Induction de requêtes guidée par schéma

Jérôme Champavère

▶ To cite this version:

Jérôme Champavère. Induction de requêtes guidée par schéma. Autre [cs.OH]. Université des Sciences et Technologie de Lille - Lille I, 2010. Français. NNT : . tel-00517358

HAL Id: tel-00517358
https://theses.hal.science/tel-00517358

Submitted on 14 Sep 2010

HAL is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L'archive ouverte pluridisciplinaire **HAL**, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d'enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.

UNIVERSITÉ LILLE 1 – SCIENCES ET TECHNOLOGIES
LABORATOIRE D'INFORMATIQUE FONDAMENTALE DE LILLE

THÈSE

présentée en première version
en vue d'obtenir le grade de

Docteur en Informatique

par

Jérôme Champavère

Induction de requêtes guidée par schéma

(*Schema-Guided Query Induction*)

Thèse soutenue le 10 septembre 2010 devant le jury composé de :

M.	Jean-Marc CHAMPARNAUD	Université de Rouen	Rapporteur
M.	Colin DE LA HIGUERA	Université de Nantes	Rapporteur
M.	Henning FERNAU	Universität Trier	Examinateur
Mme	Françoise GIRE	Université de Paris 1	Examinateur
M.	Aurélien LEMAY	Université de Lille 3	Examinateur
M.	Marc TOMMASI	Université de Lille 3	Examinateur
M.	Joachim NIEHREN	Inria Lille – Nord Europe	Directeur

Version du 9 septembre 2010.

Mis en page et composé avec $\LaTeX 2_\varepsilon$.

Résumé

XML est un langage générique de description de données destiné à l'origine au stockage, au traitement et à l'échange d'informations sur Internet ; il s'agit aujourd'hui d'un format standard pour les communautés bases de données, documents ou technologies Web, qui est utilisé dans de nombreuses applications. Le format des données traitées par celles-ci est généralement spécifié par un schéma XML. Il s'agit d'une méta-description permettant de contraindre la structure et le type des données des documents XML qui le respectent.

Interroger les documents afin d'en extraire des informations est une tâche essentielle en informatique. Les requêtes de sélection de nœuds sont ainsi à la base de la transformation de documents XML. Cependant, la plupart des outils existants pour définir des requêtes sur les documents XML présupposent des connaissances techniques de la part de l'utilisateur. L'induction de requêtes supervisée est au contraire un moyen d'élaborer des tâches d'extraction d'information sans prérequis. Dans un tel système, une interface graphique permet à l'utilisateur d'annoter des documents qui servent d'exemples. Un algorithme d'apprentissage est alors utilisé pour inférer la requête.

Dans cette thèse, nous proposons d'utiliser les connaissances fournies par le schéma XML dans les algorithmes d'induction de requêtes basés sur une technique d'inférence grammaticale. En tant que langages réguliers d'arbres, les schémas peuvent être facilement représentés par des automates d'arbres. Leur utilisation dans des algorithmes d'inférence d'automates apparaît donc particulièrement appropriée. Nous en avons distingué deux.

1. La première idée est de contraindre la requête inférée à être consistante avec le schéma. Pour cela, nous avons mis au point un test d'inclusion efficace dans les automates d'arbres factorisés déterministes, un modèle d'automates permettant de représenter les DTD de façon compacte que nous avons introduit.
2. La seconde idée est que les informations contenues dans le schéma peuvent être précieuses pour élaguer les arbres correspondants à des documents annotés. L'élagage est nécessaire lorsque les documents traités sont gros et/ou annotés partiellement. En contrepartie, il n'est plus possible d'inférer toutes les requêtes régulières. Nous donnons une caractérisation de la classe de requêtes apprenables à partir d'un ensemble d'arbres annotés élagués, à savoir les requêtes stables.

Nous avons implémenté et testé nos algorithmes d'induction de requêtes guidée par schéma. Le système développé permet de simuler le comportement d'un utilisateur lors de la définition d'une nouvelle requête. Les résultats de nos expériences soutiennent la pertinence de notre approche. Ils montrent en effet que l'usage du schéma permet d'améliorer l'apprentissage.

Titre : Induction de requêtes guidée par schéma

Mots-clés : requêtes, schémas XML, inférence grammaticale, arbres, automates

Abstract

XML is a generic data description language originally designed for storing, processing and exchanging information on the Internet. It has raised as a standard for database, document or Web communities, and it is used in numerous applications nowadays. The data format processed by the latter is usually specified by an XML schema. This is a meta-description that constrains the structure of XML documents and their data type.

Querying documents in order to extract information they contain is an essential task. Node selecting queries are for instance the basis for transforming XML documents. However, most existing tools for defining queries over XML documents require technical skills from the user. In contrast, inductive query learning is a way of designing information extraction tasks without any prior knowledge. In such a system, the user annotates some example documents with a graphical interface. A learning algorithm is then used in order to infer the query.

In this thesis, we suggest to use the knowledge provided by XML schemas into query induction algorithms based on grammatical inference techniques. As regular tree languages, schemas can be easily represented by tree automata. Thus their use is especially appropriate to automata inference algorithms. We have distinguished two of them.

1. The first idea is to force inferred queries to be consistent with the schema. For this purpose, we have designed an efficient inclusion test in deterministic factorized tree automata, a model of automata we have defined in order to represent DTDs in a compact manner.
2. The second idea is that information contained in XML schemas might be useful for tree pruning heuristics. Pruning is necessary when processed documents are pretty large and/or partially annotated. The counterpart is that some regular queries cannot be inferred anymore. We give a characterization of the class of queries that can be learned from a sample of pruned annotated trees, namely stable queries.

We have implemented and tested our schema-guided query induction algorithms. The developed system enables to simulate the user behavior when defining a new query. The results of our experiments supports the relevance of our approach. They indeed show that schema-guidance improves the learning process.

Title: *Schema-Guided Query Induction*

Keywords: *queries*, *XML schemas*, *grammatical inference*, *trees*, *automata*

"Le luxe de demain sera la lenteur dans le silence."
Anonyme

"Tout n'est pas avantage chez le chercheur. Plus il trouve, moins il a du temps pour connaître sa nouvelle ignorance."
Henri Michaux

Avant-propos

TRADITIONNELLEMENT, ce chapitre s'intitule « Remerciements ». C'est le seul endroit de la thèse qui ne contient pas de science. Je ne le conçois pas comme un exercice obligé, mais comme un espace de liberté et de fantaisie. Il sera plus lu que les pages qui suivent. C'est dire son importance !

Je suis venu à Lille pour la première fois à la fin d'un hiver, le 15 juin 2006. J'y ai rencontré Sophie Tison et Joachim Niehren, qui ont encouragé ma candidature pour obtenir une allocation du ministère de l'Enseignement supérieur et de la recherche. Deux semaines plus tard, j'apprenais que ce financement m'était accordé. Je suis donc reconnaissant à Sophie et à Joachim pour leur appui, point de départ de cette aventure.

Quatre ans ont passé. Je présente aujourd'hui publiquement mes travaux de recherche. Jean-Marc Champarnaud et Colin de la Higuera ont accompli la mission titanesque de rapporter le présent manuscrit ; Françoise Gire et Henning Fernau ont la tâche exaltante d'examiner mon travail. J'espère qu'ils ont attendu son achèvement avec autant d'impatience que moi. Tous ont courageusement accepté de se déplacer dans le Nord pour assister à ma soutenance. Qu'ils soient remerciés de l'intérêt qu'ils portent à mes résultats.

L'aboutissement de ce travail est le fruit d'une collaboration avec trois personnes : Joachim Niehren et Rémi Gilleron l'ont dirigé, main de fer et gant de velours ; Aurélien Lemay m'a efficacement guidé au jour le jour. J'ai apprécié nos multiples séances de réflexion au tableau noir et leur traduction sur le papier, qui m'ont fait comprendre que la vérité d'hier n'est pas nécessairement celle d'aujourd'hui. Je remercie donc chaleureusement mes encadrants.

Il m'est impossible d'énumérer la liste de tous les collègues et néanmoins amis de l'équipe Mostrare que j'ai côtoyés avec plaisir au long de ces quatre années. Pour faire des jaloux, je vais en citer trois. Marc Tommasi a insisté pour faire partie du jury ; je n'ai pas pu le lui refuser. Grégoire Laurence m'a supplié pour écrire des scripts ; difficile de dire non quand vous partagez le même bureau. Benoît Groz m'a menacé pour relire deux chapitres ; j'espère qu'il n'en profitera pas pour me plagier. J'ai su résister à toutes les autres pressions. D'ailleurs, à ce jour, personne n'a réussi à me faire ingurgiter de la bière. Merci à tous pour ces bons moments au travail et en dehors.

Je souhaite aussi remercier les enseignants des universités Lille 1 et Lille 3, en particulier Isabelle Tellier, Yves Roos, Francesco De Comité et Éric Wegrzynowski, à qui j'ai pu faire part de mon ébahissement à la vue de hordes d'étudiants surmotivés. J'ajoute néanmoins que si j'ai la fibre enseignante, je n'ai aucun mérite : c'est génétique.

Mon parcours universitaire, avant de débarquer à Lille, a été marqué par des rencontres décisives, notamment à Lyon. Je pense particulièrement à celles d'Emmanuel Coquery et de Stéphane Bonnevay, maîtres de conférence à l'enthousiasme communicatif, qui m'ont incité à poursuivre ma route en thèse. L'innocent étudiant en master que j'étais ne savait pas encore où il allait mettre les pieds. Je tiens également à bénir les professeurs de français, d'histoire-géographie ou de philosophie, qui m'ont encouragé dans les sciences, ainsi que ceux de mathématiques, de physique-chimie et de biologie, qui m'ont aidé à choisir l'informatique.

Je mesure aussi combien ma mère et mon frère Arnaud supportent difficilement l'éloignement du petit dernier. La lecture régulière de cette thèse les consolera un peu. Je les remercie d'avoir le courage de le faire, tout en leur souhaitant bonne chance. Je compte aussi sur eux pour expliquer en détail mon travail au reste de la famille.

Enfin, ce dernier paragraphe est dédié à ma muse, Elsa. Elle n'a pas seulement relu minutieusement cet ouvrage rempli de symboles mathématiques, synonymes de gros mots pour le commun des mortels. Ce travail lui appartient aussi, car elle fait entièrement partie de moi. Sans son amour et sa présence quotidiens, l'accouchement de cette thèse aurait été nettement plus compliqué. J'espère bien lui rendre la pareille dans d'autres circonstances...

Commis à Lille, le 2 septembre 2010.

Table des matières

Introduction **1**

1 Automates d'arbres et schémas XML **17**

1.1 Expressions régulières et automates de mots 19
1.1.1 Mots et langages 19
1.1.2 Expressions régulières 19
1.1.3 Automates de mots 21
1.1.4 Automates de Glushkov et expressions régulières déterministes . . 23
1.2 Arbres 25
1.2.1 Arbres d'arité bornée 26
1.2.2 Arbres d'arité non bornée 28
1.2.3 Codages binaires 28
1.3 Automates d'arbres 30
1.3.1 Automates pour les arbres d'arité bornée 31
1.3.2 Automates pour les arbres d'arité non bornée 38
1.3.3 Automates *stepwise* factorisés 43
1.4 Schémas XML et automates 45
1.4.1 Définitions de type de documents (DTD) 47
1.4.2 DTD étendues (EDTD) 51
Mémorandum 55

2 Inclusion efficace dans les automates déterministes et les schémas XML **57**

2.1 Introduction 59
2.2 Calcul de complexité avec Datalog clos 62
2.3 Inclusion dans les automates *stepwise* 65

2.3.1 Caractérisation de l'inclusion . . . 66
2.3.2 Test de la caractérisation avec Datalog clos . . . 68
2.3.3 Test d'inclusion en temps $\mathcal{O}(|A| \times |B|)$ . . . 71
2.3.4 Algorithme efficace . . . 73
2.3.5 Détection "au plus tôt" de l'échec de l'inclusion . . . 77
2.3.6 Incrémentalité . . . 79
2.4 Inclusion dans les automates *stepwise* factorisés . . . 79
2.4.1 Test d'inclusion en temps $\mathcal{O}(|A| \times |F|)$ . . . 80
2.4.2 Algorithme efficace . . . 86
2.4.3 Détection "au plus tôt" de l'échec de l'inclusion . . . 88
2.5 Inclusion dans les automates descendant . . . 90
2.6 Inclusion dans les schémas XML . . . 91
2.7 Expériences . . . 92
2.7.1 Données synthétiques . . . 92
2.7.2 Données réalistes . . . 95

3 Requêtes par automate 97
3.1 Introduction . . . 99
3.2 Arbres annotés et projection . . . 102
3.3 Requêtes monadiques . . . 103
3.4 Transducteurs de sélection de nœuds (TSN) . . . 103
3.4.1 Définition d'une requête monadique avec un TSN . . . 104
3.4.2 Calcul de la réponse d'une requête avec un TSN . . . 106

4 Induction de requêtes guidée par schéma 109
4.1 Introduction . . . 111
4.2 Inférence de requêtes régulières . . . 112
4.2.1 Inférence d'automates d'arbres . . . 113
4.2.2 Algorithme $\mathcal{RPNI}$ pour les langages réguliers d'arbres . . . 115
4.2.3 Algorithme t$\mathcal{RPNI}$ pour les requêtes monadiques . . . 117
4.3 Induction de requêtes consistantes avec un schéma . . . 121
4.3.1 Requêtes consistantes avec un schéma . . . 122
4.3.2 Vérification de la consistance par un test d'inclusion . . . 123
4.3.3 Vérification de la consistance par un typage des états . . . 129
4.4 Élagage guidé par schéma . . . 134
4.4.1 Fonction élagage . . . 135

4.4.2 Élaguer relativement au schéma 136
4.4.3 Transducteurs de sélection de nœuds élagueurs 137
4.4.4 Algorithme d'apprentissage avec élagage $\mathrm{t}\mathcal{RPNI}_{\mathrm{D}}^{\mathcal{P}}$ 144
4.5 Apprenabilité des requêtes stables par élagage 148
4.5.1 Stabilité des requêtes monadiques 150
4.5.2 Résultat d'apprenabilité des requêtes stables par élagage 152
4.5.3 Élagages utilisés en pratique 158

5 Évaluation expérimentale 165
5.1 Introduction 167
5.2 Paramètres d'expérimentation 168
5.3 Jeux de données 169
5.3.1 XML 169
5.3.2 HTML 177
5.3.3 Récapitulatif 178
5.4 Protocole d'expérimentation 179
5.5 Résultats et analyse 183
5.5.1 Effets de la vérification de la consistance avec le schéma 184
5.5.2 Apports de l'élagage sans vérifier la consistance avec le schéma . . 187
5.5.3 Combinaison de l'élagage et de la consistance avec le schéma . . . 191
5.5.4 Remarques conclusives 194

Conclusion 197

A Annexes 201
A.1 Calculer la réponse d'une requête représentée par un TSNé 202
A.2 DTD de XMark 204

Bibliographie 207

Index 219

Introduction

De nombreuses applications informatiques, issues aussi bien du mouvement des logiciels libres que des éditeurs de logiciels commerciaux, fonctionnent avec des standards ouverts destinés à favoriser l'interopérabilité, c'est-à-dire, selon la définition élaborée par l'Association francophone des utilisateurs de logiciels libres, *« la capacité que possède un produit ou un système, dont les interfaces sont intégralement connues, à fonctionner avec d'autres produits ou systèmes existants ou futurs et ce sans restriction d'accès ou de mise en œuvre. »* [1]

C'est dans cette optique que le langage XML (Bray *et al.*, 2008) a été recommandé, en février 1998, par le *World Wide Web Consortium* [2] (W3C), un organisme de standardisation à but non lucratif auquel participent notamment des industriels comme HP, IBM, Microsoft, etc., et des institutions comme le MIT, l'ERCIM ou l'Université Keio. XML, pour *eXtensible Markup Language*, est un langage à base de balises autodescriptif et extensible dédié au stockage et au transfert de données de type texte Unicode. Il a été conçu pour être intelligible par les humains et donc facile à mettre en œuvre dans les programmes informatiques. XML est aujourd'hui un format standard pour les communautés bases de données, documents et technologies Internet (Abiteboul *et al.*, 2000; Vianu, 2001).

Un document XML décrit une structure arborescente par l'intermédiaire de balises imbriquées pouvant contenir du texte. En ce sens, un document XML est semi-structuré : structure et contenu y sont mêlés, ce qui le distingue d'une base de données relationnelle, où les données sont stockées dans des tables. Un exemple de document XML décrivant un pays est montré à la figure 1 ; sa représentation sous forme arborescente est donnée figure 2.

1. `http://www.aful.org/gdt/interop`.
2. `http://www.w3.org/`.

```
<country>
  <name>France</name>
  <city>Paris</city>
  <region>
    <name>Nord—Pas de Calais</name>
    <population>3 996 588</population>
    <city>Lille</city>
  </region>
  <region>
    <name>Vallée du Rhône</name>
    <city>Lyon</city>
    <city>Valence</city>
  </region>
</country>
```

Figure 1. *Document XML décrivant un pays.*

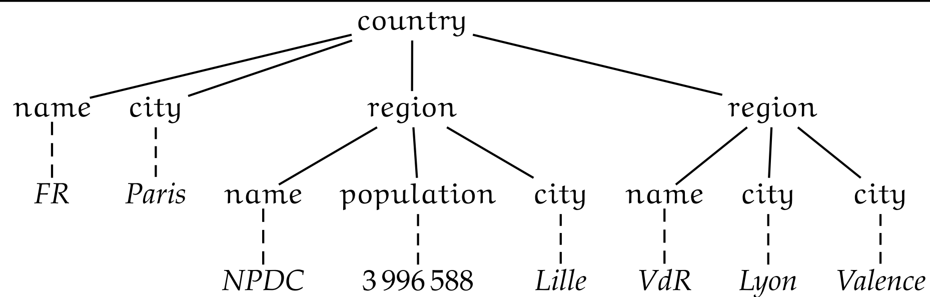


Figure 2. *Représentation arborescente du document XML de la fig. 1.*

```
<!ELEMENT country    (name, city , region*)>
<!ELEMENT region     (name, population?, city*)>
<!ELEMENT name       (#PCDATA)>
<!ELEMENT city       (#PCDATA)>
<!ELEMENT population (#PCDATA)>
```

Figure 3. *DTD permettant de décrire des pays. Le document XML de la fig. 1 se conforme à cette DTD.*

Schémas Un schéma XML détermine, pour une application ou une famille d'applications données, l'ensemble des balises qu'un document peut utiliser, leur articulation, ainsi que le type des données qu'il contient. Il est généralement conçu dans le but de procurer de la sémantique au contenu des documents XML. La présence d'un schéma au sein des applications permet d'améliorer l'efficacité de nombreux traitements (Martens, 2006).

