en vue. On peut se référer à [Grahne and Thomo, 2001] ou à [Xu and Ozsoyoglu, 2005] pour trouver une définition plus précise du problème.

Une perspective à plus long terme est l'intégration de plusieurs données. En effet les requêtes graphes et les méthodes d'indexation étudiées ne se limitent pas à un unique document. Comment peut-on utiliser les méthodes d'indexation pour trouver un schéma commun à plusieurs données ? Les méthodes d'indexation seraient-elles mieux adaptées pour découvrir les points communs et les différences entre deux documents (entre deux schémas) ? A quoi doit ressembler un schéma pour un document XML coloré (compatible avec les requêtes graphes) ?

Chapitre 6

Nouveau chapitre de la thèse

Cadre général et enjeux de ma thèse

Depuis quelques années, le nombre d'informations présentes sur internet ne cesse d'augmenter. Outre cette augmentation quantitative, on constate une diversité de plus en plus grande des données accessibles sur un site web (commerce électronique, texte, jeux, forum,...). Je ne parle même pas des différents formats de ces données (texte, image, son,...) et des problèmes de traduction d'une langue vers une autre.

C'est dans ce contexte que le projet Mostrare de l'I.N.R.I.A. (institut de recherche en informatique) est né à Lille en 2003 avec la collaboration de deux équipes du CNRS. Le but est de proposer des idées pour extraire automatiquement de l'information présente sur le Web : par exemple, savoir extraire le prix d'un produit d'un concurrent (ou d'un sous-traitant) depuis son site, permet de simplifier les tâches de veille et d'avoir une réactivité plus forte.

Nous travaillons aussi en collaboration avec deux groupes de travail financés par le ministère (ACI). Ces groupes permettent à différents laboratoires français de travailler ensemble. Pour finir, l'équipe Mostrare, travaille en collaboration avec des chercheurs étrangers, notamment en Autriche.

Les vingt cinq personnes (environ) qui travaillent dans Mostrare ont des profils très différents. Il y a, en plus des thésards, des permanents de l'I.N.R.I.A., des enseignants-chercheurs, des ingénieurs, des assistantes de projets...

Mostrare est donc un projet jeune, travaillant sur des problèmes très actuels. J'apporte ma contribution à cet élan en étudiant plus particulièrement :

Modélisation et requêtes des documents semi-structurés : exploitation de la structure de graphe

L'un des mes objectifs est de simplifier la recherche d'information sur des sites web. Par exemple, il est fréquent de passer plusieurs fois par la page d'accueil d'un site alors que des liens permettent de *court-circuiter* ces retours en arrière. Plus généralement, mon travail consiste à définir des outils pour interroger des documents type *site web*. Il consiste aussi à optimiser le temps de recherche en évitant des redondances.

A ce stade de ma présentation, deux grands enjeux peuvent être identifiés :

1. Donner vie à Mostrare : il s'agit de mettre en place le fonctionnement interne de l'équipe (répartition des pistes à étudier, organisation des séminaires internes...) mais aussi de nous faire connaître à l'extérieur (pour créer des groupes de travail inter-université, pour nouer des contacts avec des entreprises...). Il va sans dire que l'I.N.R.I.A., le CNRS ou l'université attendent de nous un maximum de bonnes publications.
2. Le second enjeu est plus économique : dans le domaine des technologies du web, les entreprises ont négligé la recherche fondamentale. Notre travail peut permettre de valider ou de réfuter des choix faits (depuis plusieurs années) par ces entreprises. Notons, qu'un tel cas de figure s'est déjà produit : une équipe de chercheurs autrichienne a mis en défaut l'algorithme choisi par Microsoft (et d'autres) pour évaluer une requête Xpath. C'est pourquoi, et c'est une volonté très forte de l'I.N.R.I.A, des liens se créent entre Mostrare et des entreprises (par exemple Xerox ou Lixto) afin de lier la recherche fondamentale et ses applications industrielles.

Déroulement, gestion et coût de mon projet

Tout d'abord, je vais expliquer pourquoi j'ai choisi de faire une thèse. La première raison est la volonté de devenir enseignant. Alors que mon projet initial était l'enseignement dans le secondaire, j'ai changé d'avis après avoir rencontré plusieurs professeurs qui m'ont dit que l'enseignement en lycée se réduisait souvent à des *recettes* pour le bac et qu'il était difficile de faire autrement. Je me suis alors dirigé vers la voie universitaire. La seconde raison est la notion de *challenge* : challenge d'étudier des problèmes que personne n'a étudiés avant nous et même plus simplement, challenge de répondre à des questions dont la réponse n'est pas simplement dans notre cours. Il faut donc inventer son propre chemin pour trouver les solutions.

Il faut donc franchir le pas qui va d'un système scolaire (cours, exercices sanctionnés par un examen) vers la recherche effectuée de manière autonome. Le DEA sert à nous évaluer (pour l'obtention des postes de thésards) et à nous initier à la recherche pendant un stage de cinq mois. La thèse est donc à la fois une formation (formation à la recherche par la recherche) et une première expérience non scolaire.

Je peux identifier trois grandes phases dans ma thèse. Le passage d'une phase vers la suivante s'est effectué progressivement, dans la continuité et après discussion avec toutes les personnes qui travaillent avec moi.

Le début d'une nouvelle phase n'a jamais signifié une rupture avec la (les) phase(s) précédente(s). Je suis, au contraire, revenu chaque fois que nécessaire dans des travaux antérieurs afin de les terminer et de les valoriser au mieux.

J'ai proposé ces changements de phases afin d'élargir mes connaissances mais aussi pour m'approprier mon sujet et pour suivre mes pistes de manière de plus en plus autonome.