Plusieurs langages existent pour écrire des schémas XML, différant essentiellement par leur syntaxe et leur expressivité. Les plus utilisés sont *Document Type Definition* (définition de type de documents, DTD) (Bray *et al.*, 2008) et XML Schema (Fallside et Walmsley, 2004; Martens *et al.*, 2006), tous deux recommandés par le consortium W3C. Les automates d'arbres (Comon *et al.*, 2007) fournissent un cadre formel pour la définition de langages de schémas (Murata *et al.*, 2005), ainsi que pour l'étude de nombreuses tâches XML (Schwentick, 2007). Un exemple de schéma XML en syntaxe DTD est montré à la figure 3.

Requêtes L'une des tâches les plus basiques lorsque l'on dispose d'une collection de documents consiste à les interroger afin d'en extraire des informations. La profusion de données au format XML a rendu essentielles les questions de la définition et du traitement des requêtes dans les documents XML utilisés comme bases de données. En outre, les requêtes de sélection de nœuds sont à la base de la transformation des documents XML, une tâche essentielle dans un contexte où la structure de ces derniers est caractérisée par une très grande variabilité.

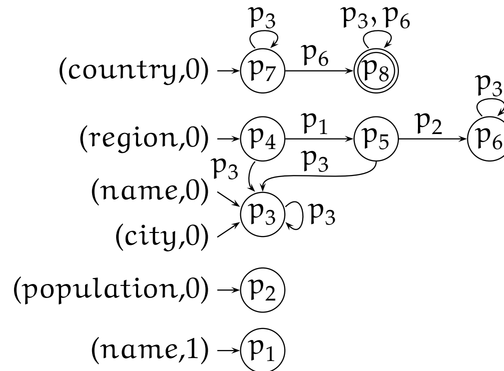
XPath (Clark et DeRose, 1999; Berglund *et al.*, 2007) est le langage recommandé par le W3C pour interroger les documents XML. Il est utilisé par les langages de transformation XQuery (Boag *et al.*, 2007) et XSLT (Clark, 1999), deux standards également élaborés au sein du W3C. Le noyau de XPath (Core XPath 2.0) permet de définir des requêtes de sélection de nœuds dont l'expressivité est équivalente à celle de la logique du premier ordre sur les arbres d'arité non bornée (Marx, 2005). Gottlob et Koch (2004) ont proposé Monadic Datalog comme langage de requêtes monadiques. L'expressivité de Monadic Datalog équivaut à la logique monadique du second ordre (MSO) sur le codage frère-fils des arbres d'arité non bornée. D'après un célèbre théorème de Thatcher et Wright (1968), toute requête définissable avec une formule MSO peut aussi être représentée par un automate d'arbres d'arité bornée. Ce résultat s'applique également aux automates pour les arbres d'arité non bornée, par exemple *via* l'utilisation de codages binaires (Niehren *et al.*, 2005). Une variété de modèles d'automates d'arbres ont été proposés pour la représentation de

« Sélectionner les noms des régions dont on connaît la population. »

(a) langage naturel

//region[population]/name

(b) XPath



(c) transducteur de sélection de nœuds

Figure 4. *Requête XML exprimée en langage naturel et sa représentation en XPath et par un transducteur de sélection de nœuds L'unique réponse à cette requête, calculée sur le document de la fig. 1, serait NPDC.*

requêtes (Neven et Schwentick, 2002; Frick *et al.*, 2003). La figure 4 propose un exemple simple de requête, exprimée d'une part en langage naturel et représentée d'autre part par une expression XPath et un transducteur de sélection de nœuds (Carme *et al.*, 2007).

Automates d'arbres Les premiers résultats formels sur les automates d'arbres remontent à la fin des années 1960 (voir Comon *et al.*, 2007). De nombreux problèmes étudiés ces dernières années dans le cadre du langage XML ont trouvé une issue grâce à ce formalisme qui bénéficie de solides fondations. Des problèmes intéressants sont par exemple l'inclusion de schémas XML ou l'inclusion de requêtes. La possibilité de traiter efficacement ce type de problème dépend essentiellement de la représentation choisie. Les automates d'arbres déterministes constituent un formalisme adéquat dans de nombreuses situations, où le déterminisme est une notion essentielle pour bénéficier d'algorithmes en temps polynomial. L'expressivité de ces automates varie selon que l'on considère un déterminisme ascendant ou bien descendant. Les automates d'arbres déterministes ascendants sont équivalents aux automates non déterministes, donc reconnaissent des

ensembles d'arbres définissables en logique MSO, tandis que l'expressivité des automates déterministes descendants est restreinte aux langages de chemins.

Un modèle d'automates d'arbres alternatif pour les schémas XML est par exemple celui des automates *hedge* (Brüggemann-Klein *et al.*, 2001). Mais ceux-ci disposent d'une notion de déterminisme insuffisante pour pouvoir traiter efficacement le problème d'inclusion proposé au-dessus. Les expressions régulières sont également un formalisme courant dans le contexte des schémas XML. Cependant, elles nécessitent des restrictions pour obtenir des algorithmes d'inclusion polynomiaux. De même, si l'on considère le problème d'inclusion de requêtes, il n'existe aucune notion de déterminisme pour le langage standard XPath, contrairement aux automates d'arbres. Sans aucune restriction, la transformation d'une requête XPath en un automate d'arbres déterministe est une opération de complexité non élémentaire. Les requêtes XPath présentent l'intérêt d'être succinctes et intelligibles comparativement aux automates, comme le montre la figure 4, mais l'utilisation des seconds est plus avantageuse du point de vue opérationnel. La même remarque est valable pour des représentations de schémas XML basées sur des expressions régulières.

Techniques de conception de requêtes

Une première façon de concevoir une requête pour un ensemble de documents XML est d'écrire un programme dans un langage de script comme Perl ou Python. Le principal désavantage de cette approche est qu'elle s'adresse à des experts ayant une bonne maîtrise de ce type d'environnement de programmation. De plus, elle se révèle en pratique coûteuse, source d'erreurs et inadaptée dans le cas de modifications dans la structure des documents XML. Une solution plus appropriée est d'utiliser les outils du W3C comme XPath, XQuery ou XSLT. Mais cette option nécessite tout de même la prise en main d'un formalisme également voué à des programmeurs chevronnés et demeure sensible au changement de structure.

Pour remédier aux limitations des deux approches précédentes, que l'on peut qualifier d'écriture manuelle de requêtes (ou tâches d'extraction, ou *wrappers*), divers systèmes de spécification assistée ont été mis au point, par exemple W4F (Sahuguet et Azavant, 1999), XWrap (Liu *et al.*, 2000), Lixto (Baumgartner *et al.*, 2001) ou encore, plus récemment, Retrozilla (Estievenart *et al.*, 2006). Le principe de ces différents systèmes est de proposer à l'utilisateur un ensemble d'outils pour élaborer des requêtes. Mais, si la complexité du

travail se trouve réduite grâce à l'usage d'une interface graphique, ces systèmes requièrent un niveau de compréhension des outils — souvent des formalismes logiques — qui n'est pas à la portée d'un utilisateur non expert.

Induction de requêtes supervisée Une approche générique, pour laquelle la définition de requêtes sur des documents XML ne nécessite pas de connaissance technique particulière — pas même celle du langage XML — de la part de l'utilisateur, est l'induction supervisée de requêtes. Comme dans les outils évoqués au-dessus, la conception d'une requête se fait à l'aide d'une interface graphique. Mais au lieu de spécifier lui-même les règles d'extraction, l'utilisateur est simplement invité à annoter un ou des documents qu'il souhaite traiter ; la définition de la requête est inférée par le système grâce à un mécanisme d'apprentissage, éventuellement après quelques interactions entre l'utilisateur et le système. Dans la littérature, ce problème est généralement appelé induction de *wrapper* (Kushmerick, 1997) et est appliqué pour automatiser l'extraction d'information dans les documents semi-structurés, notamment les documents Web.

Les premiers travaux dans le domaine ont porté sur l'induction de *wrappers* basés sur les chaînes. La plupart des systèmes mis au point — WIEN (Kushmerick, 1997), SoftMeatly (Hsu et Dung, 1998), Stalker (Muslea *et al.*, 2001), voir aussi Freitag et McCallum (1999), Freitag et Kushmerick (2000), Kushmerick (2000) et Chidlovskii (2001) — reposent sur des approches par machines à états finies (motifs, automates et transducteurs, modèles de Markov cachés). Kushmerick (2002) étudie ces différentes approches. Un inconvénient de celles-ci est qu'elles ne tiennent pas compte de la structure arborescente sous-jacente des documents Web/XML.

Des travaux plus récents se sont intéressés à l'induction de *wrappers* basés sur les arbres. Les techniques d'apprentissage employées dans ces travaux sont variées : programmation logique inductive (Cohen *et al.*, 2002), classification (Gilleron *et al.*, 2006), champs aléatoires conditionnels (Jousse *et al.*, 2006), inférence d'automates d'arbres (Kosala *et al.*, 2003; Carme *et al.*, 2007; Raeymaekers *et al.*, 2008).

Inférence d'automates d'arbres L'inférence d'automates d'arbres relève du domaine de l'inférence grammaticale (de la Higuera, 2010), parfois appelée induction de grammaires ou induction d'automates. D'une manière générale, cette technique d'apprentissage supervisé consiste à inférer la représentation d'un langage formel (grammaire ou automate, le plus souvent de mots) à partir d'un ensemble d'exemples, éléments

appartenant (ou, éventuellement, n'appartenant pas) au langage cible. Une grande variété de problèmes peuvent être modélisés comme des problèmes d'induction de langages, par exemple en reconnaissance de formes ou en traitement de la langue naturelle, et de nombreux algorithmes d'inférence grammaticale ont été élaborés pour les résoudre (voir les études exhaustives de Pitt (1989), Sakakibara (1997) et de la Higuera (2005)).

Dans le cadre de l'induction supervisée de requêtes, ces dernières sont représentées par des langages d'arbres et les exemples sont des arbres annotés par la requête. La méthode proposée par Kosala *et al.* (2003) et Raeymaekers *et al.* (2008) permet d'inférer des requêtes de sélection de nœuds basées sur des propriétés locales, en s'inspirant de l'apprentissage de langages k-testables (García et Vidal, 1990; García, 1993). En comparaison, l'approche de Carme *et al.* (2007), reposant sur l'algorithme $\mathcal{RPNI}$ (Oncina et García, 1992) appliqué aux langages d'arbres (García et Oncina, 1993), est capable d'identifier — au sens de Gold (1967, 1978) — la classe des langages réguliers de requêtes, donc des requêtes plus expressives. Ces deux techniques infèrent des automates d'arbres à partir d'exemples positifs seuls, mais des exemples négatifs sont implicitement disponibles du fait de la nature du problème.

Induction de requêtes guidée par schéma

La réussite des algorithmes d'apprentissage repose sur la présence de biais utilisés pour guider le processus de généralisation (Mitchell, 1990) ; l'essentiel des techniques citées au-dessus s'appuient ainsi soit sur un ensemble fini d'attributs, soit sur des propriétés locales des arbres XML. Mais dans tous ces travaux, aucune information tirée du schéma n'a été prise en compte dans le but d'améliorer la qualité de l'apprentissage ; ni la DTD du langage Web HTML/XHTML, ni une approximation du schéma XML inférée en amont avec un algorithme indépendant (Papakonstantinou et Vianu, 2000; Fernau, 2001; Bex *et al.*, 2008), si aucun schéma n'est disponible dans le contexte de l'application cible, n'ont été exploitées. Si le schéma, qui apporte une connaissance globale sur le domaine, n'a pas été employé dans les algorithmes d'apprentissage jusqu'à présent, c'est parce que la plupart des approches ne permettent pas ou difficilement son intégration.

L'inférence d'automates d'arbres est l'un des rares paradigmes d'apprentissage pour lequel l'utilisation d'un schéma apparaît entièrement possible. En effet, automates d'arbres et schémas XML sont intimement liés dans le sens où tous deux servent à représenter des langages réguliers d'arbres (Murata *et al.*, 2005; Comon *et al.*, 2007). L'objet principal de la

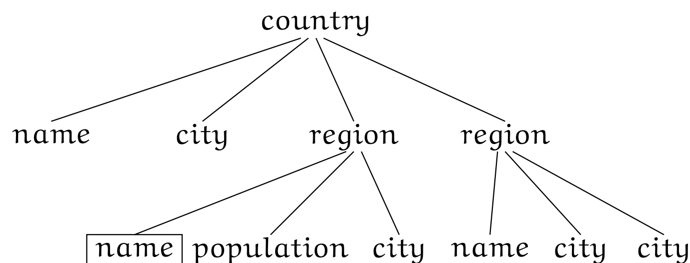
présente thèse est la mise en œuvre et l'évaluation d'algorithmes d'induction de requêtes guidée par schéma, où requêtes et schémas sont représentés par des automates d'arbres. Nos travaux se sont développés autour des deux idées suivantes.

1. La première idée consiste à apprendre des requêtes consistantes avec le schéma des documents XML de l'application cible. Autrement dit, les requêtes inférées sous cette condition ne sont valables que pour des documents satisfaisant le schéma. Coste *et al.* (2004) ont étudié un principe similaire dans le cadre de l'inférence d'automates de mots sous le nom de "biais de domaine". Il s'agit de fournir à l'algorithme d'apprentissage des exemples négatifs implicites afin d'empêcher des généralisations incorrectes. Pour pouvoir mettre en œuvre cette idée, il est essentiel de disposer d'un test d'inclusion extrêmement efficient.

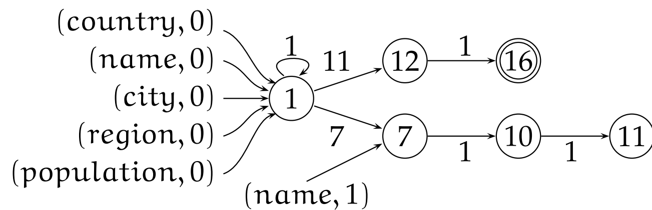
2. La seconde idée est que les informations contenues dans le schéma s'avèrent utiles pour les heuristiques d'élagage employées dans les algorithmes d'induction de requêtes (Carme *et al.*, 2007). L'élagage est notamment utilisé pour réduire la taille des exemples nécessaires à l'identification des requêtes. Toutefois, son usage restreint la classe de requêtes qui peuvent être apprises. La principale difficulté réside dans la formalisation de ce problème d'apprentissage.

Nous soutenons que l'utilisation du schéma, lorsque celui-ci est disponible, dans les algorithmes d'induction de requêtes par inférence d'automates d'arbres permet d'améliorer la qualité de l'apprentissage sans altérer le temps d'exécution.

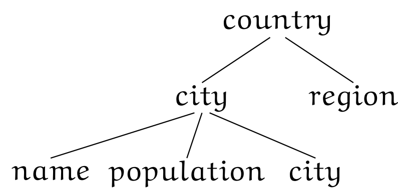
Illustration Supposons qu'un utilisateur souhaite inférer la requête proposée à la figure 4, à savoir celle qui sélectionne le nom des régions dont la population est connue, et qu'il fournisse comme unique exemple l'arbre annoté représenté ci-dessous.



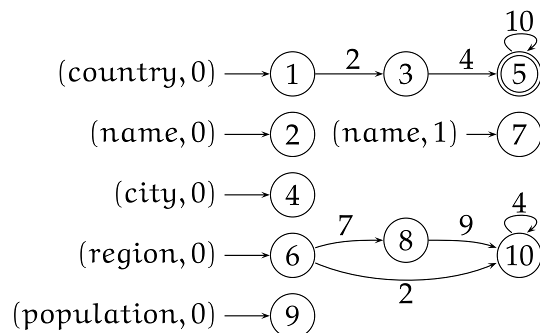
Sans aucune heuristique, un algorithme d'induction de requêtes basé sur l'algorithme standard $\mathcal{RPNI}$ (García et Oncina, 1993) inférerait la requête représentée par le transducteur de sélection de nœuds suivant :



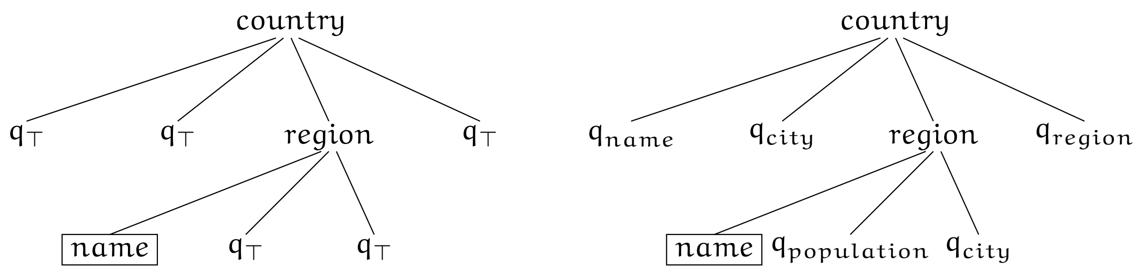
Le principal problème avec cette requête est qu'elle pourrait être appliquée à un arbre-document qui n'est pas conforme par rapport au schéma (DTD de la fig. 3), par exemple :



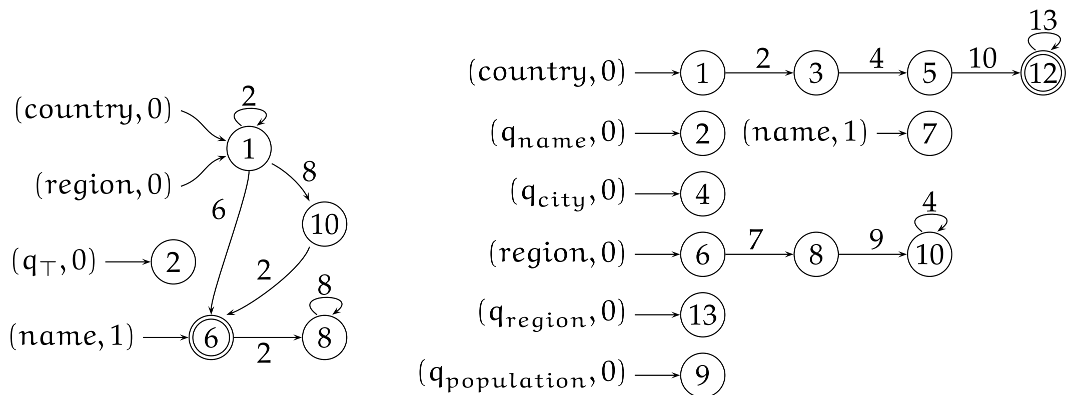
Ici, le nœud $name$ en bas à gauche serait sélectionné par l'application de la requête représentée plus haut. Dans la pratique, cependant, on ne dispose en principe que de documents consistants avec le schéma. On pourrait donc se contenter d'inférer des requêtes qui agissent correctement pour tous les arbres valides sans se préoccuper de leur comportement sur des arbres invalides. Néanmoins, il est clair que cette simple approche n'est pas complètement satifaisante, puisque la requête cible devrait logiquement être consistante avec le schéma. Garantir cette consistance durant tout le processus d'apprentissage a pour but d'empêcher les généralisations, potentiellement mauvaises, qui la violent. Dans notre exemple courant, en considérant comme schéma la DTD de la figure 3, cela conduirait à inférer, avec une simple variante de l'algorithme précédent, la requête consistante représentée comme suit (qui correspond ici à la cible) :



L'élagage consiste, *grosso modo*, à remplacer des parties entières d'un document (typiquement des sous-arbres), jugées non pertinentes pour l'apprentissage, par des étiquettes (symboles) génériques. Il permet un apprentissage à partir d'exemples complètement ou partiellement annotés. Différentes stratégies peuvent être adoptées pour réaliser cette opération. Les deux arbres ci-dessous sont des élagués de l'exemple présenté plus haut.



Dans l'arbre de gauche, les sous-arbres supprimés ont tous été remplacés par l'unique symbole $q_\top$. Dans l'arbre de droite, on a utilisé des informations provenant du schéma. Par exemple, le symbole q_{region} permet de remplacer n'importe quel sous-arbre dont la racine serait étiquetée par $region$. Bien évidemment, d'autres stratégies sont imaginables. Intuitivement, il apparaît que la requête donnée en exemple plus haut ne peut pas être inférée avec le premier élagage, tandis que le second le permet. En effet, l'information utile pour sélectionner le nœud $name$ est la présence de $population$ parmi ses successeurs, ce qui est perdu dans l'arbre de gauche mais pas dans celui de droite, où le symbole $q_{population}$ porte cette information grâce à l'usage du schéma. Le même processus d'apprentissage que précédemment, adapté aux arbres élagués, conduirait respectivement aux deux requêtes représentées comme suit :



À gauche, il n'est pas possible de distinguer si un nœud étiqueté par $name$ a un successeur étiqueté par $population$. Appliquer cette requête pourrait donc conduire à sélectionner des noms de régions dont on ne connaît pas la population. L'utilisation du schéma dans la stratégie d'élégage permet ici au contraire d'apprendre la requête cible ; nous dirons qu'elle est stable par cet élagage.

Limites Avant de détailler les contributions de cette thèse, nous en fixons les contours. Les requêtes généralement considérées dans le contexte des bases de données permettent de sélectionner des ensembles de tuples constitués de n éléments. Nous nous cantonnons ici à des requêtes monadiques, c'est-à-dire qui sélectionnent des tuples à un seul élément. L'extension de notre approche pour l'induction de requêtes n-aires est triviale pour ce qui concerne la consistance avec le schéma, mais différents choix pourraient être effectués pour l'élagage. Par ailleurs, les requêtes que nous considérons ne portent que sur la structure des arbres XML et laissent de côté leur contenu textuel éventuel. Des techniques appropriées de recherche d'information dans les chaînes utilisées en combinaison de notre approche permettraient de capturer ces requêtes.

Contributions

Nos contributions s'inscrivent principalement dans deux domaines : la théorie des langages formels et l'inférence grammaticale. Elles sont à la fois d'ordre théorique et d'ordre pratique.

Nous avons proposé une nouvelle notion d'automates d'arbres, à savoir les automates factorisés. Ceux-ci correspondent à des automates *stepwise* (Carme *et al.*, 2004) avec ϵ-règles et sont destinés à représenter des automates *stepwise* déterministes de façon compacte. La transformation d'une DTD directement en automate *stepwise* déterministe conduit à une explosion quadratique de la taille de celui-ci par rapport à la taille de la DTD d'entrée ; la factorisation a pour but d'éviter cela. Nous montrons comment transformer une DTD déterministe D sur un alphabet Σ en un automate factorisé déterministe en temps $\mathcal{O}(|\Sigma| \times |D|)$, où $|D|$ (resp. $|\Sigma|$) est la taille de D (resp. Σ). D'autres constructions d'automates sont clarifiées.

Une contribution majeure est la présentation d'algorithmes efficaces pour vérifier l'inclusion dans les automates d'arbres déterministes. Dans le cas non déterministe, la complexité du problème d'inclusion est DEXPTIME-complet (Seidl, 1990). La méthode usuelle pour

vérifier l'inclusion $\mathcal{L}(A) \subseteq \mathcal{L}(B)$, où B est déterministe et A possiblement non déterministe avec ϵ-règles, consiste à se ramener au problème équivalent $\mathcal{L}(A) \cap \mathcal{L}(B)^c = \emptyset$ (Comon *et al.*, 2007). L'écueil de cette approche est qu'elle implique le calcul du complémentaire de l'automate B, ce qui nécessite de le compléter. Cette dernière opération a un coût de $\mathcal{O}(|\Sigma| \times |B|^2)$ pour un automate d'arbres binaires B sur Σ, d'où un test d'inclusion en temps $\mathcal{O}(|A| \times |\Sigma| \times |B|^2)$. En comparaison, notre algorithme d'inclusion s'exécute en temps $\mathcal{O}(|A| \times |B|)$ dans le pire des cas, indépendamment de la taille l'alphabet. La rapidité du test d'inclusion est renforcée par un mécanisme de détection de l'échec de l'inclusion "au plus tôt". Des expériences confirment l'efficacité de cette approche. En outre, l'algorithme d'inclusion est incrémental par rapport à l'ajout d'ϵ-règles à l'automate A ; cela permet de réduire le coût global des tests d'inclusion lors de leur utilisation dans un algorithme d'inférence d'automate par fusions d'états. Ces résultats sont étendus à l'inclusion dans les automates *stepwise* factorisés, puis aux schémas XML déterministes D dans lesquels l'inclusion peut être vérifiée en temps $\mathcal{O}(|A| \times |\Sigma| \times |D|)$. Le facteur $|\Sigma|$ est lié à la transformation des schémas en automates d'arbres déterministes et ne peut être éliminé. Selon le modèle de schéma considéré, différents formalismes d'automates peuvent être choisis pour A : automates *stepwise* ou frère-fils pour les arbres d'arité non bornée, ou automates *hedge* (Brüggemann-Klein *et al.*, 2001). L'intérêt de cette contribution est à la fois théorique et pratique.

Le cœur de ce travail est l'induction de requêtes guidée par schéma. Nous avons étudié en particulier deux utilisations du schéma dans un algorithme d'inférence de langages d'arbres basé sur une variante de $\mathcal{RPNI}$ (García et Oncina, 1993), un algorithme classique d'inférence grammaticale qui généralise un langage représenté par un automate par le biais de fusions d'états.

1. La première utilisation consiste à assurer que la requête apprise demeure consistante avec le schéma. Pour cela, on applique le test d'inclusion discuté au-dessus, où A est l'automate appris et D le schéma ; on tire parti de son incrémentalité en modélisant les fusions d'états par des ϵ-règles. En pratique, le coût supplémentaire lié à la vérification de la consistance avec le schéma est compensé par une réduction importante du nombre de fusions tentées par l'algorithme d'apprentissage. Notez que nous prouvons que les requêtes consistantes avec le schéma sont apprenables. À notre connaissance, bien que l'idée de vérifier la consistance avec un domaine ait été avancée par Coste *et al.* (2004) dans le cas des mots, ce résultat formel n'avait pas encore été démontré.

2. La deuxième utilisation du schéma que nous avons considérée s'inscrit dans le cadre de l'élagage. Notre apport sur ce point est double : d'une part, nous présentons une généralisation des heuristiques d'élagage examinées par Carme *et al.* (2007) qui tient compte de la connaissance du schéma, d'autre part, nous donnons pour la première fois une caractérisation précise de la classe de requêtes apprenable relativement à un élagage, à savoir la classe des requêtes stables. Un élagage qui fait usage du schéma conserve un certain nombre d'informations perdues par des élagages plus brutaux, ce qui permet d'élargir la classe des requêtes qui peuvent être inférées. Ces nouveaux résultats comblent un fossé qui existait entre la pratique et la théorie.

Outre les contributions théoriques passées en revue ci-dessus, nous avons effectué une évaluation expérimentale de l'induction de requêtes guidée par schéma. Pour la réaliser, le développement d'un système complet d'inférence d'automates d'arbres a été nécessaire. Celui-ci a été testé en simulant le comportement d'un utilisateur lors de la définition d'une nouvelle requête. Les effets de la consistance et de l'élagage sont étudiés, d'abord indépendamment l'un de l'autre, puis en conjonction l'un avec l'autre. Les résultats expérimentaux montrent l'intérêt de l'utilisation du schéma pour l'induction de requêtes.

Plan de la thèse

Le présent manuscrit est organisé en cinq chapitres.

Le premier chapitre pose les bases formelles sur lesquelles reposent les résultats théoriques présentés dans la suite. Nous donnons les définitions et les notations pour les automates de mots et les expressions régulières, les automates d'arbres et les langages de schémas. Dans ce chapitre, nous introduisons également le modèle des automates *stepwise* factorisés et présentons deux constructions qui permettent de transformer des schémas déterministes en automates d'arbres déterministes en temps polynomial.

Le second chapitre est consacré à l'inclusion efficace dans les automates d'arbres déterministes. Les résultats de complexité sont prouvés à l'aide de Datalog clos, outil que nous introduisons en début de chapitre. L'algorithme est d'abord étudié pour les automates d'arbres *stepwise* avant d'être étendu aux automates factorisés. Ce passage est particulièrement technique. L'inclusion dans les schémas XML déterministes en devient triviale. Le chapitre se conclut par quelques expériences qui permettent de vérifier les complexités théoriques obtenues.

Le troisième chapitre présente les requêtes par automate d'arbres. Ce court chapitre de transition ne propose pas de nouvelle contribution. Il discute différents formalismes de requête existant pour le langage XML et argumente le choix des automates *stepwise* pour représenter les requêtes. Plus précisément, le modèle retenu est celui des transducteurs de sélection de nœuds. Ce sont des automates *stepwise* qui permettent de représenter des requêtes et d'en calculer efficacement les réponses.

Le quatrième chapitre s'intéresse à l'induction de requêtes guidée par schéma. Il présente le modèle d'apprentissage retenu pour ce problème, à savoir l'identification en temps polynomial et données de cardinalité polynomiale. Des algorithmes d'inférence d'automates d'arbres basés sur $\mathcal{RPNI}$ permettant d'intégrer d'une part la consistance et d'autre part l'élagage sont examinés. Les deux heuristiques dépendent du schéma, cependant l'élagage restreint la classe de requêtes apprenable par cette technique. Une dernière section caractérise précisément cette classe.

Le cinquième chapitre expose une évaluation expérimentale de l'induction de requêtes guidée par schéma. Il discute les paramètres d'expérimentation et introduit un ensemble de jeux de données pertinent pour cette évaluation. Les effets respectifs et combinés de la vérification de la consistance et de l'élagage guidés par schéma sont comparés et analysés par le biais de simulations.

Publications associées

Plusieurs résultats présentés dans cette thèse ont conduit à diverses publications, récapitulées ci-dessous. L'idée de guider l'algorithme d'induction de requêtes avec le schéma XML a été présentée une première fois au *workshop* [CAGI07], puis développée lors de la conférence [ICGI08]. La caractérisation des requêtes apprenables avec un élagage et l'évaluation expérimentale sont inédits. Les algorithmes d'inclusion efficaces pour les automates d'arbres et les schémas XML déterministes ont été publiés initialement à la conférence [LATA08]. Ces travaux ont été suivis d'une publication dans un journal [I&C09].

[I&C09] Jérôme CHAMPAVÈRE, Rémi GILLERON, Aurélien LEMAY et Joachim NIEHREN : *Efficient Inclusion Checking for Deterministic Tree Automata and XML Schemas*. Information and Computation, 207(11):1181–1208, novembre 2009.

[ICGI08] Jérôme CHAMPAVÈRE, Rémi GILLERON, Aurélien LEMAY et Joachim NIEHREN : *Schema-Guided Induction of Monadic Queries*. *In* 9th International Colloquium on Grammatical Inference, 2008.

[LATA08] Jérôme CHAMPAVÈRE, Rémi GILLERON, Aurélien LEMAY et Joachim NIEHREN : *Efficient Inclusion Checking for Deterministic Tree Automata and DTDs*. *In* 2nd International Conference on Language and Automata Theory and Applications, 2008.

[CAGI07] Jérôme CHAMPAVÈRE, Rémi GILLERON, Aurélien LEMAY et Joachim NIEHREN : *Towards Schema-Guided XML Query Induction*. *In* ICML Workshop on Challenges and Applications of Grammar Induction, 2007.

1

Automates d'arbres et schémas XML

Les automates d'arbres constituent une abstraction naturelle pour représenter des schémas XML. Dans ce chapitre, nous nous intéressons principalement à des modèles d'automates d'arbres déterministes destinés à représenter des définitions de schémas XML déterministes (DTD et DTD étendues notamment). À cet effet, nous examinons deux codages binaires pour les arbres d'arité bornée ou non, à savoir le codage curryfié et le codage frère-fils, et arguons que le premier est bien adapté à la perspective "déterminisme ascendant" tandis que le second convient plutôt au "déterminisme descendant". Par ailleurs, nous introduisons une nouvelle notion d'automates d'arbres, les automates factorisés, et montrons comment convertir une DTD déterministe D sur un alphabet Σ en un automate factorisé déterministe en temps $\mathcal{O}(|\Sigma| \times |D|)$. Des constructions classiques sur les expressions régulières, les automates de mots et d'arbres sont présentées préalablement à cette contribution. Nous discutons plus généralement des modèles d'automates d'arbres déterministes pour les arbres d'arité non bornée, en relation avec les problèmes traités dans les chapitres qui suivent.

Sommaire

1.1 Expressions régulières et automates de mots 19

1.1.1 Mots et langages 19

1.1.2 Expressions régulières 19

1.1.3 Automates de mots 21

1.1.4 Automates de Glushkov et expressions régulières déterministes . 23

1.2 Arbres 25

1.2.1 Arbres d'arité bornée 26

1.2.2 Arbres d'arité non bornée 28

1.2.3 Codages binaires 28

1.3 Automates d'arbres 30

1.3.1 Automates pour les arbres d'arité bornée 31

1.3.2 Automates pour les arbres d'arité non bornée 38

1.3.3 Automates *stepwise* factorisés 43

1.4 Schémas XML et automates 45

1.4.1 Définitions de type de documents (DTD) 47

1.4.2 DTD étendues (EDTD) 51

Mémorandum 55

1.1 Expressions régulières et automates de mots

Cette première section est consacrée à un bref rappel de résultats élémentaires de la théorie des langages formels (Hopcroft et Ullman, 1979). On s'intéresse ici aux langages réguliers de mots et à leur représentation par automates finis ou expressions régulières, deux formalismes équivalents. Nous étudions en particulier le lien entre expressions régulières déterministes, utilisées pour définir les modèles de contenu dans différents langages de schémas XML, et automates de Glushkov déterministes.

1.1.1 Mots et langages

Un *alphabet* est un ensemble fini non vide de symboles. La *taille* d'un alphabet Σ, dénotée $|\Sigma|$, est sa cardinalité.

Un *mot* (ou une *chaîne*) sur un alphabet Σ est une séquence de symboles de l'alphabet Σ. Le *mot vide* est désigné par ϵ. La concaténation de deux mots w_1, w_2 sur Σ est notée $w_1 \cdot w_2$.

Un *langage de mots* sur Σ est un ensemble de mots sur Σ. Le *langage vide* est noté $\emptyset$. Le *langage universel* sur Σ, dénoté Σ^*, est l'ensemble de tous les mots sur Σ. L'union de deux langages $L_1, L_2 \subseteq \Sigma^*$ est l'ensemble $L_1 \cup L_2 = \{w \mid w \in L_1 \vee w \in L_2\}$. La concaténation de L_1 et L_2 est l'ensemble $L_1 \cdot L_2 = \{w_1 \cdot w_2 \mid w_1 \in L_1 \wedge w_2 \in L_2\}$. On note L_1^i pour dénoter i fois $L_1 \cdot L_1 \cdots L_1$; l'étoile (de Kleene) de L_1 est l'ensemble $L_1^* = \{\epsilon\} \cup \bigcup_{i=1}^{\infty} L_1^i$.

La classe des langages réguliers de mots est le plus petit ensemble de langages sur Σ close par union, concaténation et étoile, contenant $\emptyset$, $\{\epsilon\}$ et $\{a\}$ pour tout $a \in \Sigma$.

1.1.2 Expressions régulières

Les expressions régulières caractérisent la classe des langages réguliers de mots.

Une *expression régulière* (ou *expression rationnelle*) e sur Σ et le langage $\mathcal{L}(e)$ qu'elle décrit sont définis inductivement comme suit :

- $e = \emptyset$ dénote le langage $\mathcal{L}(e) = \emptyset$;
- $e = \epsilon$ dénote le langage $\mathcal{L}(e) = \{\epsilon\}$;
- $e = a$ avec $a \in \Sigma$ dénote le langage $\mathcal{L}(e) = \{a\}$;
- soient e_1 et e_2 deux expressions régulières sur Σ avec $\mathcal{L}(e_1)$ et $\mathcal{L}(e_2)$ leurs langages respectifs, alors :

- $e = e_1 + e_2$ dénote le langage $\mathcal{L}(e) = \mathcal{L}(e_1) \cup \mathcal{L}(e_2)$;
- $e = e_1 \cdot e_2$ dénote le langage $\mathcal{L}(e) = \mathcal{L}(e_1) \cdot \mathcal{L}(e_2)$;
- $e = e_1^*$ dénote le langage $\mathcal{L}(e) = \{\epsilon\} \cup \bigcup_{i=1}^{\infty} \mathcal{L}(e_1)^i$.

L'opérateur (unaire) étoile * est plus prioritaire que l'opérateur (binaire) de concaténation $\cdot$, lui-même plus prioritaire que l'opérateur (binaire) d'union $+$. L'usage de parenthèses permet de changer la priorité des opérations.

Une expression régulière sur $\Sigma = \{a, b\}$ est par exemple $b^* \cdot a \cdot (b^* \cdot a \cdot b^* \cdot a)^* \cdot b^*$. Celle-ci représente le langage des mots qui contiennent un nombre impair de a.

La *taille* d'une expression régulière e, dénotée $|e|$, est donnée par le nombre de symboles utilisés dans sa représentation (symboles de l'alphabet plus opérateurs).

Déterminisme Les langages de schémas XML qui utilisent la norme Iso SGML (*Standard Generalized Markup Language*), par exemple DTD ou W3C XML Schema, imposent aux expressions régulières utilisées comme modèle de contenu d'être déterministes (ou 1-non-ambiguës (Brüggemann-Klein et Wood, 1998)) afin de faciliter la validation des documents XML. Nous verrons plus loin les implications de cette notion de déterminisme en termes opératoires.

Informellement, une expression régulière est déterministe si, lors de la lecture de gauche à droite d'un mot appartenant au langage décrit par celle-ci, tout symbole du mot peut être associé à un unique symbole de l'expression régulière sans qu'il soit nécessaire de lire le symbole suivant dans le mot.