Phase 1 :

J'ai découvert une partie du travail de chercheur : apprendre à lire un article, apprendre à rédiger un papier, apprendre à exposer...Mais j'ai aussi découvert le domaine dans lequel j'allais travailler et les travaux déjà existants.

Lors de réunions hebdomadaires, j'exposai mon travail aux critiques de mes encadrants. Ils me firent remarquer mes erreurs et me donnèrent des conseils pour avoir des présentations plus synthétiques et plus faciles à comprendre pour une tierce personne.

J'ai beaucoup progressé pendant cette phase. Je suis en effet passé de résultats (presque toujours) faux, à des résultats justes mais mal démontrés et enfin à des démonstrations correctes. J'ai acquis ces compétences grâce aux connaissances et à l'expérience de mes encadrants.

J'ai tiré un enseignement de cette période : je suis beaucoup plus à l'aise à l'oral qu'à l'écrit pour présenter mes travaux, mes idées... Dorénavant, j'ai remplacé mes synthèses hebdomadaires écrites par des exposés (éventuellement, j'utilise quelques transparents).

Phase 2 :

Peu à peu, en parallèle à ce travail théorique, j'ai eu envie d'étudier des cas pratiques que le travail théorique m'avais permis d'identifier.

Nos réunions sont toujours hebdomadaires mais, désormais, nous travaillons avec des étudiants (stagiaires). Dans cette phase j'ai commencé à être suffisamment indépendant pour être force de proposition et pour aider les stagiaires.

J'ai du faire faire face à un nouveau problème : concevoir et réaliser une application (des outils) ce que personne dans le groupe n'avait l'habitude de faire. La conception a été source de beaucoup de débats puisque nous n'étions pas d'accord sur le point de vue à adopter. Pour clore ce débat, j'ai choisi une voie intermédiaire (qui convenait aussi à l'étudiant qui travaillait avec nous). En ce qui concerne la réalisation, je me suis formé à Ocaml, langage de programmation que je n'avais jamais utilisé.

Une fois les outils terminés, un nouveau problème s'est présenté : l'analyse des résultats. Les résultats proposés par nos outils étaient-ils bons ? Nos outils étaient-ils plus efficaces que les outils proposés par les autres ? Pour répondre à cette problématique, nous avons testé nos outils sur des benchmarks reconnus par la communauté, mais aussi proposé nos propres documents afin de bien montrer les spécificités de notre étude.

Pendant ces quelques mois, je me suis rendu compte que j'aime réaliser des applications concrètes et pas uniquement des procédés théoriques. En effet, il m'est plus facile de savoir où j'en suis.

Phase 3 :

Nos réunions hebdomadaires furent maintenues mais l'ordre du jour ne porta plus forcément sur mon travail de la semaine. En fait, nos réunions furent consacrées au bilan du travail effectué, à la rédaction d'articles de synthèses (sur les travaux des deux premières phases).

Parallèlement à cela, j'ai suivi mes pistes de recherche que je développai avec mon point de vue, mes connaissances, mes envies... J'ai présenté de temps en temps mon travail afin d'avoir un miroir de ce que je faisais et pour avoir un retour de personnes extérieures.

J'ai donc, pour la première fois, mené mon idée du début à la fin sans attendre d'aide extérieure. Malheureusement, j'ai fait une faute de raisonnement qui a entâché la qualité de mes résultats. Néanmoins, Cette expérience fut très enrichissante car :

- Elle m'a permis de prendre des initiatives et d'avoir une vision globale de la rédaction d'un article (bien expliquer le contexte, mettre en avant les applications...). En ce sens, je me suis approprié mon sujet.

- Elle a mis en évidence l'un de mes axes de progression : il faut que j'apprenne à prendre de la distance avec mes idées. Une solution, que j'expérimente pour la rédaction de ma thèse, consiste à faire *autre chose* pendant quelques jours puis à revenir sur mon idée avec un regard neuf.
- Elle m'a appris à relativiser et à savoir rebondir après une déception (i.e. apprendre à bien identifier où est le problème, quelles sont les parties du travail qui ne sont pas affectées, que valent les nouveaux résultats...)

Comme je l'ai montré, j'ai eu la souplesse dans la gestion de ma thèse. Malheureusement, il me semble que les objectifs à atteindre n'ont jamais été très bien définis. A condition de ne pas s'enfermer dans ses objectifs, je pense qu'avoir une (des) cible(s) à atteindre (dans un délai déterminé) aurait été pour moi une source de motivation et un bon moyen de m'évaluer. Pour résumer, je regrette de n'avoir jamais réussi à évaluer la rentabilité de mon travail.

Tout ce travail a un coût dont je donne les grandes lignes dans le tableau suivant :

Dépense	Somme	
Salaire (charges comprises)	**108 500**	
• Salaire personnel	60 000	université
• Environnement (secrétariat, séminaire Mostrare, gestion des bourses de thèse...)	36 000	université, CNRS, INRIA
• Réunion de travail (600h)	30 000	université
dont 150h pour ma thèse	7 500	université
• encadrement individuel (100h)	5 000	université
Déplacements	**6 000**	université, CNRS, INRIA
Formations personnelles	**2 100**	
• Doctoriales	1 000	A.B.G., ministère de la recherche, Europe
• N.C.T.	600	A.B.G..
• Formation à l'enseignement	300	C.I.E.S.
• École doctorale	200	
Total	**116 600**	