Une expression régulière non déterministe est par exemple $(a + b)^* \cdot a \cdot a^*$. En effet, le mot baa peut être lu de deux façons différentes, comme l'illustre le dessin ci-dessous.

Pour définir formellement ce concept, on effectue un marquage de l'expression afin de distinguer les occurrences d'un même symbole dans un mot. Par exemple, un marquage de $e = (a + b)^* \cdot a \cdot a^*$ est $e_m = (a_1 + b_1)^* \cdot a_2 \cdot a_3^*$. Étant donné un alphabet Σ, l'ensemble des symboles marqués de Σ est dénoté Σ_m.

Soient e une expression régulière sur Σ et e_m un marquage de e sur Σ_m. L'expression e est *déterministe* (ou *1-non-ambiguë*) s'il n'existe pas deux mots $w \cdot a_i \cdot w'$ et $w \cdot a_j \cdot w''$ dans $\mathcal{L}(e_m)$ tels que $i \neq j$, avec $w, w', w'' \in \Sigma_m^*$ et $a \in \Sigma$.

Un langage régulier est 1-non-ambigu s'il peut être défini par une expression régulière déterministe. Le langage dénoté par l'expression $e = (a + b)^* \cdot a \cdot a^*$ est 1-non-ambigu car il existe une expression équivalente $e' = b^* \cdot a \cdot (b^* \cdot a)^*$ qui dénote le même langage. Cependant, toute expression régulière non déterministe n'est pas équivalente à une expression régulière déterministe. C'est le cas par exemple de l'expression $(a + b)^* \cdot a \cdot (a + b)^n$, $n \geqslant 1$. Par conséquent, les langages 1-non-ambigus forment une sous-classe stricte des langages réguliers. Les détails de leur caractérisation ont été fournis par (Brüggemann-Klein et Wood, 1998).

Dans la suite, nous définissons les automates de mots, puis nous étudions le lien entre expressions régulières déterministes et automates de mots déterministes.

1.1.3 Automates de mots

Les automates de mots sont des machines à états finies qui reconnaissent des ensembles de mots.

Un *automate de mots* (ou *automate fini*) A sur Σ est un quadruplet $\big(\mathtt{sta}(A)$, $\mathtt{init}(A)$, $\mathtt{fin}(A)$, $\mathtt{rul}(A)\big)$, où $\mathtt{sta}(A)$ est un ensemble fini d'*états* avec $\mathtt{init}(A) \subseteq \mathtt{sta}(A)$ (resp. $\mathtt{fin}(A) \subseteq \mathtt{sta}(A)$) les *états initiaux* (resp. *finaux*) de A, et $\mathtt{rul}(A) \subseteq \big(\mathtt{sta}(A) \times \Sigma \times \mathtt{sta}(A)\big) \uplus \mathtt{sta}(A)^2$ un ensemble fini de *règles* (ou *transitions*) de la forme $p_1 \xrightarrow{a} p_2$ ou $p_1 \xrightarrow{\epsilon} p_2$, avec $a \in \Sigma$ et $p_1, p_2 \in \mathtt{sta}(A)$. Les règles $p_1 \xrightarrow{\epsilon} p_2 \in \mathtt{rul}(A)$ sont appelées *epsilon-règles* (ϵ-*règles*), et l'on note $p_1 \xrightarrow{\epsilon}_A p_2$ si et seulement si $p_1 \xrightarrow{\epsilon} p_2 \in \mathtt{rul}(A)$; la notation $\xrightarrow{\epsilon}_A^*$ correspond à la clôture transitive de la relation $\xrightarrow{\epsilon}_A$.

On représente graphiquement un automate de mots A par un (multi-)graphe orienté dont les sommets sont étiquetés par les états de l'automate et les arêtes par les symboles de l'alphabet. Une règle $p_1 \xrightarrow{a} p_2$ (resp. $p_1 \xrightarrow{\epsilon} p_2$) de A est représentée par une arête de p_1 vers p_2 étiquetée par a (resp. ϵ). On distingue les états initiaux de A par des flèches entrantes et ses états finaux par des doubles cercles. Un exemple est donné figure 5.

Soit A un automate de mots sur Σ. Un état p de A est accessible s'il existe un chemin depuis un état initial jusqu'à p dans le graphe représentant l'automate ; de même, un

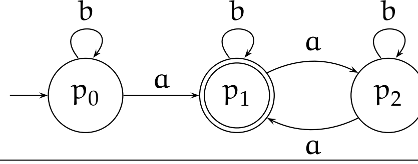


Figure 5. *Représentation graphique d'un automate de mots (déterministe) sur un alphabet* $\Sigma = \{a, b\}$. *Son état initial est* p_0 *et son unique état final est* p_1. *Cet automate reconnaît le langage des mots sur* Σ *qui contiennent un nombre impair de* a.

état p est coaccessible s'il existe un chemin menant de p à un état final. L'automate A est productif (ou émondé) si tous ses états sont à la fois accessibles et coaccessibles.

Tout automate de mots A sur Σ définit une *fonction d'évaluation* $eval_A^{\rightarrow} : \Sigma^* \to 2^{sta(A)}$ telle que :

$$eval_A^{\rightarrow}(w) = \begin{cases} \{p \mid p' \in init(A) \land p' \xrightarrow{\epsilon}_A^* p\} & \text{si } w = \epsilon \\ \{p \mid p' \in eval_A^{\rightarrow}(w') \land p' \xrightarrow{a} p'' \in rul(A) \land p'' \xrightarrow{\epsilon}_A^* p\} & \text{si } w = w' \cdot a \end{cases}$$

Un mot $w \in \Sigma^*$ est *reconnu* par A si $eval_A^{\rightarrow}(w) \cap fin(A) \neq \emptyset$. Le *langage* de A, dénoté $\mathcal{L}(A)$, est l'ensemble des mots reconnus par l'automate A :

$$\mathcal{L}(A) = \{w \in \Sigma^* \mid eval_A^{\rightarrow}(w) \cap fin(A) \neq \emptyset\}$$

On note $\mathcal{L}_p(A)$ l'ensemble des mots reconnus à partir d'un état p de A, c'est-à-dire :

$$\mathcal{L}_p(A) = \{w \in \Sigma^* \mid u = v \cdot w \land p \in eval_A^{\rightarrow}(v) \land u \in \mathcal{L}(A)\}$$

Un automate de mots A est *déterministe* s'il n'a pas d'ϵ-règles et si pour tout $p \in sta(A)$ et tout $a \in \Sigma$ il existe au plus un état p' tel que $p \xrightarrow{a} p' \in rul(A)$. Tout automate non déterministe sur Σ peut être rendu déterministe en temps exponentiel dans le nombre d'états de l'automate. Un automate déterministe A est *complet* si pour tout $a \in \Sigma$ et tout $p \in sta(A)$ il existe un état $p' \in sta(A)$ tel que $p \xrightarrow{a} p' \in rul(A)$. On peut compléter un automate déterministe A en temps $\mathcal{O}(|\Sigma| \times |sta(A)|)$, où $|sta(A)|$ dénote la cardinalité de $sta(A)$, en ajoutant un état puits $p_\perp$ et les règles $p \xrightarrow{a} p_\perp$ manquantes. Notons qu'un automate complet déterministe construit de cette façon n'est pas productif car l'état puits n'est pas coaccessible.

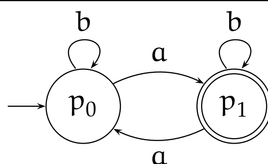


Figure 6. *Automate (complet) déterministe minimal qui reconnaît le langage des mots sur* $\Sigma = \{a, b\}$ *contenant un nombre impair de* a. *Le langage de l'état initial* p_0 *correspond à l'ensemble des mots sur* Σ *qui contiennent un nombre pair de* a, *l'état final* p_1 *à l'ensemble des mots sur* Σ *qui en contiennent un nombre impair.*

La famille des langages reconnus par automates de mots est close par union, intersection et complémentaire. Un résultat classique de la théorie des langages montre que les langages reconnaissables par automates finis sont exactement la classe des langages réguliers (Kleene, 1956). Autrement dit, à toute expression régulière correspond un automate de mots qui reconnaît le même langage, et réciproquement. Nous reviendrons sur ce point dans la section suivante.

Soit A un automate de mots déterministe sur Σ, deux états p et p' de A sont dits équivalents si $w \in \mathcal{L}_p(A) \Leftrightarrow w \in \mathcal{L}_{p'}(A)$ pour tout mot $w \in \Sigma^*$. L'automate A est *minimal* si aucun état de $\mathrm{sta}(A)$ n'est équivalent à un autre. Soit L un langage reconnaissable par automate de mots, le théorème dit de Myhill-Nerode (Myhill, 1957; Nerode, 1958) permet d'établir qu'il existe un unique automate minimal en nombre d'états qui reconnaît L. La figure 6 montre l'automate minimal pour le langage des mots sur $\{a, b\}$ contenant un nombre impair de a. Notons qu'il est possible de minimiser un automate fini déterministe A sur Σ en temps $\mathcal{O}(|\Sigma| \times |\mathrm{sta}(A)| \times \log(|\mathrm{sta}(A)|))$ (voir p. ex. Hopcroft, 1971; Blum, 1996; Knuutila, 2001; Béal et Crochemore, 2008). Watson (1993b) étudie les différentes méthodes pour les automates complets déterministes.

On s'intéresse dans la suite à la transformation d'expressions régulières déterministes en automates de mots déterministes.

1.1.4 Automates de Glushkov et expressions régulières déterministes

Kleene (1956) a montré que les langages réguliers sont exactement les langages reconnaissables par automates finis, c'est-à-dire que la classe de langages définie par les expressions régulières est la même que celle des automates de mots. Il existe une variété d'algorithmes pour transformer une expression régulière en un automate fini équivalent, et *vice versa*.

On s'intéresse ici au premier problème, à savoir transformer une expression régulière en automate de mots, pour lequel il existe des algorithmes efficaces, c'est-à-dire en temps polynomial ; dans l'autre cas, c'est-à-dire automate fini vers expression régulière, la transformation peut conduire, dans le pire des cas, à une explosion exponentielle de la taille de la sortie (Ehrenfeucht et Zeiger, 1976).

Différentes approches ont été proposées pour construire un automate de mots à partir d'une expression régulière. Watson (1993a) en propose une taxonomie. On peut distinguer les approches qui produisent des automates avec ϵ-transitions, notamment la construction de Thompson (1968), de celles qui produisent des automates sans ϵ-transitions. Parmi ces dernières, Berry et Sethi (1986) ont explicité une construction donnée indépendamment par McNaughton et Yamada (1960) et Glushkov (1961) ; différentes améliorations en ont été proposées par Brüggemann-Klein (1993), Chang et Paige (1997), et Ziadi *et al.* (1997). D'autres méthodes de construction sans ϵ-transitions ont été développées ces dernières années (Antimirov, 1996; Champarnaud et Ziadi, 2002; Ilie et Yu, 2003) ; Ouardi (2007) étudie les alternatives.

Nous nous intéressons spécialement à la construction de Glushkov (1961). Un automate de Glushkov possède la particularité d'être homogène, c'est-à-dire que toutes les transitions entrantes d'un état sont nécessairement étiquetées par le même symbole ; il n'a qu'un seul état initial et celui-ci ne comporte aucune transition entrante [1]. Son principe général de construction, étant donnée une expression régulière e sur un alphabet Σ, est le suivant :

- on considère e_m, un marquage de l'expression e ;
- on définit un état pour chaque symbole a_i de l'expression marquée e_m, et l'on ajoute un état initial q_0 ;
- on ajoute une règle $a_i \stackrel{a}{\rightarrow} a_j$ si le symbole a_j peut suivre le symbole a_i dans un mot du langage $\mathcal{L}(e_m)$;
- on ajoute une règle $q_0 \stackrel{a}{\rightarrow} a_i$ si le symbole a_i peut être le premier symbole d'un mot dans $\mathcal{L}(e_m)$;
- l'état a_i est final si a_i peut être le dernier symbole dans un mot de $\mathcal{L}(e_m)$;
- l'état q_0 est final si ϵ appartient au langage $\mathcal{L}(e_m)$.

L'automate de Glushkov possède $n + 1$ états, où n est le nombre de symboles de Σ utilisés dans e. Sa construction directe s'effectue en temps $\mathcal{O}(n^3)$; les optimisations de

1. Notons que ces propriétés ne sont pas suffisantes pour caractériser les automates de Glushkov (Caron et Ziadi, 2000).

Brüggemann-Klein (1993), Chang et Paige (1997), et Ziadi *et al.* (1997) abaissent la complexité à $\mathcal{O}(n^2)$. Notons que ces temps de calcul présupposent Σ fixé. Dans le cas général, le passage d'une expression régulière à un automate de Glushkov produit un automate fini non déterministe.

Comme nous l'avons mentionné plus haut, les expressions régulières utilisées comme modèles de contenu dans les langages de schémas recommandés par le consortium W3C — DTD et XML Schema — doivent être déterministes pour permettre une validation efficace des documents XML. Cette condition a été minutieusement étudiée et formalisée par Brüggemann-Klein et Wood (1998) en terme de 1-non-ambiguïté. Brüggemann-Klein (1993) a montré que les automates de Glushkov obtenus à partir de telles expressions sont déterministes, et réciproquement, et qu'ils peuvent être calculés efficacement. Nous résumons ces deux résultats dans le théorème suivant.

Théorème 1.1 (Brüggemann-Klein (1993))
Une expression régulière e sur Σ est déterministe si et seulement si son automate de Glushkov G_e est déterministe. Dans ce cas, l'automate G_e peut être calculé en temps [2] $\mathcal{O}(|\Sigma| \times |e|)$.

Notons que l'automate de Glushkov d'une expression régulière déterministe n'est généralement pas un automate minimal, ni un automate complet. L'automate minimal peut-être calculé avec un coût supplémentaire de $\mathcal{O}(|\Sigma| \times n \times \log n)$ en temps et de $\mathcal{O}(|\Sigma| \times n)$ en espace (Béal et Crochemore, 2008), où n est le nombre d'états de l'automate de Glushkov et Σ son alphabet.

La figure 7 montre les automates de Glushkov construits à partir des expressions régulières $(a + b)^* \cdot a \cdot a^*$ et $b^* \cdot a \cdot (b^* \cdot a)^*$. Ces deux expressions reconnaissent le même langage, à savoir l'ensemble des mots sur $\{a, b\}$ qui contiennent au moins un a et qui terminent par un nombre quelconque de a ; la seconde est déterministe.

1.2 Arbres

Les arbres sont des structures de données récursives très répandues dans les algorithmes (Knuth, 1997). Nous considérons des arbres finis, ordonnés et étiquetés. On distingue les arbres d'arité bornée, pour lesquels le nombre d'enfants de chaque nœud est fixé, des arbres d'arité non bornée. Ces derniers sont notamment utilisés comme formalisme

2. Brüggemann-Klein (1993) suppose Σ fixé et donc aboutit à une complexité en temps linéaire dans la taille de e. Nous conservons ici le facteur $|\Sigma|$ en supposant que l'expression e utilise tous les symboles de Σ.

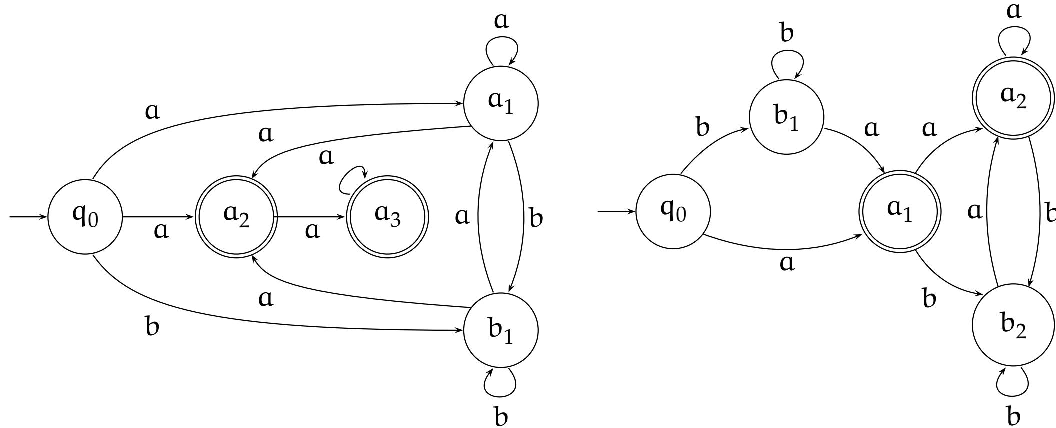


Figure 7. *Automates de Glushkov des expressions régulières* $(a+b)^* \cdot a \cdot a^*$ *(à gauche) et* $b^* \cdot a \cdot (b^* \cdot a)^*$ *(à droite). Seule la seconde est déterministe.*

de représentation pour les documents XML. Les codages binaires permettent d'établir une correspondance entre les arbres d'arité bornée et non bornée et leurs langages. Nous définissons ces notions ci-dessous.

1.2.1 Arbres d'arité bornée

On désigne par $\mathbb{N}$ l'ensemble des entiers naturels positifs et par $\mathbb{N}_0$ l'ensemble $\mathbb{N} \cup \{0\}$. L'ensemble des chaînes d'entiers positifs est dénoté par $\mathbb{N}^*$. L'élément v' est un *préfixe* de v, et l'on note $v' \preceq v$, s'il existe $v'' \in \mathbb{N}^*$ tel que $v = v' \cdot v''$. Un sous-ensemble fini $N \subseteq \mathbb{N}^*$ est *clos par préfixe* si quels que soient $v, v' \in \mathbb{N}^*$ tels que $v' \preceq v$, si $v \in N$ alors $v' \in N$. Autrement dit, un ensemble clos par préfixe contient l'ensemble des préfixes de tous ses éléments.

Un *alphabet d'arité bornée* est un couple (Σ, ar) où Σ est un ensemble fini (non vide) de symboles et ar une fonction de Σ dans $\mathbb{N}_0$, l'ensemble des entiers naturels. Soit f un élément de Σ, $ar(f)$ désigne l'*arité* de f. Un symbole d'arité 0 est appelé *constante*, d'arité 1 *monadique* (ou *unaire*), d'arité 2 *binaire* et, enfin, un symbole d'arité $n > 2$ est appelé symbole n-*aire*. Dans la suite, un alphabet d'arité bornée (Σ, ar) sera simplement noté Σ lorsque le contexte le permet.

L'ensemble des *arbres d'arité bornée* (ou *termes clos*) sur (Σ, ar), noté T^r_Σ, est le plus petit ensemble tel que $f(t_1, \ldots, t_n) \in T^r_\Sigma$ si $f \in \Sigma$, $ar(f) = n$ et $t_1, \ldots, t_n \in T^r_\Sigma$. Un *langage*

d'arbres d'arité bornée sur Σ est un sous-ensemble de T_Σ^r. Les *arbres binaires* sur Σ sont des arbres d'arité bornée où les symboles de Σ sont soit d'arité 0, soit d'arité 2 ; on note plus spécifiquement T_Σ^{bin} un ensemble d'arbres binaires sur Σ.

Formellement, un arbre fini, ordonné et étiqueté $t \in T_\Sigma^r$ définit une fonction (partielle) $t : \mathtt{nod}(t) \to \Sigma$, où $\mathtt{nod}(t) \subseteq \mathbb{N}^*$ est l'ensemble fini non vide clos par préfixe, appelé *domaine*, tel que :

$$\mathtt{nod}(f(t_1, \ldots, t_n)) = \{\epsilon\} \cup \{i \cdot v \mid v \in \mathtt{nod}(t_i)\}$$

Un élément v de $\mathtt{nod}(t)$ est communément appelé *nœud* (ou *position*), et un symbole de Σ est aussi appelé *étiquette*. Le nœud $v \cdot i$, $i \in \mathbb{N}$, s'il est défini, est le i-ème *fils* (ou *enfant*) du nœud v, les nœuds $v \cdot j$ avec $j \neq i$ sont les *frères* du nœud $v \cdot i$; le nœud v est le *père* de tous les nœuds $v \cdot i$, $i \in \mathbb{N}$. La relation d'*ancêtre* entre deux nœuds v et v' de t est notée $v' \prec v$ si $v' \preceq v$ et $v' \neq v$. On écrit $t(v)$ pour désigner l'étiquette associée au nœud v. L'élément $t(\epsilon)$ est appelé *racine* de l'arbre. Les nœuds de t qui n'ont pas de fils sont les *feuilles*. Un *sous-arbre* $t_{|v}$ de t est un arbre enraciné en v. La hauteur de t, notée $\mathtt{height}(t)$, est donnée par :

$$\mathtt{height}(t) = \begin{cases} 0 & \text{si } t = a \text{ est une feuille} \\ 1 + \max\left\{\mathtt{height}(t_i) \mid 1 \leqslant i \leqslant n\right\} & \text{si } t = f(t_1, \ldots, t_n) \end{cases}$$

Deux arbres t_1 et t_2 sur Σ sont *égaux*, et l'on note $t_1 = t_2$, si $\mathtt{nod}(t_1) = \mathtt{nod}(t_2) = d$ et pour tout nœud v de d on a $t_1(v) = t_2(v)$. Soit $N \subseteq \mathbb{N}^*$ un ensemble (clos par préfixe) quelconque de nœuds, on dénote par $=_N$ l'égalité entre les ensembles de nœuds de deux arbres telle que $\mathtt{nod}(t_1) =_N \mathtt{nod}(t_2)$ si et seulement si $\mathtt{nod}(t_1) \cap N = \mathtt{nod}(t_2) \cap N$. Notons que N n'est pas nécessairement un domaine.

Un *contexte* C sur Σ est un arbre sur $\Sigma \uplus \{o_1, \ldots, o_p\}$ où $o_1, \ldots, o_p$ sont des symboles qui ne peuvent se trouver qu'aux feuilles de C et qu'on appelle souvent *trous*. La substitution des trous $o_1, \ldots, o_p$ par des arbres $t_1, \ldots, t_p$ sur Σ est dénotée $C[t_1, \ldots, t_p]$ et définit un arbre sur Σ.

On représente graphiquement les arbres sur Σ par un graphe étiqueté (orienté) dont les sommets correspondent aux symboles de Σ. Par convention, la racine est placée en haut de la représentation ; les flèches, toujours orientées de haut en bas, sont laissées implicites. La figure 8 illustre l'ensemble des notions sur les arbres abordées jusqu'ici.

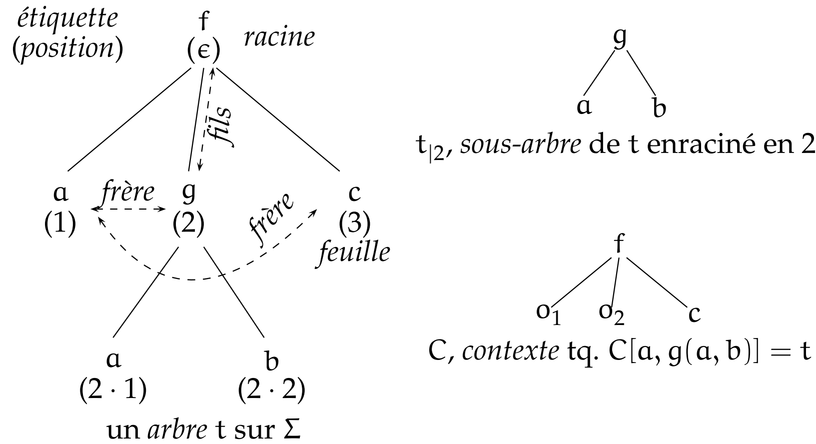


Figure 8. *Représentation graphique d'un arbre d'arité bornée sur* $\Sigma = \{f, g, a, b, c\}$ *avec* $ar(f) = 3$, $ar(g) = 2$ *et* $ar(a) = ar(b) = ar(c) = 0$. *Les positions des étiquettes sont précisées entre parenthèses (elles sont implicites dans toute la suite). Un exemple pour chacune des relations* fils, frère *et* feuille *est montré dans l'arbre. Sur la droite, nous représentons les notions de* sous-arbre *et* contexte, *qui sont aussi des arbres.*

1.2.2 Arbres d'arité non bornée

Un *alphabet d'arité non bornée* est un ensemble fini de symboles. Soit Σ un alphabet d'arité non bornée. L'ensemble des *arbres d'arité non bornée* sur Σ, noté T^u_Σ, est le plus petit ensemble tel que $f(t_1, \ldots, t_k) \in T^u_\Sigma$ si $f \in \Sigma$, $t_1, \ldots, t_k \in T^u_\Sigma$ et $k \geqslant 0$. Un *langage d'arbres d'arité non bornée* sur Σ est un sous-ensemble de T^u_Σ. Une *haie* sur Σ est une séquence d'arbres $(t_1, \ldots, t_k)$ avec $t_i \in T^u_\Sigma$ pour $1 \leqslant i \leqslant k$; $()$ désigne la haie vide.

Les notions de domaine, nœud, etc., définies pour les arbres d'arité bornée ne tiennent pas compte de la nature bornée des arbres. De ce fait, elles se transposent aux arbres d'arité non bornée. La différence essentielle tient du fait que les symboles dans les arbres d'arité bornée sont munis d'une fonction qui fixe leur arité. Cela a des répercussions principalement au niveau des automates que l'on peut définir pour opérer sur ces structures.

1.2.3 Codages binaires

Les codages binaires permettent de relier des arbres d'arité bornée ou non bornée à des arbres binaires de façon systématique. L'intérêt des codages est que de nombreux résultats de la littérature sont montrés pour les arbres binaires avant d'être généralisés aux arbres d'arité arbitraire (bornée ou non). Nous nous intéressons tout particulièrement

ici au codage curryfié, et aurons également l'occasion d'évoquer le codage frère-fils, plus classique. D'autres codages, que nous n'aborderons pas ici, ont été proposés (voir p. ex. Milo *et al.*, 2003; Frick *et al.*, 2003).

Codage curryfié

Le *codage curryfié* repose, comme son nom l'indique, sur l'opération de curryfication (Curry et Feys, 1958), qui consiste à transformer une fonction prenant plusieurs arguments en une chaîne de fonctions à un seul argument chacune. L'idée est donc de voir un arbre $f(t_1, \dots, t_n)$ comme la fonction f appliquée aux arguments $t_1, \dots, t_n$, qui sont eux-mêmes des fonctions, et d'appliquer la curryfication inductivement.

Plus formellement, nous définissons ci-dessous la curryfication comme la bijection $\mathrm{curry}^u : T^u_\Sigma \to T^{\mathrm{bin}}_{\Sigma_@}$ où $\Sigma_@ = \Sigma \uplus \{@\}$ est l'alphabet d'arité bornée tel que les symboles de Σ sont des constantes et @ est l'unique symbole binaire, appelé *opérateur d'extension*. Nous utiliserons parfois la notation infixe $t_1@t_2$ au lieu de $@(t_1, t_2)$, et l'on omettra les parenthèses sachant que l'opérateur @ est prioritaire à gauche, autrement dit nous noterons $t_1@t_2@t_3$ pour $(t_1@t_2)@t_3$. Pour $f \in \Sigma$ et $t_1, \dots, t_n \in T^u_\Sigma$, on définit :

$$\mathrm{curry}^u(f(t_1, \dots, t_n)) = f@\mathrm{curry}^u(t_1)@\dots@\mathrm{curry}^u(t_n)$$

Par exemple, $\mathrm{curry}^u(f(a, g(a, b), c))$ donne $f@a@(g@a@b)@c$, ce qui correspond à la notation infixe de l'arbre binaire $@(@(@(f, a), @(@(g, a), b)), c)$, comme illustré à la figure 9.

La bijection $\mathrm{curry}^r : T^r_\Sigma \to T^{\mathrm{bin}}_{\Sigma_@}$ pour des arbres d'arité bornée sur Σ est définie de façon analogue.

Codage frère-fils Le *codage frère-fils* (*first child-next sibling* en anglais) de Rabin (Rabin, 1969; Koch, 2003), plus largement répandu dans la littérature, est défini par la fonction $\mathrm{fc\text{-}ns}^u : T^u_\Sigma \to T^{\mathrm{bin}}_{\Sigma_\sharp}$ où $\Sigma_\sharp = \Sigma \uplus \{\sharp\}$ est l'alphabet d'arité bornée tel que tous les symboles de Σ sont d'arité 2 et $\sharp$ est l'unique constante. Pour $t \in T^u_\Sigma$, on a $\mathrm{fc\text{-}ns}^u(t) = \mathrm{fc\text{-}ns}^{u\prime}((t))$ avec :

$$\mathrm{fc\text{-}ns}^{u\prime}(h) = \begin{cases} \sharp & \text{si } h = () \, ; \\ f(\mathrm{fc\text{-}ns}^{u\prime}(h'), \mathrm{fc\text{-}ns}^{u\prime}((t_1, \dots, t_n))) & \text{si } h = (f(h'), t_1, \dots, t_n). \end{cases}$$

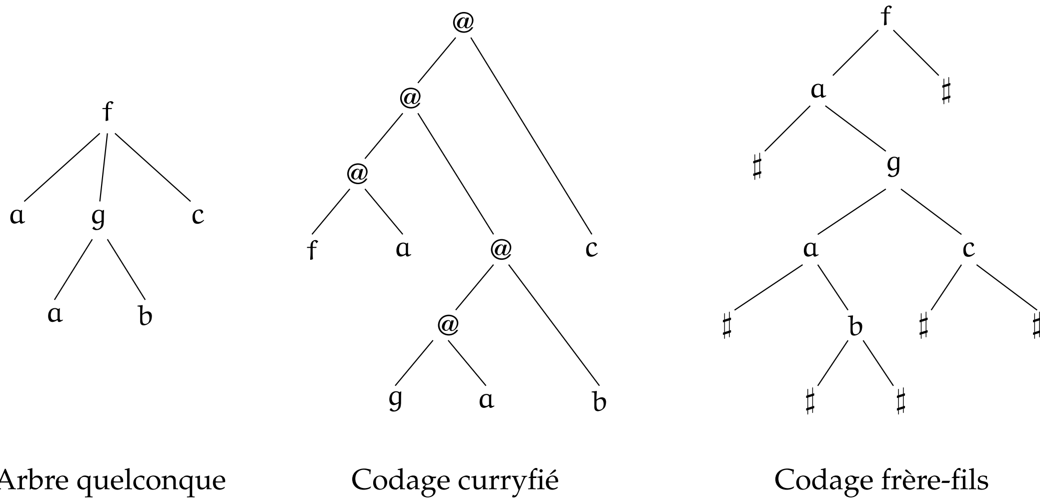


Figure 9. *Codages binaires.*

Par exemple, $\mathrm{fc\text{-}ns}^u(f(a, g(a, b), c))$ donne $f(a(\sharp, g(a(\sharp, b(\sharp, \sharp)), c(\sharp, \sharp))), \sharp)$ (cf. fig. 9).

Le codage frère-fils d'arbres d'arité bornée sur Σ est obtenu de façon analogue avec une fonction $\mathrm{fc\text{-}ns}^r : T^r_\Sigma \to T^{bin}_{\Sigma_\sharp}$.

1.3 Automates d'arbres

Les automates d'arbres sont des modèles de calcul à états finis qui reconnaissent des ensembles arbres (Comon *et al.*, 2007). Ils ont été introduits à la fin des années 60, par Thatcher et Wright (1968) et Doner (1970) notamment, en lien avec les procédures de décision de la logique monadique du second ordre (MSO), et leur développement a surtout concerné les arbres d'arité bornée. Les nombreux développements autour du langage XML ont conduit les chercheurs à s'intéresser à des modèles d'automates pour les arbres d'arité non bornée (Neven, 2002a,b; Schwentick, 2007). Nous considérons d'abord les automates d'arbres d'arité bornée, puis abordons des modèles pour les arbres d'arité non bornée. Enfin, nous introduisons les automates *stepwise* factorisés, un nouveau modèle d'automates destiné à représenter les DTD.

1.3.1 Automates pour les arbres d'arité bornée

Automates standards

Soit (Σ, ar) un alphabet d'arité bornée. Un *automate d'arbres* sur Σ est un triplet $\big(\mathrm{sta}(A), \mathrm{fin}(A), \mathrm{rul}(A)\big)$ qui consiste en un ensemble fini d'*états* $\mathrm{sta}(A)$, un sous-ensemble d'*états finaux* $\mathrm{fin}(A) \subseteq \mathrm{sta}(A)$, et un ensemble fini de *règles* $\mathrm{rul}(A) \subseteq \big(\bigcup_{n \geqslant 0}\{f \in \Sigma \mid \mathrm{ar}(f) = n\} \times \mathrm{sta}(A)^{n+1}\big) \uplus \mathrm{sta}(A)^2$.

Les automates standards opèrent sur les arbres soit de façon ascendante, c'est-à-dire depuis les feuilles jusqu'à la racine, soit de façon descendante, autrement dit depuis la racine jusqu'aux feuilles. Nous examinons ces deux perspectives.

Dans la vision ascendante, les règles sont de la forme $f(p_1, \ldots, p_n) \to p$ ou $p' \xrightarrow{\epsilon} p$, où $f \in \Sigma$, $\mathrm{ar}(f) = n$ et $p_1, \ldots, p_n, p, p' \in \mathrm{sta}(A)$. Les règles $p' \xrightarrow{\epsilon} p \in \mathrm{rul}(A)$ sont appelées ϵ-*règles* (*epsilon-règles*), et l'on note $p' \xrightarrow{\epsilon}_A p$ si et seulement si $p' \xrightarrow{\epsilon} p \in \mathrm{rul}(A)$; la notation $\xrightarrow{\epsilon}{}^*_A$ correspond à la clôture transitive de la relation $\xrightarrow{\epsilon}_A$, et $\xrightarrow{\epsilon}{}^{\leqslant 1}_A$ dénote l'union de $\xrightarrow{\epsilon}_A$ et de l'identité sur $\mathrm{sta}(A)$.

La taille des règles de la forme $f(p_1, \ldots, p_n) \to p$ est $n + 2$, celle des ϵ-règles est 2 ; la *taille* d'un automate A, dénotée $|A|$, est donnée par la somme des tailles de toutes les règles de A plus la taille de $\mathrm{sta}(A)$, notée $|\mathrm{sta}(A)|$, qui correspond simplement à la cardinalité de cet ensemble. La taille de A ne dépend pas de la taille de l'alphabet, $|\Sigma|$, car les algorithmes que nous considérons ne prennent en compte que les symboles utilisés dans les règles de l'automate.

Tout automate A sur Σ définit une *fonction d'évaluation* $\mathrm{eval}_A : T^r_\Sigma \to 2^{\mathrm{sta}(A)}$ telle que :

$$\mathrm{eval}_A(f(t_1, \ldots, t_n)) = \{\, p \mid\ p_i \in \mathrm{eval}_A(t_i) \text{ pour } 1 \leqslant i \leqslant n,\ f(p_1, \ldots, p_n) \to p' \in \mathrm{rul}(A) \wedge p' \xrightarrow{\epsilon}{}^*_A p \,\}$$

Un arbre $t \in T^r_\Sigma$ est *reconnu* (ou *accepté*) par A si $\mathrm{eval}_A(t) \cap \mathrm{fin}(A) \neq \emptyset$. Le *langage* de A, dénoté $\mathcal{L}(A)$, est l'ensemble des arbres reconnus par l'automate A, c'est-à-dire :

$$\mathcal{L}(A) = \{t \in T^r_\Sigma \mid \mathrm{eval}_A(t) \cap \mathrm{fin}(A) \neq \emptyset\}$$

Deux automates d'arbres A et A' sont dits *équivalents* s'ils définissent le même langage, c'est-à-dire $A = A'$ si et seulement si $\mathcal{L}(A) = \mathcal{L}(A')$.

Déterminisme ascendant Un automate d'arbres est *déterministe ascendant* s'il n'a pas d'ϵ-règle et si toutes ses règles ont une partie gauche distincte ; il est *complet* s'il existe une règle pour chaque partie gauche potentielle. Tout automate d'arbres non déterministe peut être rendu déterministe ascendant en temps exponentiel (Comon *et al.*, 2007).

L'ensemble des langages d'arbres reconnaissables par automate définit la classe des *langages réguliers d'arbres*. Celle-ci est close par union, intersection et complément (Comon *et al.*, 2007, d'après Doner (1970) et Thatcher et Wright (1968)). Ci-dessous, nous donnons des constructions effectives pour ces opérations de clôture.

Complément Un automate complet et déterministe (ascendant) peut être complémenté en temps linéaire en inversant ses états finaux ; plus formellement, si A est un automate sur Σ, $\mathcal{L}(A)$ son langage, alors le complémentaire de $\mathcal{L}(A)$, dénoté $\mathcal{L}(A)^c$, équivaut à $\mathcal{L}(A^c)$ avec A^c défini sur Σ, $\mathtt{sta}(A^c) = \mathtt{sta}(A)$, $\mathtt{fin}(A^c) = \mathtt{sta}(A) \setminus \mathtt{fin}(A)$ et $\mathtt{rul}(A^c) = \mathtt{rul}(A)$.

Notons que complémenter un automate non déterministe nécessite au préalable de le déterminiser puis de le compléter, ce qui implique une explosion exponentielle. Par ailleurs, la complétion d'un automate déterministe A sur Σ s'effectue dans le pire des cas en temps $\mathcal{O}(|\Sigma| \times |\mathtt{sta}(A)|^n)$, où n est l'arité maximale des symboles de Σ, car il est nécessaire d'ajouter une règle pour chaque partie gauche possible.