Compétences, savoir-faire, qualités professionnelles et personnelles

- Révéler mon sens critique : avant même le début de ma thèse, mon entourage s'étonnait souvent de la rapidité et de la pertinence de mes critiques vis-à-vis de l'actualité. Cette qualité s'est révélée très importante pendant ma thèse même si j'ai appris à l'utiliser à bon escient. Dans mon rapport de D.E.A., j'ai en effet critiqué (négativement) les résultats d'un spécialiste de mon domaine. Mes encadrants m'ont dit que ce n'était pas raisonnable et qu'en temps que stagiaire je n'avais ni les compétences ni une vision assez globale du problème pour le faire. A l'opposé, au début de ma thèse, je m'intéressais surtout aux définitions et à la découverte de mon domaine et je prenais les nouveaux résultats *pour argent comptant* sans prendre le temps de les analyser. Peu à peu, j'ai appris à critiquer ces travaux et à évaluer les points intéressants pour ma thèse. A la fin de ma thèse, on m'a proposé d'être membre du comité de lecture d'une conférence (i.e. évaluer les papiers soumis, pour accepter | refuser leur publication).

- Être honnête avec moi-même : que ce soit pour écrire une démonstration ou pour analyser un résultat, il faut être honnête avec soi-même. J'ai eu a plusieurs reprises la tentation de dire *le résultat est bon*, ou bien *mon programme fonctionne* pour faire avancer mon travail sans prendre en compte tous les paramètres. Il est très facile de sur-évaluer son travail pensant que l'on fait beaucoup mieux que les autres. Il ne faut jamais oublier de se poser la question : "pourquoi personne n'a eu cette idée avant moi ?"
- Connaître mes limites : je sais reconnaître que je ne sais pas résoudre un problème mais je sais m'adapter en conséquence. Dans ce cas, deux réponses sont possibles : soit je fais appel à de l'aide extérieure (par exemple pour une correction orthographique), soit je redéfinis le cadre de mon travail en fonction de mes compétences.
- Sens de l'accueil : les nouveaux venus dans notre équipe se souviennent tous que, le jour de leur arrivée, c'est moi qui me suis assuré qu'ils étaient bien installés et qui les ai mis au courant des habitudes du groupe (heure et lieu des repas par exemple).
- Exposer un travail scientifique : le problème majeur est de savoir adapter son discours au public concerné. Lors d'une rencontre avec mes encadrants, je dois *tout expliquer* (en donnant, par exemple, les détails techniques) alors que, lors d'un séminaire, seules les grandes idées doivent être présentées. Dans le cas d'un séminaire, la gestion du temps est également importante : j'ai appris à être précis dans le choix de mes mots afin de faire comprendre, dans un intervalle de temps assez court, des notions complexes. Mes activités en tant qu'enseignant ont renforcé cette compétence.
- La manipulation de la notion de graphe : c'est la compétence technique qui est au coeur de ma thèse. Pendant mes trois phases de travail, toutes les études que j'ai menées étaient liées aux modèles de graphes. Je suis donc un expert dans ce domaine : expert dans le sens où j'ai acquis beaucoup de techniques et de méthodes de résolutions de problèmes sur les graphes. Mais aussi expert dans le sens où j'ai acquis une vision globale du domaine. J'ai donc une capacité d'analyse de ces problèmes qui associe compétences techniques, intuition et discernement.
- Aller de la théorie à la pratique : le début de ma thèse fut très théorique (proche des mathématiques). Mon étude a mis en évidence la complexité (algorithmique) des problèmes que j'étudie. Néanmoins, j'ai eu envie de restreindre le cadre pour étudier des cas réels pour lesquels l'étude théorique était prometteuse. Pour cela, je me suis formé à deux reprises à de *nouvelles compétences*. D'une part, j'ai appris à programmer en Ocaml pour développer des outils sur les données semi-structurées. D'autre part, j'ai appris à manipuler des langages pour interroger des données (Xpath et Xquery). Dans les deux cas je me suis formé, en une dizaine de jours, à partir de livres ou de tutoriaux trouvés sur le Web. J'ai, maintenant, les compétences nécessaires pour donner un cours concernant ces deux concepts.
- Persévérer : travailler pendant trois ans sur son sujet de thèse demande de la persévérance, de la volonté et une très grande motivation. J'ai par exemple voulu résoudre un problème pour lequel toutes les solutions que j'ai envisagées pendant plusieurs mois ne me convenaient pas. Pourtant, j'ai fini par le résoudre. En effet, en suivant un exposé dans un autre domaine, l'idée m'est venue de reprendre la démarche présentée. Dés lors, je n'ai eu besoin que de quelques jours pour terminer mon étude.
- Sens de l'organisation : les séminaires Mostrare visent à la fois à présenter nos travaux en interne mais aussi à inviter des personnes extérieures. J'ai donc organisé l'un de ces séminaires du début à la fin : invitation des personnes, organisation de leur voyage, gestion de la journée et du planning de travail. Deux expériences dans le monde associatif m'ont

alors beaucoup aidé. J'ai été trésorier de l'AEI (bureau des étudiants en informatique) et j'ai, dans ce cadre, organisé plusieurs évènements ponctuels. Par ailleurs, je suis membre actif de l'association Concordia qui organise des chantiers internationaux de bénévoles. A ce titre, j'ai déjà été amené à organiser (et participer à) des chantiers mais aussi à des évènements tels que des forums aux associations, des week-ends de formation...

Résultats, impact de ma thèse

Le projet Mostrare étant un projet jeune, les problématiques abordées étant très nombreuses, j'ai eu une grande liberté pour gérer mon sujet. Trois ans plus tard, nous avons fait le constat qu'existe une trop grande dispersion dans Mostrare et qu'il faut recentrer nos sujets de recherche. Malgré l'identification de nombreux problèmes (intéressants) à résoudre, ma thèse n'aura donc pas de suite (pas de nouveau stagiaire en DEA) à Lille.

La grande diversité des travaux effectués au sein de Mostrare a eu pour moi un impact positif. J'ai acquis les connaissances et les méthodes de base pour toutes les problématiques présentes dans Mostrare (logique, apprentissage automatique, langage naturel,...) : j'ai participé à des séminaires ou à des réunions qui n'étaient pas toujours directement liés à mon travail mais qui m'ont permis de ne pas rester enfermé dans mon sujet (et donc de découvrir d'autres choses).

Un autre impact positif de ma participation dans l'équipe Mostrare est que j'ai travaillé sur des sujets très actuels. J'ai donc, dans la partie plus appliquée de ma thèse, pris en main certaines technologies de pointe (notamment liées à XML). Ces technologies sont de plus en plus utilisées en entreprise. Je peux donc mettre en valeur le fait de connaître ces outils, mais aussi d'avoir le backround théorique pour comprendre leur fonctionnement.

Pour conclure, je voudrais exposer la principale piste d'insertion professionnelle que j'ai identifiée : il s'agirait de créer une université d'entreprise dans mes compétences techniques (ou un équivalent dans la fonction publique). Cette activité me permettrait :

- de créer de toute pièce, un nouveau *département* dans une entreprise. De plus, le but de cette création est précis et bien défini tout en laissant une grande liberté quant à la forme et au contenu de cette université.
- de travailler en équipe, d'échanger des idées avec toutes les parties prenantes au projet (directeur, ressources humaines, chefs de service...), et de prendre en charge l'organisation pratique de ce projet. De plus à ce poste je pourrais être en charge de l'accueil des nouveaux arrivants dans l'entreprise.
- d'utiliser mes expériences : création d'association, gestion d'un budget, la thèse (innovation, communication, gestion de projet, travail en équipe...) et compétence pédagogique.

Cette piste correspond à un projet à long terme. Pour atteindre cet objectif, je vais continuer à faire de l'enseignement et de la recherche. Je pourrais ainsi accroître mes compétences pédagogiques et continuer à progresser dans la communication écrite (compte-rendu de réunion...). Une autre étape pourrait être de prendre des responsabilités pédagogiques ou administratives en université.

Bibliographie

[Abiteboul et al., 2003] Abiteboul, S., Baumgarten, J., Bonifati, A., Cobena, G., Cremarenco, C., Dragan, F., Manolescu, I., Milo, T., and Preda, N. (2003). Managing distributed workspaces with active xml. In *VLDB*, pages 1061–1064.

[Abiteboul et al., 2000] Abiteboul, S., Buneman, P., and Suciu, D. (2000). *Data on the Web.* Morgan Kaufmann Publishers.

[Abiteboul et al., 1995] Abiteboul, S., Hull, R., and Vianu, V. (1995). *Foundations of Databases.* Addison-Wesley.

[Abiteboul et al., 1997] Abiteboul, S., Quass, D., McHugh, J., Widom, J., and Wiener, J. (1997). The lorel query language for semistructured data. *Journal of Digital Libraries*, 1(1) :68–88.

[Abiteboul and Vianu, 1997] Abiteboul, S. and Vianu, V. (1997). Regular path queries with constraints. In *PODS*, pages 122–133. ACM Press.

[Abiteboul and Vianu, 1999] Abiteboul, S. and Vianu, V. (1999). Regular path queries with constraints. *J. Comput. Syst. Sci.*, 58(3) :428–452.

[Abrão et al., 2004] Abrão, M. A., Bouchou, B., Halfeld Ferrari, M., Laurent, D., and Musicante, M. A. (2004). Incremental constraint checking for XML documents. In *XSym*, number 3186 in LNCS, pages 112–127.

[Aho et al., 1974] Aho, A., Hopcroft, J., and Ullman, J. (1974). *The design and Analysis of Computer Algorithms.* Addison-Wesley Publishing Compagny, Reading, Mass.

[Al-Khalifa et al., 2002] Al-Khalifa, S., Jagadish, H. V., Patel, J. M., Wu, Y., Koudas, N., and Srivastava, D. (2002). Structural joins : A primitive for efficient xml query pattern matching. In *ICDE '02 : Proceedings of the 18th International Conference on Data Engineering (ICDE'02)*, pages 141–153. IEEE Computer Society.

[Alechina et al., 2003] Alechina, N., Demri, S., and de Rijke, M. (2003). A modal perspective on path constraints. *Journal of Logic and Computation*, 13(6) :939 – 956.

[Alur and Henzinger, 2004] Alur, R. and Henzinger, T. (2004). Computer-aided verification. chapter 4 : Graph minimization. Available on : http ://www.ce.sharif.edu/courses/81-82/1/ce665/.

[Amer-Yahia et al., 2001] Amer-Yahia, S., Cho, S., Lakshmanan, L. V. S., and Srivastava, D. (2001). Minimization of tree pattern queries. In *SIGMOD Conference.*

[Andre et al., 2004] Andre, Y., Caron, A., Debarbieux, D., Roos, Y., and Tison, S. (2004). Extraction and implication of path constraints. In *Proceedings of the 29th Symposium on Mathematical Foundations of Computer Science*, Lecture Notes in Computer Science, pages 863–875.

[Andre et al., 2005] Andre, Y., Caron, A.-C., Debarbieux, D., and Roos, Y. (2005). Indexes and path constraints in semistructured data. In *Workshop on Logical Aspects and Applications of Integrity Constraints (DEXA 2005)*, pages 837 – 841, Copenhagen, Denmark. IEEE - computer society.

[André et al., 1999] André, Y., Bossut, F., and Caron., A. (1999). On decidability of boundedness property for regular path queries. In *proceedings of DLT'99*, Aachen, Germany. Development in Language Theory, World Scientific Publishing Co.