Intersection Soient A et A' deux automates d'arbres (sans ϵ-règles) sur un même alphabet Σ. L'intersection $\mathcal{L}(A) \cap \mathcal{L}(A')$, ainsi que l'union $\mathcal{L}(A) \cup \mathcal{L}(A')$, reposent sur le calcul d'un *automate produit* de A et A', noté $A \times A'$. Les états de ce dernier sont $\mathtt{sta}(A) \times \mathtt{sta}(A')$ et ses règles sont obtenues comme suit :

$$\frac{f(p_1, \ldots, p_n) \to p \in \mathtt{rul}(A) \qquad f(p'_1, \ldots, p'_n) \to p' \in \mathtt{rul}(A')}{f((p_1, p'_1), \ldots, (p_n, p'_n)) \to (p, p') \in \mathtt{rul}(A \times A')}$$

Pour calculer l'intersection $\mathcal{L}(A) \cap \mathcal{L}(A')$, on définit les états finaux par :

$$\mathtt{sta}(A \times A') = \mathtt{fin}(A) \times \mathtt{fin}(A')$$

Notons que $A \times A'$ est déterministe si A et A' le sont.

Union Pour l'union $\mathcal{L}(A) \cup \mathcal{L}(A')$, on suppose en plus A et A' complets et déterministes, et les états finaux de l'automate produit sont définis par :

$$\mathtt{fin}(A \times A') = \mathtt{fin}(A) \cup \mathtt{sta}(A') \times \mathtt{sta}(A) \cup \mathtt{fin}(A')$$

Cette construction préserve également le déterminisme.

Tableau 1. *Complexité de quelques problèmes de décision, où A et B sont deux automates d'arbres (descendants) sur Σ (fixé), et $t \in T^r_\Sigma$ un arbre quelconque.*

Problème	Entrée	Sortie	Complexité
Appartenance	t, A	$t \in \mathcal{L}(A)$?	– $\mathcal{O}(\lvert t\rvert + \lvert A\rvert)$ si A est déterministe ; – $\mathcal{O}(\lvert t\rvert \times \lvert A\rvert)$ sinon
Vide	A	$\mathcal{L}(A) = \emptyset$?	$\mathcal{O}(\lvert A\rvert)$
Universalité	A	$\mathcal{L}(A) = T^r_\Sigma$?	– PTIME si A est déterministe ; – EXPTIME-complet sinon
Équivalence	A, B	$\mathcal{L}(A) = \mathcal{L}(B)$?	– PTIME si A et B sont déterministes ; – EXPTIME-complet sinon
Inclusion	A, B	$\mathcal{L}(A) \subseteq \mathcal{L}(B)$?	– PTIME si B est déterministe ; – EXPTIME-complet sinon

Nous listons dans le tableau 1 certains problèmes de décision usuels pour les automates d'arbres finis. Nous reviendrons en particulier sur l'inclusion pour deux automates d'arbres A et B sur Σ au cours du chapitre suivant, où nous présentons un algorithme qui permet de la vérifier en temps $\mathcal{O}(|A| \times |B|)$ si B est déterministe ascendant, indépendamment de la taille de Σ.

Comme dans le cas des langages réguliers de mots, il existe un unique automate (déterministe) minimal en nombre d'états pour un langage d'arbres reconnaissable (Comon *et al.*, 2007).

Déterminisme descendant Les automates d'arbres discutés au-dessus sont ascendants dans le sens où l'évaluation d'un arbre s'effectue de bas en haut, c'est-à-dire depuis les feuilles jusqu'à la racine. Dans la vision descendante, les automates opèrent au contraire à partir de la racine et parcourent l'arbre jusqu'aux feuilles. Automates ascendants et descendants ont la même expressivité. Cependant, du point de vue opératoire, les automates d'arbres descendants déterministes sont strictement moins expressifs que les non déterministes et, *a fortiori*, moins expressifs que les automates d'arbres ascendants (déterministes ou non). En effet, ils ne peuvent exprimer que des propriétés de chemin.

Formellement, un automate d'arbres pour des arbres d'arité bornée A sur Σ qui opère en descendant est syntaxiquement équivalent à un automate qui opère en ascendant. Toutefois, afin de les distinguer, nous employons quelques notations différentes. L'ensemble des états finaux est remplacé par un ensemble d'états initiaux, noté $\mathtt{init}(A)$, par lesquels débutent l'évaluation. Par ailleurs, les règles n-aires sont notées $p \xrightarrow{f} (p_1, \ldots, p_n)$ au lieu

de $f(p_1, \ldots, p_n) \to p$; les règles pour les constantes sont notées $p \xrightarrow{a} ()$. Un arbre $t \in T^r_\Sigma$ est reconnu par A si $eval_A(t) \cap init(A) \neq \emptyset$. Le langage de A est donné par :

$$\mathcal{L}(A) = \{t \in T^r_\Sigma \mid eval_A(t) \cap init(A) \neq \emptyset\}$$

L'automate d'arbres A est *déterministe descendant* s'il n'a pas d'ϵ-règle, s'il n'a qu'un seul état initial et si pour tout état $p \in sta(A)$ et tout symbole $f \in \Sigma$ d'arité n il existe au plus une règle $p \xrightarrow{f} (p_1, \ldots, p_n)$ dans $rul(A)$.

Pour voir qu'un automate déterministe descendant est moins expressif qu'un automate non déterministe, on considère par exemple le langage régulier (fini) $L_0 = \{f(a, a), f(b, b)\}$ avec $a \neq b$. Il n'est pas possible de définir un automate déterministe descendant qui reconnaisse L_0. En effet, après avoir lu la racine f, l'évaluation du sous-arbre gauche dépend de celle du sous-arbre droit ; or, un automate déterministe descendant ne peut disposer, pour évaluer un sous-arbre, que d'informations provenant d'ancêtres de ce sous-arbre. Le non déterminisme permet d'énumérer les différents cas de figure que le déterminisme empêche.

La classe de langages reconnus par les automates d'arbres descendant déterministes correspond à la classe des langages clos par chemin (Virágh, 1980; Comon *et al.*, 2007) [3].

Définition 1.2 (clôture par chemin)
L'ensemble des chemins d'un arbre $t \in T^r_\Sigma$ est le sous-ensemble de mots dénoté $paths(t) \subseteq (\Sigma \cup \mathbb{N})^*$ et défini inductivement comme suit :

$$paths(f(t_1, \ldots, t_n)) = \begin{cases} \{f\} \text{ si } n = 0 \text{ (f est une feuille)} ; \\ \{f \cdot i \cdot w \mid w \in paths(t_i) \text{ pour } 1 \leqslant i \leqslant n\} \text{ sinon.} \end{cases}$$

Par extension, l'ensemble des chemins $paths(L)$ d'un langage $L \subseteq T^r_\Sigma$ est l'union des chemins de tous les arbres de L, c'est-à-dire $paths(L) = \bigcup_{t \in L} paths(t)$.

La *clôture par chemin* d'un langage d'arbres $L \subseteq T^r_\Sigma$ est l'ensemble des arbres qui contiennent seulement des chemins d'arbres dans L :

$$path\text{-}clos(L) = \{t \mid paths(t) \subseteq paths(L)\}$$

Un langage $L \subseteq T^r_\Sigma$ est *clos par chemin* si $L = path\text{-}clos(L)$. ◀

3. Pour la petite histoire, les automates descendant sont appelés "ascendant" par Virágh (1980) et d'autres auteurs, ceux-ci considèrant assez logiquement que la racine d'un arbre se situe en bas et ses feuilles en haut.

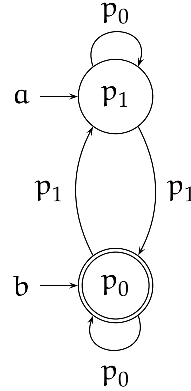


Figure 10. *Représentation graphique d'un automate stepwise (déterministe)* A *sur* $\Sigma = \{a, b\}$, *avec* $\mathtt{sta}(A) = \{p_0, p_1\}$, $\mathtt{fin}(A) = \{p_0\}$ *et* $\mathtt{rul}(A) = \{a \to p_1, b \to p_0, p_1@p_0 \to p_1, p_1@p_1 \to p_0, p_0@p_1 \to p_1, p_0@p_0 \to p_0\}$. *L'automate* A *reconnaît l'ensemble des arbres sur* $\Sigma_@$ *qui ont un nombre pair de feuilles étiquetées par le symbole* a, *p. ex. les arbres* $a@(a@b)$ *et* $b@b@b$ *sont reconnus par* A *tandis que* $b@a$ *ne l'est pas.*

Un langage régulier d'arbres non clos par chemin est par exemple $L_0 = \{f(a, a), f(b, b)\}$, avec $a \neq b$. On a $\mathtt{paths}(L_0) = \{f \cdot 1 \cdot a, f \cdot 2 \cdot a, f \cdot 1 \cdot b, f \cdot 2 \cdot b\}$ et donc $\mathtt{path\text{-}clos}(L_0) = L_0 \cup \{f(a, b), f(b, a)\} \neq L_0$.

Proposition 1.3
Soit A *un automate d'arbres déterministe descendant. Alors le langage* $\mathcal{L}(A)$ *est clos par chemin.*

Automates *stepwise*

Un *automate d'arbres stepwise* (Carme *et al.*, 2004) sur Σ est un automate d'arbres sur l'alphabet $\Sigma_@ = \Sigma \uplus \{@\}$. On note $a \to p$ les règles pour les constantes $a \in \Sigma$, et $p_1@p_2 \to p$ au lieu de $@(p_1, p_2) \to p$ les règles pour l'unique symbole binaire @ ; on appelle ces règles *règles constantes* et *règles binaires*, respectivement. Notons que la taille des règles binaires est de 3 car le symbole @ peut être laissé implicite. La notation des ϵ-règles n'est pas affectée.

On utilise pour les automates *stepwise* une représentation graphique proche de celle des automates de mots. Les règles constantes $a \to p$ sont représentées par une transition non étiquetée d'un sommet a vers un sommet p ; on distingue les sommets symboles des sommets états en n'encerclant pas les premiers. Les règles binaires $p_1@p_2 \to p$ sont représentées par une arête de p_1 vers p étiquetée par p_2, les ϵ-règles $p' \xrightarrow{\epsilon} p$ par une

arête de p' vers p étiquetée par ϵ. Les états finaux sont entourés par un double cercle. Un exemple est donné figure 10.

Les automates *stepwise* peuvent être interprétés sur les arbres d'arité bornée *via* le codage curryfié. Soit A un automate d'arbres sur Σ. Nous transformons A en un automate *stepwise* $\mathsf{s}(A)$ sur $\Sigma_@$ de telle sorte que le langage est préservé avec la curryfication, c'est-à-dire de sorte que pour tout $t \in T^r_\Sigma$:

$$t \in \mathcal{L}(A) \Leftrightarrow \mathtt{curry}^r(t) \in \mathcal{L}(\mathsf{s}(A))$$

Les états de $\mathsf{s}(A)$ sont les préfixes des parties gauches des règles de A, autrement dit des chaînes dans $\Sigma \cdot (\mathtt{sta}(A))^* \uplus \mathtt{sta}(A)$:

$$\mathtt{sta}(\mathsf{s}(A)) = \{f \cdot p_1 \cdot \dots \cdot p_i \mid f(p_1, \dots, p_n) \to p \in \mathtt{rul}(A) \text{ pour } 0 \leqslant i < n\} \uplus \mathtt{sta}(A)$$

Les états finaux sont préservés, c'est-à-dire $\mathtt{fin}(\mathsf{s}(A)) = \mathtt{fin}(A)$. Les règles de $\mathsf{s}(A)$ sont données à la figure 11. Elles étendent les préfixes pas à pas par les états p_i selon les règles de A. Comme les constantes ne peuvent pas être étendues, il est nécessaire de distinguer les deux cas. Une règle n-aire de A, de taille $n+2$, produit dans $\mathsf{s}(A)$ une règle constante et n règles binaires de taille 3 moyennant un renommage des états. Un exemple simple de transformation est montré à la figure 12.

Lemme 1.4
La transformation d'un automate d'arbres A sur Σ en un automate d'arbres stepwise $\mathsf{s}(A)$ sur $\Sigma_@$ préserve le déterminisme ascendant, le langage d'arbres modulo curryfication, et la taille avec un facteur constant de 3.

Les automates *stepwise* interprétés sur les arbres d'arité bornée héritent par définition des propriétés de reconnaissabilité et de clôture des automates d'arbres standards.

En conséquence du lemme 1.4, décider l'inclusion $\mathcal{L}(A) \subseteq \mathcal{L}(B)$ pour deux automates d'arbres d'arité bornée A et B équivaut à décider $\mathcal{L}(\mathsf{s}(A)) \subseteq \mathcal{L}(\mathsf{s}(B))$ moyennant une transformation en temps linéaire. Cette observation est également valable pour les autres problèmes de décision usuels (appartenance, vide, universalité, etc.).

$$\frac{a \to p \in \mathrm{rul}(A)}{a \to p \in \mathrm{rul}(\mathsf{s}(A))} \qquad \frac{p' \xrightarrow{\epsilon}_A p}{p' \xrightarrow{\epsilon}_{\mathsf{s}(A)} p}$$

$$\frac{f(p_1, \ldots, p_n) \to p \in \mathrm{rul}(A) \qquad 1 \leqslant i < n}{\begin{array}{rcl} f & \to & f \in \mathrm{rul}(\mathsf{s}(A)) \\ f \cdot p_1 \cdot \cdots \cdot p_{i-1} @ p_i & \to & f \cdot p_1 \cdot \cdots \cdot p_i \in \mathrm{rul}(\mathsf{s}(A)) \\ f \cdot p_1 \cdot \cdots \cdot p_{n-1} @ p_n & \to & p \in \mathrm{rul}(\mathsf{s}(A)) \end{array}}$$

Figure 11. *Transformation de l'automate d'arbres d'arité bornée* A *dans l'automate stepwise* $\mathsf{s}(A)$.

$f(p_a, p_g, p_c) \to p_f$	$f \to f$ $f @ p_a \to f \cdot p_a$ $f \cdot p_a @ p_g \to f \cdot p_a \cdot p_g$ $f \cdot p_a \cdot p_g @ p_c \to p_f$
$g(p_a, p_b) \to p_b$	$g \to g$ $g @ p_a \to g \cdot p_a$ $g \cdot p_a @ p_b \to p_g$
$a \to p_a$ $b \to p_b$ $c \to p_c$	$a \to p_a$ $b \to p_b$ $c \to p_c$

Figure 12. *Exemple de transformation d'un automate d'arbres d'arité bornée (à gauche) en un automate stepwise reconnaissant le même langage modulo curryfication (à droite).*

Automates frère-fils

Nous définissons les automates dits frères-fils similairement aux *stepwise* mais destinés à être interprétés avec le codage frère-fils dans une perspective descendante.

Un *automate d'arbres frère-fils* sur Σ est un automate d'arbres sur l'alphabet $\Sigma_\sharp = \Sigma \uplus \{\sharp\}$. Les automates frère-fils peuvent être interprétés sur les arbres d'arité bornée *via* le codage frère-fils. Soit A un automate d'arbre sur Σ, l'automate A peut être transformé en automate frère-fils $\mathsf{ff}(A)$ sur $\Sigma_\sharp$ de telle sorte que le langage est préservé avec l'encodage, autrement dit pour tout $t \in T^r_\Sigma$:

$$t \in \mathcal{L}(A) \Leftrightarrow fc\text{-}ns^r(t) \in \mathcal{L}(\mathsf{ff}(A))$$

Les états de $\mathsf{ff}(A)$ sont les suffixes des parties droites des règles de A, c'est-à-dire des mots dans $sta(A)^*$, où ϵ est l'état pour la constante $\sharp$:

$$sta(\mathsf{ff}(A)) = \{p_i \cdot \dots \cdot p_n \mid p \xrightarrow{f} (p_1, \dots, p_n) \in rul(A) \text{ pour } 1 \leqslant i < n\} \uplus sta(A)$$

Les états initiaux de $\mathsf{ff}(A)$ sont les mêmes que ceux de A. Les règles de $\mathsf{ff}(A)$ sont obtenues à partir de celles de A par l'ensemble des transformations données à la figure 13. Un exemple simple de conversion est montré à la figure 14.

Lemme 1.5
La transformation d'un automate d'arbres A sur Σ en un automate frère-fils $\mathsf{ff}(A)$ sur $\Sigma_\sharp$ préserve le déterminisme descendant, le langage d'arbres modulo encodage, et la taille avec un facteur constant de 4.

La construction de la figure 13 produit, à partir d'une règle n-aire de l'automate A, n règles binaires de taille 4 dans l'automate $\mathsf{ff}(A)$ moyennant un renommage des états. Cela est dû au fait que les règles binaires dans un automate frère-fils ont une taille constante de 4 car elles sont composées de trois états et d'un symbole binaire qui, contrairement au cas des *stepwise*, ne peut pas être laissé implicite.

1.3.2 Automates pour les arbres d'arité non bornée

Les développements autour du langage XML ont réactivé les investigations sur des modèles d'automates pour les arbres d'arité non bornée (Neven, 2002a,b; Schwentick, 2007). On considère essentiellement deux approches : des automates qui opèrent directement

$$\frac{p \xrightarrow{f} (p_1, \ldots, p_n) \in \mathrm{rul}(A) \qquad p \in \mathrm{init}(A)}{p \xrightarrow{f} (p_1 \cdot \cdots \cdot p_n, \epsilon) \in \mathrm{rul}(\mathrm{ff}(A))}$$

$$\frac{p \xrightarrow{f} (p_1, \ldots, p_n) \in \mathrm{rul}(A) \qquad 1 \leqslant i \leqslant n \qquad p_i \xrightarrow{g} (q_1, \ldots, q_k) \in \mathrm{rul}(A)}{p_i \cdot \cdots \cdot p_n \xrightarrow{g} (q_1 \cdot \cdots \cdot q_k, p_{i+1} \cdot \cdots \cdot p_n) \in \mathrm{rul}(\mathrm{ff}(A))}$$

$$\frac{}{\epsilon \xrightarrow{\sharp} () \in \mathrm{rul}(\mathrm{ff}(A))} \qquad \frac{p' \xrightarrow{\epsilon}_A p}{p' \xrightarrow{\epsilon}_{\mathrm{ff}(A)} p}$$

Figure 13. *Transformation de l'automate d'arité bornée A dans l'automate frère-fils $\mathrm{ff}(A)$.*

$p_f \xrightarrow{f} (p_a, p_g, p_c)$	$p_f \xrightarrow{f} (p_a \cdot p_g \cdot p_c, \epsilon)$
$p_a \xrightarrow{a} ()$	$p_a \cdot p_g \cdot p_c \xrightarrow{a} (\epsilon, p_g \cdot p_c)$
$p_g \xrightarrow{g} (p_a, p_b)$	$p_g \cdot p_c \xrightarrow{g} (p_a \cdot p_b, p_c)$
$p_b \xrightarrow{b} ()$	$p_c \xrightarrow{c} (\epsilon, \epsilon)$
$p_c \xrightarrow{c} ()$	$p_a \cdot p_b \xrightarrow{a} (\epsilon, p_b)$
	$p_b \xrightarrow{b} (\epsilon, \epsilon)$
	$\epsilon \xrightarrow{\sharp} ()$

Figure 14. *Exemple de transformation d'un automate d'arbres d'arité bornée (à gauche) en un automate frère-fils reconnaissant le même langage modulo le codage binaire (à droite), où p_f est l'unique état initial.*

sur les arbres d'arité non bornée, appelés automates *hedge* (de haies), ou bien des automates qui opèrent sur un codage binaire (curryfié ou frère-fils) de ces arbres. Ces deux approches sont équivalentes en terme d'expressivité, à savoir que toutes deux permettent de reconnaître l'ensemble des langages réguliers d'arbres d'arité non bornée, mais la notion de déterminisme est plus solide dans le cas des codages binaires.