[Arenas and Libkin, 2004] Arenas, M. and Libkin, L. (2004). A normal form for xml documents. *ACM Trans. Database Syst.*, 29(1) :195–232.

[Berglund et al., 2002a] Berglund, A., Boag, S., Chamberlin, D., Fernandez, M., Kay, M., Robie, J., and Siméon, J. (2002a). XML Path Language (XPath) 2.0, w3c recommentation. The W3C's Xpath web pages : `http://www.w3.org/TR/xpath/`.

[Berglund et al., 2002b] Berglund, A., Boag, S., Chamberlin, D., Fernandez, M., Kay, M., Robie, J., and Siméon, J. (2002b). XML Path Language (XPath) 2.0, w3c recommentation. The W3C's Xquery web pages : `http://www.w3.org/TR/xquery/`.

[Berstel, 1979] Berstel, J. (1979). *Transductions and Context-Free Languages.* Teubner Studienbücher, Stuttgart.

[Bertino et al., 2004] Bertino, E., Christodoulakis, S., Plexousakis, D., Christophides, V., Koubarakis, M., Böhm, K., and Ferrari, E., editors (2004). *Advances in Database Technology - EDBT 2004, 9th International Conference on Extending Database Technology, Heraklion, Crete, Greece, March 14-18, 2004, Proceedings*, volume 2992 of *Lecture Notes in Computer Science*. Springer.

[Bidoit et al., 2004] Bidoit, N., Cerrito, S., and Thion, V. (2004). A first step towards modelling semistructured data in hybrid modal logic. *Journal of Non Classical Logics.*

[Bloom and Paige, 1995] Bloom, B. and Paige, R. (1995). Transformational design and implementation of a new efficient solution to the ready simulation problem. *Sci. Comput. Program.*, 24(3) :189–220.

[Bouchou et al., 2003] Bouchou, B., Halfeld Ferrari Alves, M., and Musicante, M. A. (2003). Tree automata to verify xml key constraints. In *WebDB*, pages 37–42.

[Bray et al., 1998] Bray, T., Paoli, J., and Sperberg-McQueen, C. (1998). Extensible markup language (xml) 1.0. `http://www.w3.org/XML`.

[Bruno et al., 2002] Bruno, N., Koudas, N., and Srivastava, D. (2002). Holistic twig joins : optimal xml pattern matching. In *SIGMOD Conference*, pages 310–321.

[Bry et al., 2001] Bry, F., Olteanu, D., and Schaffert, S. (2001). Grouping Constructs for Semistructured Data. In *Proceedings of Workshop at Dexa'01, Munich, Germany (3rd September 2001).*

[Büchi, 1960] Büchi, J. (1960). On a decision method in restricted second order arithmetic. pages 1–11.

[Buneman et al., 2000a] Buneman, P., Davidson, S., Fan, W., Hara, C., and Tan, W. (2000a). Reasoning about keys for xml.

[Buneman et al., 1996] Buneman, P., Davidson, S., Hillebrand, G., and Suciu, D. (1996). A query language and optimization techniques for unstructured data. In *SIGMOD*, pages 505–516, Montreal.

[Buneman et al., 1997] Buneman, P., Davidson, S. B., Fernandez, M. F., and Suciu, D. (1997). Adding structure to unstructured data. In Afrati, F. N. and Kolaitis, P., editors, *Database Theory—ICDT'97, 6th International Conference*, volume 1186, pages 336–350, Delphi, Greece. Springer.

[Buneman et al., 1999] Buneman, P., Fan, W., and Weinstein, S. (1999). Query optimization for semistructured data using path constraints in a deterministic data model. In *Lecture Notes in Computer Science 1949*, pages 208–223. 7th International Workshop on Database Programming Languages.

[Buneman et al., 2000b] Buneman, P., Fan, W., and Weinstein, S. (2000b). Path constraints in semistructured databases. *Journal of Computer and System Sciences*, 61(2).

[Büchi and Hosken, 1970] Büchi, J. R. and Hosken, W. (1970). Canonical systems which produce periodic sets. *Mathematical Systems Theory*, 4(1).

[Caron et al., 2003] Caron, A., Debarbieux, D., and Roos, Y. (2003). Modèles de données semi-structurées et contraintes d'inclusion. In *RSTI série RIA-ECA*, volume 17, pages 461–472, Lyon, France. Extraction et Gestion des Connaissances, Hermes.

[Caucal, 1990] Caucal, D. (1990). On the regular structure of prefix rewritings. In Springer, editor, *Selected papers of the conference on Fifteenth colloquium on trees in algebra and programming*, pages 87 – 102, Copenhagen, Denmark.

[Caucal, 1992] Caucal, D. (1992). Monadic theory of term rewritings. In *Proceedings, Seventh Annual IEEE Symposium on Logic in Computer Science*, volume 1232, pages 266 – 273. IEEE Computer Society Press.

[Chen et al., 2005] Chen, L., Gupta, A., and Kurul, M. E. (2005). Efficient algorithms for pattern matching on directed acyclic graphs. In *IEEE Int. Conf. on Data Engineering (ICDE)*, Tokyo, Japan.

[Chen et al., 2003] Chen, Q., Lim, A., and Ong, K. W. (2003). D(k)-index : an adaptive structural summary for graph-structured data. In *Proceedings of the 2003 ACM SIGMOD international conference on Management of data*, pages 134–144. ACM Press.

[Chidlovskii, 2002] Chidlovskii, B. (2002). Schema extraction from xml collections. In *JCDL '02 : Proceedings of the 2nd ACM/IEEE-CS joint conference on Digital libraries*, pages 291–292. ACM Press.

[Comon et al., 1997] Comon, H., Dauchet, M., Gilleron, R., Jacquemard, F., Lugiez, D., Tison, S., and Tommasi, M. (1997). Tree automata techniques and applications. Available on : http ://www.grappa.univ-lille3.fr/tata.

[Cormen et al., 1990] Cormen, T., Leiserson, C., and R.Rivest (1990). *Introduction to Algorithms*. MIT Press.

[Dalzilio and Lugiez, 2003] Dalzilio, S. and Lugiez, D. (2003). Xml schemas, tree logic and sheave automata. In *RTA 03*.

[Dauchet and Tison, 1990] Dauchet, M. and Tison, S. (1990). The theory of ground rewrite systems is decidable. In *Proc 5th IEEE Symp Logic in Computer Science*, pages 242–256.

[De Giacomo, 1995] De Giacomo, G. (1995). *Decidability of Class-Based Knowledge Representation Formalisms*. PhD thesis, Italia.

[Debarbieux, 2004] Debarbieux, D. (2004). Données semi-structurées et contraintes de chemin. In *MAJECSTIC'04*.

[Debarbieux and Luchier, 2004] Debarbieux, D. and Luchier, J. (2004). Qric : Query rewriting with inclusion constraints. www.lifl.fr/~debarbie/QRIC/.

[Debarbieux et al., 2004] Debarbieux, D., Roos, Y., and Tison, S. (2004). Models of path constraints. In *Proceedings of the 10th Mons Theoretical Computer Science Days*.

[Debarbieux et al., 2003] Debarbieux, D., Roos, Y., Tison, S., Andre, Y., and Caron, A. (2003). Path rewriting in semistructured data. In *proceedings of words'03 : 4th International Conference on Combinatorics on Words*, pages 358–369, Turkü, Finland. TUCS General Publication.

[Denis et al., 2001] Denis, F., Lemay, A., and Terlutte, A. (2001). Residual finite state automata. In *Lecture Notes in Computer Science 2001*, pages 144–157. 18th Annual Symposium on Theoretical Aspects of Computer Science.

[Deutsch et al., 1999] Deutsch, A., Fernandez, M., Florescu, D., Levy, A., and Suciu, D. (1999). A query language for xml. In *Proc. of World Wide Web Conference*, Toronto.

[eXist, 2000] eXist (2000). exist : Open source xml database. `http://exist-db. org/`.

[Fagin, 1983] Fagin, R. (1983). Degrees of acyclicity for hypergraphs and relational database schemes. *Journal of the ACM*, 30(3) :514 – 550.

[Fan and Simeon, 2000] Fan, W. and Simeon, J. (2000). Integrity constraints for XML. In *Symposium on Principles of Database Systems*, pages 23–34.

[Fernandez, 1990] Fernandez, J.-C. (1990). An implementation of an efficient algorithm for bisimulation equivalence. *Sci. Comput. Program.*, 13(2-3) :219–236.

[Fernau, 2001] Fernau, H. (2001). Learning xml grammars. In *MLDM '01 : Proceedings of the Second International Workshop on Machine Learning and Data Mining in Pattern Recognition*, pages 73–87. Springer-Verlag.

[Gardarin and Yeh, 2005] Gardarin, G. and Yeh, L. (2005). Sioux : An efficient index for processing strucutural xquery. In *DEXA'05 : Proceedings of the 16th International conference on Database and Expert Systems Applications*, pages 564 – 575. Springer Verlag.

[Garey and Johnson, 1978] Garey, M. and Johnson, D. (1978). *Computers and Intractability : A guide to the Theory of NP-completeness*. Freeman.

[Gluskov, 1961] Gluskov, V. (1961). the abstract theory of automata. In *Russian mathematical survey*, volume 16, pages 1–53.

[Goldman and Widom, 1997] Goldman, R. and Widom, J. (1997). Dataguides : Enabling query formulation and optimization in semistructured databases. In *Proc. of International Conf. on Very Large Data Bases*, pages 436–445.

[Goldman and Widom, 1999] Goldman, R. and Widom, J. (1999). Approximate dataguides. In *Proceedings of the Workshop on Query Processing for Semistructured Data and Non-Standard Data Formats*, Jerusalem, Israel.

[Gottlob et al., 2002] Gottlob, G., Koch, C., and Pichler, R. (2002). Efficient algorithms for processing xpath queries. In *Proc. of the 28th International Conference on Very Large Data Bases (VLDB'02)*, Hong Kong.

[Gottlob et al., 2003] Gottlob, G., Koch, C., and Pichler, R. (2003). The complexity of xpath query evaluation. In *Proceedings of the twenty-second ACM SIGMOD-SIGACT-SIGART symposium on Principles of database systems*, pages 179–190. ACM Press.

[Gottlob et al., 2004] Gottlob, G., Koch, C., and Schulz, K. (2004). Conjunctive queries over trees. In *PODS*, pages 189–200.

[Gottlob et al., 1998] Gottlob, G., Leone, N., and Scarcello, F. (1998). The complexity of acyclic conjunctive queries. In *FOCS '98 : Proceedings of the 39th Annual Symposium on Foundations of Computer Science*, page 706, Washington, DC, USA. IEEE Computer Society.

[Gottlob et al., 2001] Gottlob, G., Leone, N., and Scarcello, F. (2001). The complexity of acyclic conjunctive queries. *J. ACM*, 48(3) :431–498.