Automates ***stepwise***

Les automates d'arbres *stepwise* peuvent être interprétés sur les arbres d'arité non bornée *via* la curryfication. Soient A un automate *stepwise*, $T^{bin}_{\Sigma_@}$ l'ensemble des arbres binaires sur l'alphabet $\Sigma_@$ et $\mathcal{L}^{bin}(A) \subseteq T^{bin}_{\Sigma_@}$ l'ensemble des arbres binaires reconnus par A. Tout arbre d'arité non bornée $t \in T^u_\Sigma$ peut être encodé par la curryfication en un arbre binaire $curry^u(t)$ sur $T^{bin}_{\Sigma_@}$. Le langage $\mathcal{L}^u(A) \subseteq T^u_\Sigma$ d'arbres d'arité non bornée reconnu par un automate *stepwise* A est l'ensemble :

$$\mathcal{L}^u(A) = \{t \in T^u_\Sigma \mid curry^u(t) \in \mathcal{L}^{bin}(A)\}$$

Si $t \in \mathcal{L}^u(A)$, on note $eval^u_A(t) = eval_A(curry^u(t))$.

Les automates *stepwise* interprétés sur les arbres d'arité non bornée héritent par définition des propriétés de reconnaissabilité et de clôture des automates d'arbres standards. En particulier, ils reconnaissent l'ensemble des langages réguliers d'arbres sur $\Sigma_@$. En corollaire, les automates *stepwise* reconnaissent tous les langages réguliers d'arbres d'arité non bornée sur Σ. Par nature, les *stepwise* disposent d'une bonne notion de déterminisme ascendant relativement au codage curryfié (cf. lemme 1.4).

Automates frère-fils

De façon analogue, les automates frère-fils peuvent interprétés sur les arbres d'arité non bornée modulo le codage frère-fils. Ils reconnaissent par définition l'ensemble des langages réguliers d'arbres sur $\Sigma_\sharp$ et donc l'ensemble des langages réguliers d'arbres d'arité non bornée sur Σ. Les automates frère-fils possèdent une bonne notion de déterminisme descendant relativement au codage frère-fils (cf. lemme 1.5).

Automates *hedge*

Les automates *stepwise* ou frère-fils interprétés sur les arbres d'arité non bornée opèrent sur un codage binaire des arbres d'arité non bornée. Les *automates hedge* (automates de haies), définis par Brüggemann-Klein *et al.* (2001), opèrent au contraire directement sur les arbres d'arité non bornée, avec un mécanisme proche des automates standards en hauteur et avec des langages réguliers de mots en largeur.

Formellement, un automate *hedge* H sur un alphabet Σ est un triplet $\big(\mathtt{sta}(H), \mathtt{fin}(H), \mathtt{rul}(H)\big)$, avec $\mathtt{sta}(H)$ un ensemble fini d'états, $\mathtt{fin}(H) \subseteq \mathtt{sta}(H)$ un sous-ensemble d'états finaux, et $\mathtt{rul}(H)$ un ensemble fini de règles de la forme $f(L) \to p$, où $f \in \Sigma$, $p \in \mathtt{sta}(H)$ et L est un langage (régulier) de mots sur $\mathtt{sta}(H)$, appelé *langage horizontal* de H. N'importe quel formalisme de langage régulier de mots peut être choisi pour spécifier L (p. ex. automates finis, expressions régulières) ; la taille de H, notée $|H|$, dépend de la représentation utilisée. On a $|H| = \sum_{f(L)\to p \in \mathtt{rul}(H)} 2 + |L|$, où $|L|$ est la taille de la représentation de L.

Tout automate *hedge* H sur Σ définit une fonction d'évaluation $eval^{\Rsh}_H : T^u_\Sigma \to 2^{\mathtt{sta}(H)}$ comme suit :

$$eval^{\Rsh}_H(t) = \begin{cases} \{p \mid a(L) \to p \in \mathtt{rul}(H) \wedge \epsilon \in L\} & \text{si } t = a \text{ est une feuille} \\ \{p \mid p_i \in eval^{\Rsh}_H(t_i) \text{ pour } 1 \leqslant i \leqslant n, & \text{si } t = f(t_1, \ldots, t_n) \\ \quad f(L) \to p \in \mathtt{rul}(H) \wedge p_1 \cdot \cdots \cdot p_n \in L\} & \end{cases}$$

Un arbre $t \in T^u_\Sigma$ est reconnu par H si $eval^{\Rsh}_H(t) \cap \mathtt{fin}(H) \neq \emptyset$. Le langage $\mathcal{L}(H)$ de l'automate H est l'ensemble des arbres reconnus par H, c'est-à-dire :

$$\mathcal{L}(H) = \{t \in T^u_\Sigma \mid eval^{\Rsh}_H(t) \cap \mathtt{fin}(H) \neq \emptyset\}$$

Un automate *hedge* H sur Σ est déterministe (ascendant) si pour toute paire de règles $a(L) \to q$ et $a(L') \to q'$, soit $L \cap L' = \emptyset$, soit $q = q'$. Similairement aux automates standards, il est possible de déterminiser tout automate *hedge* non déterministe en temps exponentiel. Un langage d'arbres d'arité non bornée est reconnaissable par automate *hedge* si et seulement s'il est reconnaissable par automate *stepwise* ou frère-fils modulo encodage binaire (cf. proposition suivante). La classe des langages d'arbres d'arité non bornée reconnus par les automates *hedge* est close par union, intersection et complément.

$$\frac{\mathsf{f}(\mathcal{L}(W)) \to \mathsf{p} \in \mathsf{rul}(\mathsf{H}) \qquad \mathsf{q}_1 \xrightarrow{\mathsf{p}'} \mathsf{q}_2 \in \mathsf{rul}(W)}{\begin{array}{rcll} \mathsf{f} & \to & \mathsf{f} \in \mathsf{rul}(\mathsf{s}(\mathsf{H})) & \\ \mathsf{f}@\mathsf{q}_1 & \to & \mathsf{p}' \in \mathsf{rul}(\mathsf{s}(\mathsf{H})) & \text{si } \mathsf{q}_1 \in \mathsf{init}(W) \\ \mathsf{q}_1@\mathsf{p}' & \to & \mathsf{q}_2 \in \mathsf{rul}(\mathsf{s}(\mathsf{H})) & \\ \mathsf{p}'@\mathsf{q}_2 & \to & \mathsf{p} \in \mathsf{rul}(\mathsf{s}(\mathsf{H})) & \text{si } \mathsf{q}_2 \in \mathsf{fin}(W) \end{array}}$$

$$\frac{\mathsf{f}(\mathcal{L}(W)) \to \mathsf{p} \in \mathsf{rul}(\mathsf{H}) \qquad \mathsf{q}' \xrightarrow{\epsilon}_W \mathsf{q}}{\mathsf{q}' \xrightarrow{\epsilon}_{\mathsf{s}(\mathsf{H})} \mathsf{q}}$$

Figure 15. *Transformation d'un automate hedge* H *en automate stepwise* $\mathsf{s}(\mathsf{H})$ *reconnaissant le même langage modulo curryfication ; l'ensemble des états de* $\mathsf{s}(\mathsf{H})$ *est* $\mathsf{sta}(\mathsf{s}(\mathsf{H})) = \Sigma \uplus \mathsf{sta}(\mathsf{H})$*, ses états finaux sont* $\mathsf{fin}(\mathsf{s}(\mathsf{H})) = \mathsf{fin}(\mathsf{H})$.

Proposition 1.6

Soit H *un automate hedge sur* Σ *dont les langages horizontaux sont représentés par des automates de mots.*

1. *On peut transformer* H *en un automate stepwise* $\mathsf{s}(\mathsf{H})$ *sur* $\Sigma_@$ *reconnaissant le même langage modulo curryfication en temps linéaire dans la taille de* H.
2. *On peut transformer* H *en un automate frère-fils* $\mathsf{ff}(\mathsf{H})$ *sur* $\Sigma_\sharp$ *reconnaissant le même langage modulo le codage frère-fils en temps linéaire dans la taille de* H.

La conversion d'un automate hedge en stepwise ou en frère-fils ne préserve ni le déterminisme ascendant, ni le déterminisme descendant.

Une preuve est donnée par Comon *et al.* (2007, chap. 8). Les constructions sont présentées figures 15 et 16 respectivement.

Contrairement au modèle des automates *stepwise* et frère-fils, les automates *hedge* ne jouissent pas d'une bonne notion de déterminisme. La définition fournit seulement une notion de pseudo-déterminisme, qui se traduit en ambiguïté dans les *stepwise*. Par ailleurs, vérifier si un automate *hedge* est complet est un problème PSPACE-complet (Comon *et al.*, 2007), par conséquent l'universalité n'est pas traitable si le nombre de langages horizontaux associé à un symbole n'est pas borné. En outre, Martens et Niehren (2007) ont montré que, étant donné un langage régulier d'arbres d'arité non bornée, il n'existe pas (en général) un unique automate *hedge* déterministe minimal qui le représente, même si les langages horizontaux sont des automates de mots déterministes.

$$\frac{p \xrightarrow{f} \mathcal{L}(W) \in \mathtt{rul}(H) \quad p \in \mathtt{init}(H) \quad q \in \mathtt{init}(W)}{p \xrightarrow{f} (q,h) \in \mathtt{rul}(\mathsf{ff}(H))} \qquad \frac{}{h \xrightarrow{\sharp} () \in \mathtt{rul}(\mathsf{ff}(H))}$$

$$\frac{p \xrightarrow{f} \mathcal{L}(W) \in \mathtt{rul}(H) \quad q_1 \xrightarrow{p'} q_2 \in \mathtt{rul}(W) \quad p' \xrightarrow{g} \mathcal{L}(W') \in \mathtt{rul}(H) \quad q' \in \mathtt{init}(W')}{q_1 \xrightarrow{g} (q', q_2) \in \mathtt{rul}(\mathsf{ff}(H))}$$

$$\frac{p \xrightarrow{f} \mathcal{L}(W) \in \mathtt{rul}(H) \quad q \in \mathtt{fin}(W)}{q \xrightarrow{\sharp} () \in \mathtt{rul}(\mathsf{ff}(H))} \qquad \frac{p \xrightarrow{f} \mathcal{L}(W) \in \mathtt{rul}(H) \quad q' \xrightarrow{\epsilon}_W q}{q' \xrightarrow{\epsilon}_{\mathsf{ff}(H)} q}$$

Figure 16. *Transformation d'un automate hedge* H *en automate frère-fils* $\mathsf{ff}(H)$ *reconnaissant le même langage modulo le codage frère-fils. Dans la perspective descendante, on considère des états initiaux* $\mathtt{init}(H)$ *plutôt que de états finaux et les règles de* H *et* $\mathsf{ff}(H)$ *sont notées* $p \xrightarrow{f} L$ *au lieu de* $f(L) \to p$ *; les états de* $\mathsf{ff}(H)$ *sont* $\mathtt{sta}(\mathsf{ff}(H)) = \mathtt{sta}(H) \uplus \{h\}$ *avec* h *l'unique état pour le fils droit de la racine, ses états initiaux sont* $\mathtt{init}(\mathsf{ff}(H)) = \mathtt{init}(H)$.

1.3.3 Automates *stepwise* factorisés

Nous introduisons la notion d'automates d'arbres *stepwise* factorisés. Ceux-ci sont des automates *stepwise* avec ϵ-règles destinés à représenter des automates *stepwise* (déterministes) de façon compacte. Comme ces derniers, ils bénéficient par définition d'une bonne notion de déterminisme ascendant.

Définition 1.7

Un *automate (stepwise) factorisé* F sur une signature Σ consiste en un automate *stepwise* avec ϵ-règles et une partition d'états $\mathtt{sta}(F) = \mathtt{sta}_1(F) \uplus \mathtt{sta}_2(F)$ telle que pour toute règle $q_1 @ q_2 \to q \in \mathtt{rul}(F)$ on a $q_1 \in \mathtt{sta}_1(F)$ et $q_2 \in \mathtt{sta}_2(F)$. ◀

On dit d'un état $q \in F$ qu'il est *de sorte* i si $q \in \mathtt{sta}_i(F)$. La sorte détermine quels états peuvent être utilisés à la i-ème position relativement au symbole binaire @ dans les règles de F. Les états de sorte 1 sont aussi appelés *états gauches* et ceux de sorte 2 *états droits*.

Tout automate factorisé F définit un automate sans ϵ-règles $\mathsf{b}(F)$ qui reconnaît le même langage. Les deux automates sont définis sur la même signature et le même ensemble d'états. Les règles de $\mathsf{b}(F)$ sont obtenues à partir de celles de F comme suit.

$$(E_1)\ \frac{a \to q \in \mathtt{rul}(F)}{a \to q \in \mathtt{rul}(\mathsf{b}(F))} \qquad (E_2)\ \frac{q_1 \xrightarrow{\epsilon}{}^{*}_F r_1 \quad q_2 \xrightarrow{\epsilon}{}^{*}_F r_2 \quad r_1 @ r_2 \to q \in \mathtt{rul}(F)}{q_1 @ q_2 \to q \in \mathtt{rul}(\mathsf{b}(F))}$$

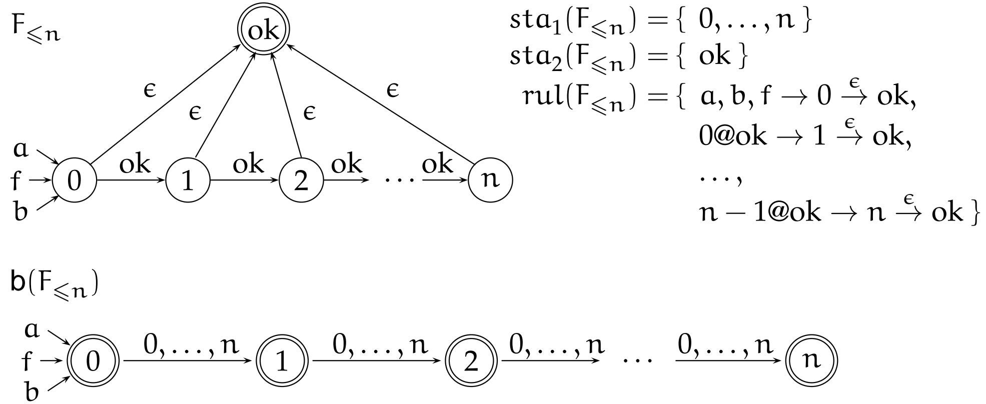


Figure 17. *Automate factorisé $F_{\leqslant n}$ et son automate $b(F_{\leqslant n})$ associé.*

Les états finaux sont donnés par $fin(b(F)) = \{q \in sta(F) \mid q \xrightarrow{\epsilon}{}^{*}_{F} r \wedge r \in fin(F)\}$. Remarquons que la taille de $b(F)$ peut être en $\mathcal{O}(|rul(F)| \times |sta(F)|^2)$, ce qui est cubique dans la taille de F dans le pire des cas.

Un exemple d'automate factorisé $F_{\leqslant n}$ est donné figure 17. L'ensemble des constantes de $F_{\leqslant n}$ est $\Sigma = \{a, b, f\}$. Cet automate reconnaît l'ensemble des arbres binaires sur $\Sigma_@$ dont la longueur des chemins sur la gauche est bornée par n. Ces arbres représentent le codage curryfié des arbres d'arité non bornée dont les nœuds ont au plus n fils. Les états de sorte 1 de $F_{\leqslant n}$ sont $\{0, \ldots, n\}$. Pour chaque nœud, ils comptent la longueur du chemin jusqu'à la feuille la plus à gauche. L'unique état de sorte 2 est l'état final ok. Il peut être assigné à tout nœud qui enracine un sous-arbre dans le langage de $F_{\leqslant n}$. Les règles de l'automate sont $a, b, f \to 0$, $i - 1@ok \to i$ et $i \xrightarrow{\epsilon} ok$ pour $1 \leqslant i \leqslant n$. La taille de $F_{\leqslant n}$ est donc en $\mathcal{O}(n)$. L'automate $b(F_{\leqslant n})$ correspondant est de taille $\mathcal{O}(n^2)$ car il a pour règles $a, b, f \to 0$ et $i - 1@j \to i$ pour $1 \leqslant i \neq j \leqslant n$. Notons que $b(F_{\leqslant n})$ est l'unique automate déterministe minimal qui reconnaît le langage de $F_{\leqslant n}$.

Outre leur compacité, l'intérêt des automates factorisés réside dans leur notion de déterminisme. L'automate $F_{\leqslant n}$ de l'exemple précédent est déterministe dans le sens suivant.

Définition 1.8
Un automate factorisé F est *déterministe (ascendant)* si les deux conditions suivantes sont satisfaites [4] :

$F \models d_0 \overset{\text{déf.}}{\Leftrightarrow}$ la partie de F exempte d'ϵ-règles est déterministe ascendant ;

4. On note $F \models \Phi$ pour signifier que l'automate F satisfait la condition Φ.

$F \models d_1 \overset{\text{déf.}}{\Leftrightarrow}$ pour tout $q \in sta(F)$ et chaque sorte $i \in \{1, 2\}$, il existe au plus un état r de sorte i tel que $q \overset{\epsilon}{\to}_F^* r$. ◀

Les ϵ-règles qui ne sont pas redondantes sont donc forcément définies sur deux états de sorte différente. Si $q \overset{\epsilon}{\to}_F r$ pour deux états de même sorte alors $q = r$ par $F \models d_0$ et $q \overset{\epsilon}{\to}_F^* q$. Un argument similaire montre que toute chaîne d'ϵ-règles correctement définies est redondante, de telle sorte que $\overset{\epsilon}{\to}_F^*$ est égal à $\overset{\epsilon}{\to}_F^{\leqslant 1}$.

Une autre conséquence du déterminisme est que la taille de $b(F)$ est au plus quadratique dans celle de F car le nombre de règles de la forme $q_1@q_2 \to q$ est borné par $|sta(F)|^2$.

Proposition 1.9
L'automate $b(F)$ *est déterministe pour tout automate factorisé déterministe* F.