[Graham, 1979] Graham, W. (1979). On the universal relation. In *Technical report Univ. Toronto.*

[Grahne and Thomo, 2001] Grahne, G. and Thomo, A. (2001). Algebraic rewritings for optimizing regular path queries. In den Bussche, J. V. and Vianu, V., editors, *Database Theory - ICDT 2001, 8th International Conference, London, UK, January 4-6, 2001, Proceedings*, volume 1973 of *Lecture Notes in Computer Science*, pages 301–315. Springer.

[Grahne and Thomo, 2003] Grahne, G. and Thomo, A. (2003). Query containment and rewriting using views for regular path queries under constraints. In *proceedings of PODS'03*, pages 111–122. Symposium on Principles of Database Systems, ACM.

[Henzinger et al., 1995] Henzinger, M. R., Henzinger, T. A., and Kopke, P. W. (1995). Computing simulations on finite and infinite graphs. In *IEEE Symposium on Foundations of Computer Science*, pages 453–462.

[Hopcroft and Ullman, 1979] Hopcroft, J. and Ullman, J. (1979). *Introduction to Automata Theory, Languages and Computation.* Addison-Wesley.

[Jagadish et al., 2004] Jagadish, H. V., Lakshmanan, L. V. S., Scannapieco, M., Srivastava, D., and Wiwatwattana, N. (2004). Colorful xml : one hierarchy isn't enough. In *SIGMOD '04 : Proceedings of the 2004 ACM SIGMOD international conference on Management of data*, pages 251–262. ACM Press.

[Jiang et al., 2004] Jiang, H., Lu, H., and Wang, W. (2004). Efficient processing of xml twig queries with or-predicates. In *SIGMOD '04 : Proceedings of the 2004 ACM SIGMOD international conference on Management of data*, pages 59–70, New York, NY, USA. ACM Press.

[Kaushik et al., 2002a] Kaushik, R., Bohannon, P., Naughton, J. F., and Korth, H. F. (2002a). Covering indexes for branching path queries. In *Proceedings of the 2002 ACM SIGMOD international conference on Management of data*, pages 133–144. ACM Press.

[Kaushik et al., 2002b] Kaushik, R., Shenoy, P., Bohannon, P., and Gudes, E. (2002b). Exploiting local similarity for efficient indexing of paths in graph structured data. In *ICDE.*

[Kepser, 2002] Kepser, S. (2002). A proof of the turing-completeness of xslt and xquery. In *Technical report SFB 441, Eberhard Karls Universitat Tubingen.*

[Kerbrat, 1994] Kerbrat, A. (1994). Méthodes symboliques pour la vérification de processus communicants : étude et mise en oeuvre. Thèse de doctorat soutenue à l'université Joseph Fourier- Grenoble 1.

[Klarlund et al., 2004] Klarlund, N., Schwentick, T., and Suciu, D. (2004). Xml : Model, schemas, types, logics and queries. In *Technical report.*

[Koch, 2005a] Koch, C. (2005a). On the complexity of nonrecursive xquery and functional query languages on complex values. In *PODS '05 : Proceedings of the twenty-fourth ACM SIGMOD-SIGACT-SIGART symposium on Principles of database systems*, pages 84–97, New York, NY, USA. ACM Press.

[Koch, 2005b] Koch, C. (2005b). On the role of composition in xquery. In *To appear in Proc. WebDB 2005.*

[Kripke, 1971] Kripke, S. (1971). *Semantic considerations on modal logic.* Oxford : Clarendon Press.

[Lore, 1997] Lore (1997). Lore. `http://www-db.stanford.edu/lore/`.

[Lu et al., 2005] Lu, J., Ling, T. W., Yu, T., Li, C., and Ni, W. (2005). Efficient processing of ordered xml twig pattern. In *DEXA'05 : Proceedings of the 16th International conference on Database and Expert Systems Applications*, pages 300 – 309. Springer Verlag.

[Maier, 1986] Maier, D. (1986). Computer science press.

[Marx, 2004] Marx, M. (2004). Xpath with conditional axis relations. In *EDBT*, pages 477–494.

[McHugh and Widom, 1999] McHugh, J. and Widom, J. (1999). Optimizing branching path expressions. Technical report, Stanford University.

[McHugh et al., 1998] McHugh, J., Widom, J., Abiteboul, S., Luo, Q., and Rajamaran, A. (1998). Indexing semistructured data. Technical report, Stanford University, Computer Science Department.

[Miklau and Suciu, 2002] Miklau, G. and Suciu, D. (2002). Containment and equivalence for an XPath fragment. In *Symposium on Principles of Databases Systems.*

[Milner, 1980] Milner, R. (1980). A calculus for communicating processes. In Verlab, S., editor, *LNCS*, volume 92.

[Milo and Suciu, 1999] Milo, T. and Suciu, D. (1999). Index structures for path expressions. In *ICDT '99 : Proceeding of the 7th International Conference on Database Theory*, pages 277–295, London, UK. Springer-Verlag.

[Nestorov et al., 1998] Nestorov, S., Abiteboul, S., and Motwani, R. (1998). Extracting schema from semistructured data. In *Proc. of ACM SIGMOD International Conference on Management of Data.*

[Nestorov et al., 1997] Nestorov, S., Ullman, J. D., Wiener, J. L., and Chawathe, S. S. (1997). Representative objects : Concise representations of semistructured, hierarchial data. In *ICDE*, pages 79–90.

[Neven and Schwentick, 2002] Neven, F. and Schwentick, T. (2002). Xpath containment in the presence of disjunction, dtds, and variables. In *ICDT '03 : Proceedings of the 9th International Conference on Database Theory*, pages 315–329, London, UK. Springer-Verlag.

[Paige and Tarjan, 1987] Paige, R. and Tarjan, R. E. (1987). Three partition refinement algorithms. *SIAM J. Comput.*, 16(6) :973–989.

[Park, 1981] Park, D. (1981). Concurency and automata on infinite sequences. In Verlab, S., editor, *5th GI-conf*, volume 104. LNCS.

[Ramanan, 2003] Ramanan, P. (2003). Covering indexes for xml queries : Bisimulation - simulation = negation. In *VLDB'03 : International conference on Very Large Data Base*, pages 165–176.

[Rogie et al., 1998] Rogie, J., L.Lapp, and Schach, D. (1998). Xml query manguage (xql). In *QL'98 - The query manguages Workshop*.

[Rys et al., 2005] Rys, M., Chamberlin, D., and Florescu, D. (2005). Xml and relational database management systems : the inside story. In *SIGMOD '05 : Proceedings of the 2005 ACM SIGMOD international conference on Management of data*, pages 945–947, New York, NY, USA. ACM Press.

[Schmidt et al., 2001] Schmidt, A. R., Waas, F., Kersten, M. L., Florescu, D., Manolescu, I., Carey, M. J., and Busse, R. (2001). The XML Benchmark Project. Technical Report INS-R0103, CWI, Amsterdam, The Netherlands.

[Schwentick, 2004] Schwentick, T. (2004). Xpath query containment. *SIGMOD Rec.*, 33(1) :101–109.

[Segoufin and Vianu, 2002] Segoufin, L. and Vianu, V. (2002). Validating streaming xml documents. In *PODS '02 : Proceedings of the twenty-first ACM SIGMOD-SIGACT-SIGART symposium on Principles of database systems*, pages 53–64, New York, NY, USA. ACM Press.

[Shostak, 1978] Shostak, R. (1978). An algorithm for reasoning about equality. *Commun. ACM*, 21(7) :583–585.

[Stockmeyer and Meyer, 1973] Stockmeyer, L. J. and Meyer, A. R. (1973). Word problems requiring exponential time : Preliminary report. In *STOC*, pages 1–9.

[Tarjan, 1975] Tarjan, R. (1975). Efficiency of a good but non linear set union algorithm. In *Journal of the ACM*, volume 22 of *2*, pages 215 – 225.

[Thompson et al., 2001] Thompson, H., Beech, D., Maloney, M., and Mendelsohn, N. (2001). Ml schema : Structures, w3c recommentation. The W3C's web pages : `http://www.w3.org/XML/Schema`.

[Tox, 2002] Tox (2002). Toxgene - the tox xml data generator. www.cs.toronto.edu/tox/toxgene/.

[Ullman, 1989] Ullman, J. D. (1989). *Principles of Database and Knowledge-Base Systems, Volume II*. Computer Science Press.

[Vagena et al., 2004] Vagena, Z., Moro, M. M., and Tsotras, V. J. (2004). Twig query processing over graph-structured xml data. In *Proceedings of the 7th International Workshop on the Web and Databases*, pages 43–48. ACM Press.

[Weigel et al., 2004] Weigel, F., Meuss, H., Bry, F., and Schulz, K. U. (2004). Content-aware dataguides : Interleaving ir and db indexing techniques for efficient retrieval of textual xml data. In *ECIR 04 : Advances in Information Retrieval, 26th European Conference on IR Research*, volume 2997 of *Lecture Notes in Computer Science*, pages 378–393. Springer.

[XMLbase, 2004] XMLbase (2004). Xml and databases. `www.rpbourret.com/xml/XMLAndDatabases.htm`.

[Xu and Ozsoyoglu, 2005] Xu, W. and Ozsoyoglu, M. (2005). Rewriting xpath queries using materialized views. In *VLDB'05 : 31 st International conference on Very Large Data Base.*

[Yamini and Gupta, 2005] Yamini, S. and Gupta, A. (2005). Spatiotemporal annotation graph (stag) : A data model for composite digital objects. In *Demonstration Track, IEEE Int. Conf. on Data Engineering (ICDE)*, Tokyo, Japan.

[Yi et al., 2004] Yi, K., He, H., Stanoi, I., and Yang, J. (2004). Incremental maintenance of xml structural indexes. In *Proceedings of the 2004 ACM SIGMOD international conference on Management of data*, pages 491–502. ACM Press.

[Yu and Ozsoyoglu, 1979] Yu, C. and Ozsoyoglu, M. (1979). An algorithm for tree query membership for a distributed query. In *IEEE COMPSAC*, pages 306–312.

Index

Modélisation et requêtes des documents semi-structurés : exploitation de la structure de graphe

La notion de données semi-structurées est liée au monde Web. On appelle donnée semi-structurée une donnée dont le schéma n'est pas défini a priori. Il peut s'agir d'une page HTML ou d'un site Web tout entier ou encore d'un document XML.

Cette thèse étudie les requêtes sur les données semi-structurées modélisées par des graphes. On s'intéresse à différentes représentations des données semi-structurées par des graphes et on considère différents langages de requêtes associés. Un problème différent est étudié pour chaque couple (représentation, langage).

Dans le cas des graphes orientés, on utilise des techniques de réécriture et d'automates pour étudier - à des fins d'optimisation de requêtes - les contraintes d'inclusions. Ces contraintes portent sur les chemins qui permettent de naviguer dans la donnée. Pour exploiter l'information liée à la structure d'une donnée, on génère un index qui préserve les contraintes d'inclusions.

On étend cette étude pour obtenir le concept de requête graphe. Son intérêt est de permettre la composition de requêtes et de définir celle-ci graphiquement. Appliquées au cas des documents "XML coloré", les requêtes graphes permettent d'étudier formellement l'expressivité et la complexité de langages de requêtes inspirés de XPath et de XQuery.

Les résultats théoriques sont validés par des expérimentations.