Preuve
On pose $B = b(F)$. Par construction, B est exempt d'ϵ-règles. Pour tout symbole $a \in \Sigma$, l'unicité de q tel que $a \to q \in rul(B)$ découle de $F \models d_0$. Pour toute règle $q_1@q_2 \to q \in rul(B)$, on doit montrer que q est déterminé de façon unique par q_1 et q_2. Par $F \models d_1$, il existe au plus un état gauche r_1 tel que $q_1 \overset{\epsilon}{\to}_F^* r_1$ et au plus un état droit r_2 tel que $q_2 \overset{\epsilon}{\to}_F^* r_2$. La condition $F \models d_0$ implique qu'il existe au plus un état q tel que $r_1@r_2 \to q \in rul(F)$.□

Inversement, tout automate déterministe B peut être transformé en temps $\mathcal{O}(|B|)$ en un automate factorisé déterministe F tel que $b(F)$ est égal à B moyennant un renommage des états. Les états de F sont donnés par $sta_i(F) = sta(B) \times \{i\}$ pour $i \in \{1, 2\}$. Les règles de F sont obtenues à partir de celles de B comme suit.

$$\frac{a \to q \in rul(B)}{\begin{array}{c} a \to (q, 1) \in rul(F) \\ (q, 1) \overset{\epsilon}{\to} (q, 2) \in rul(F) \end{array}} \qquad \frac{q_1@q_2 \to q \in rul(B)}{\begin{array}{c} (q_1, 1)@(q_2, 2) \to (q, 1) \in rul(F) \\ (q, 1) \overset{\epsilon}{\to} (q, 2) \in rul(F) \end{array}}$$

1.4 Schémas XML et automates

Les arbres d'arité non bornée (finis et ordonnés) forment une abstraction courante pour représenter des documents XML. La figure 18 montre un exemple de document XML contenant des informations sur un pays, des régions et des villes. Sa représentation par un arbre d'arité non bornée est donnée à la figure 19. Un document XML est constitué

```
<country>
  <name>France</name>
  <city>Paris</city>
  <region>
    <name>Nord–Pas de Calais</name>
    <population>3 996 588</population>
    <city>Lille</city>
  </region>
  <region>
    <name>Vallée du Rhône</name>
    <city>Lyon</city>
    <city>Valence</city>
  </region>
</country>
```

Figure 18. *Document XML décrivant un pays, des régions et des villes.*

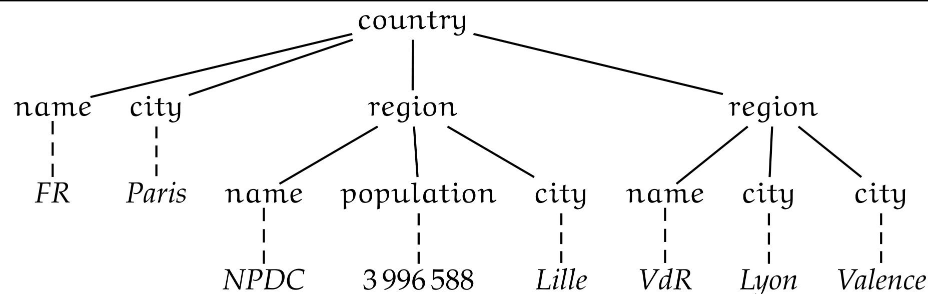


Figure 19. *Arbre d'arité non bornée décrivant le document XML de la fig. 18.*

de balises ouvrantes (p. ex. <city>) et fermantes (p. ex. </city>) bien imbriquées qui correspondent aux étiquettes et reflètent la structure de l'arbre, et de contenu textuel (p. ex. *"Paris"*) toujours situé aux feuilles. Dans la suite, on ne tiendra pas compte du contenu textuel.

Un schéma XML définit un ensemble de documents XML valides. Il permet de contraindre à la fois leur structure et le type des données qu'ils contiennent, à l'instar des schémas utilisés dans le contexte des bases de données relationnelles (Codd, 1979).

Il existe de nombreux langages de schéma XML, qui diffèrent évidemment par leurs syntaxes (Lee et Chu, 2000), mais surtout par leurs expressivités (Murata *et al.*, 2005). Les plus souvent cités sont *Document Type Definition* (définition de type de documents, DTD) (Bray *et al.*, 2008) et W3C XML Schema (XSD) (Fallside et Walmsley, 2004; Martens *et al.*, 2006) — deux langages normalisés du consortium W3C [5] —, ou encore

5. http://www.w3.org/.

Relax NG (van der Vlist, 2003) — standard développé au sein de l'association Oasis [6]. Pour une vision plus complète des langages de schéma, nous invitons le lecteur à se référer aux études détaillées de (Murata *et al.*, 2005), Schwentick (2007) et Comon *et al.* (2007, chap. 8).

Du point de vue de la théorie des langages, les schémas sont souvent représentés par des grammaires algébriques dont les arbres de dérivation représentent des langages réguliers d'arbres (Berstel et Boasson, 2002; Murata *et al.*, 2005). Ces modélisations laissent généralement de côté les aspects purement données pour se concentrer sur la structure des arbres représentant les documents XML, ce qui s'accorde avec notre perspective.

Dans le cadre spécifique de cette thèse, nous voyons un schéma comme un langage d'arbres d'arité non bornée et nous considérons des représentations par automates qui reconnaissent les arbres valides relativement au schéma. Nous nous intéressons plus particulièrement à une formalisation des DTD standards, ainsi qu'à leur version étendue (EDTD), qui permet de capturer aussi bien XML Schema que Relax NG. Nous étudions divers représentations des DTD par automates ; en particulier, nous montrons comment transformer efficacement une DTD déterministe en automate factorisé déterministe et une EDTD *restrained competition* en automate déterministe descendant.

1.4.1 Définitions de type de documents (DTD)

Les définitions de type de document (DTD) constituent le langage de schéma le plus répandu pour définir un ensemble de documents XML (Bex *et al.*, 2004). Fondamentalement, une DTD est une grammaire algébrique (*context-free*) dont les parties droites des règles sont des expressions régulières. L'ensemble des arbres de dérivation de cette grammaire définit l'ensemble des documents XML satisfaisant la DTD. Berstel et Boasson (2002), par exemple, étudient les propriétés formelles des DTD. La figure 20 présente une DTD pour représenter des pays ; le document de la figure 18 est un exemple de document conforme par rapport à celle-ci.

Formellement, une DTD D sur Σ est une fonction qui associe des éléments a d'un alphabet Σ à des expressions régulières e sur Σ, auquel cas nous écrivons $a \to_D e$. L'un des éléments de Σ est appelé *symbole initial*.

6. `http://www.oasis-open.org/`.

```
<!ELEMENT country
        (name, city , region*)>
<!ELEMENT region
    (name, population?, city*)>
<!ELEMENT name        (#PCDATA)>
<!ELEMENT city        (#PCDATA)>
<!ELEMENT population  (#PCDATA)>
```

$$
\begin{array}{lll}
\mathrm{country} & \rightarrow & \mathrm{name}\cdot\mathrm{city}\cdot\mathrm{region}^* \\
\mathrm{region} & \rightarrow & \mathrm{name}\cdot(\mathrm{population}+\epsilon)\cdot\mathrm{city}^* \\
\mathrm{name} & \rightarrow & \epsilon \\
\mathrm{city} & \rightarrow & \epsilon \\
\mathrm{population} & \rightarrow & \epsilon
\end{array}
$$

Figure 20. *Une DTD décrivant des pays (à gauche) et sa grammaire correspondante (à droite). La syntaxe W3C utilise le symbole ',' pour dénoter la concaténation et '?' pour signifier zéro ou une occurrence, c.-à-d.* `a?` *équivaut à* $a+\epsilon$. *L'élément* `#PCDATA` *correspond à du contenu textuel quelconque que nous remplaçons par* ϵ *dans la grammaire.*

Soit $\mathcal{L}(e) \subseteq \Sigma^*$ le langage de mots défini par e. Le langage $\mathcal{L}_a(D) \subseteq T^u_\Sigma$ d'un élément a d'une DTD D est le plus petit ensemble d'arbres d'arité non bornée tel que :

$$\mathcal{L}_a(D) = \{\ a(t_1,\dots,t_n) \mid t_i \in \mathcal{L}_{a_i}(D) \text{ pour } 1 \leqslant i \leqslant n,\ a_1 \cdot \dots \cdot a_n \in \mathcal{L}(e) \wedge a \rightarrow_D e\ \}$$

Le *langage* d'une DTD D est $\mathcal{L}(D) = \mathcal{L}_a(D)$ où a est le symbole initial de D. On dit qu'un arbre t *satisfait* D ou qu'il est *valide par rapport à* D si t appartient au langage de D.

La *taille* totale de D, notée $|D|$, est la somme des tailles des règles qui la composent, autrement dit $|D| = \sum_{a\rightarrow_D e}(|a| + |e|) = |\Sigma| + \sum_{a\rightarrow_D e}|e|$, en supposant que tous les éléments de Σ sont utilisés en partie gauche des règles de D.

DTD déterministes

La recommandation du W3C (Bray *et al.*, 2008) spécifie que les modèles de contenu — c.-à-d. les expressions régulières — utilisés dans les DTD doivent être déterministes. Par conséquent, une DTD est dite *déterministe* si toutes les expressions régulières qui la composent sont déterministes. C'est le cas de l'exemple de la figure 20. Le théorème suivant est une conséquence directe du théorème 1.1 (p. 25).

Théorème 1.10
La collection d'automates de Glushkov pour une DTD déterministe D *sur* Σ *peut être calculée en temps* $\mathcal{O}(|\Sigma| \times |D|)$.

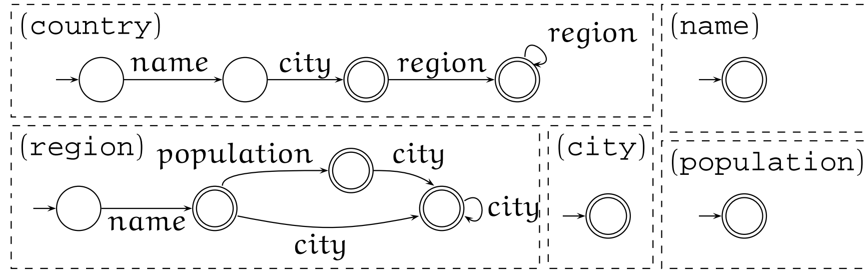


Figure 21. *Collection des automates de Glushkov pour les expressions régulières de la DTD de la fig. 20.*

Preuve
Chaque automate correspond à une règle $a \to_D e$ et est calculé en temps $\mathcal{O}(|\Sigma| \times |e|)$ (th. 1.1). La taille de la collection d'automates est donc bornée par $\sum_{a \to_D e} |\Sigma| \times |e| = |\Sigma| \times \sum_{a \to_D e} |e|$. Or $|\Sigma| \times \sum_{a \to_D e} |e| \leqslant |\Sigma| \times (|\Sigma| + \sum_{a \to_D e} |e|)$ ssi $|\Sigma| \times \sum_{a \to_D e} |e| \leqslant |\Sigma| \times |D|$.□

La figure 21 montre la collection d'automates de Glushkov pour la DTD (déterministe) de la figure 20.

DTD déterministes vers automates factorisés déterministes

Nous montrons comment convertir une DTD (déterministe) D sur un alphabet Σ en un automate factorisé déterministe F sur Σ en temps $\mathcal{O}(|\Sigma| \times |D|)$. Dans un premier temps, on calcule la collection d'automates de Glushkov de la DTD en temps $\mathcal{O}(|\Sigma| \times |D|)$ (cf. th. 1.10). Puis on construit l'automate factorisé déterministe F comme suit.

L'ensemble des états gauches de F est l'union disjointe de tous les états des automates de Glushkov des expressions régulières de D ; les états droits correspondent aux éléments de D. On note a un élément de Σ et a' son état correspondant dans $\mathrm{sta}(F)$ afin de prévenir toute ambiguïté. Pour tout élément a de Σ, on connecte tous les états finaux q de l'automate de Glushkov de l'expression régulière à laquelle il est associé à l'état a' de F, c'est-à-dire $q \xrightarrow{\epsilon} a' \in \mathrm{rul}(F)$. L'unique état final de F est celui correspondant au symbole initial de D. Pour tout élément $a \in \Sigma$, on construit une règle $a \to q_0 \in \mathrm{rul}(F)$ pour l'unique état initial q_0 de l'automate de Glushkov de l'expression régulière associée à a. Pour chaque transition $q_1 \xrightarrow{a} q_2$ présente dans les automates de Glushkov, on ajoute une règle $q_1@a' \to q_2$ à F. Cette construction s'effectue clairement en temps $\mathcal{O}(|\Sigma| \times |D|)$.

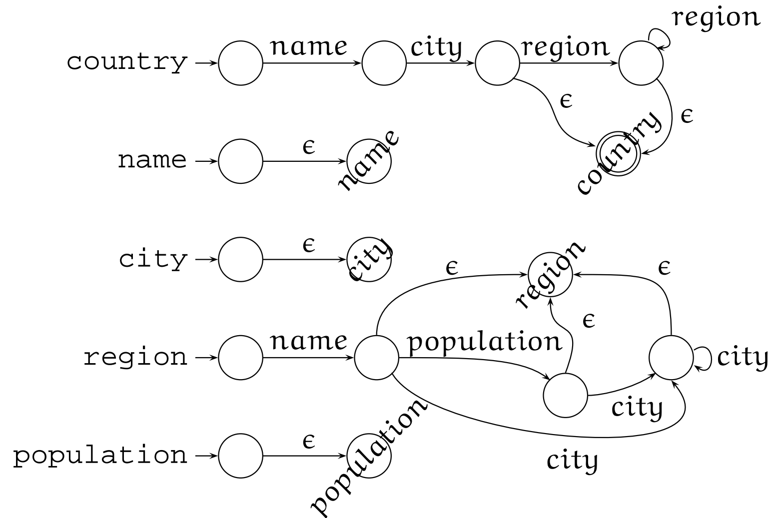


Figure 22. *Automate d'arbres factorisé déterministe reconnaissant le même langage que la DTD de la fig. 20, dont la collection d'automates de Glushkov est illustrée à la fig. 21. Les états de sorte* 1 *sont représentés sans étiquette ; les états de sorte* 2 *ont le même nom que les symboles de l'alphabet de la DTD — notés, eux, avec une police à chasse fixe.*

L'automate présenté à la figure 22 est ainsi obtenu à partir de la combinaison des automates de Glushkov correspondant aux expressions régulières de la DTD de la figure 20.

Le résultat s'apparente à une collection d'automates finis non déterministes de séquences qui représentent un automate d'arbres factorisé déterministe. En effet, la partie sans ϵ-règles de F est déterministe puisque tous les automates de Glushkov le sont, ce qui induit la condition $F \models d_0$. Soit q un état de l'automate de Glushkov pour un élément a. Le seul état de sorte 1 que q peut atteindre par ϵ-transition dans F est a_s et le seul état de sorte 2 est q lui-même ; tous les autres états de F correspondent à des éléments de Σ qui n'ont pas d'ϵ-transition sortante, d'où $F \models d_1$.

Notons que la taille de l'automate grandirait de façon quadratique si l'on éliminait les ϵ-règles. Par exemple, dans l'automate de la figure 22, chaque transition étiquetée par region se verrait remplacée par deux transitions puisque cet état de sorte 2 possède deux ϵ-transitions entrantes.

Théorème 1.11
Une DTD D sur Σ peut être transformée en temps $\mathcal{O}(|\Sigma| \times |D|)$ en un automate factorisé déterministe (ascendant) qui reconnaît le même langage.

Preuve
La traduction de la collection d'automates de Glushkov d'une DTD en un automate factorisé se fait en temps linéaire. On vérifie facilement qu'elle préserve les langages d'arbres d'arité non bornée. Le résultat est donc une conséquence du théorème 1.10. □

Les DTD sont strictement moins expressives que les langages réguliers d'arbres d'arité non bornée. Cela est dû au fait que le type des enfants d'un nœud est déterminé par l'étiquette du père, indépendamment du contexte (Papakonstantinou et Vianu, 2000). En particulier, une DTD D possède la propriété suivante : si un arbre satisfait D, et si deux de ses sous-arbres ont la même étiquette à leur racine, alors intervertir ces deux sous-arbres ou substituer l'un par l'autre conduit à un nouvel arbre valide par rapport à D. Ainsi, les DTD appartiennent à la famille de *langages locaux* (Murata *et al.*, 2005).

1.4.2 DTD étendues (EDTD)

Les DTD étendues (ou spécialisées) ont été proposées par Papakonstantinou et Vianu (2000) dans le but de s'affranchir du manque d'expressivité des DTD. L'idée est d'associer un type (optionnel) à chaque étiquette afin de prendre en compte le contexte et de définir les modèles de contenu (c.-à-d. les expressions régulières) sur les couples étiquette/type. De cette façon, les DTD étendues capturent la classe des langages réguliers d'arbres d'arité non bornée.

Considérons l'exemple suivant. Supposons que l'on veuille enrichir notre description d'un pays afin de distinguer des régions de type "division administrative" (p. ex. Nord-Pas de Calais, Rhône-Alpes, etc.) de celles de type "zone géographique" (Flandre, Vallée du Rhône, etc.), sachant que les premières possèdent une information sur le nombre d'habitants qui n'existe pas au niveau des secondes. Dans une DTD standard, cette distinction n'est pas envisageable et l'on doit se contenter d'un élément $\mathtt{population}$ optionnel. La figure 23 propose une DTD étendue l'autorisant grâce à l'introduction de deux types *Admin* et *Geog* associés à l'élément $\mathtt{region}$.

Formellement, une DTD étendue (EDTD) D sur un alphabet Σ consiste en un ensemble fini d'états $\mathtt{sta}(D) \subseteq \Sigma \times \mathbb{N}$ (autrement dit des symboles associés à des types), un sous-ensemble d'états initiaux $\mathtt{init}(D) \subseteq \mathtt{sta}(D)$ et une collection de règles donnée par une fonction qui associe des états $(a, i) \in \mathtt{sta}(D)$ à des expressions régulières e sur $\mathtt{sta}(D)$,

$$\begin{array}{lll} \mathtt{country} & \rightarrow & \mathtt{name} \cdot \mathtt{city} \cdot (\mathtt{region}^{Admin} + \mathtt{region}^{Geog})^* \\ \mathtt{region}^{Admin} & \rightarrow & \mathtt{name} \cdot \mathtt{population} \cdot \mathtt{city}^* \\ \mathtt{region}^{Geog} & \rightarrow & \mathtt{name} \cdot \mathtt{city}^* \\ \mathtt{name} & \rightarrow & \epsilon \\ \mathtt{city} & \rightarrow & \epsilon \\ \mathtt{population} & \rightarrow & \epsilon \end{array}$$

Figure 23. *Une DTD étendue décrivant des pays et des régions.*

auquel cas nous écrivons $(a, i) \rightarrow_D e$. Le langage $L_{(a,i)}(D) \subseteq T^u_\Sigma$ d'un état $(a, i) \in sta(D)$ est le plus petit ensemble d'arbres d'arité non bornée tel que :

$$\mathcal{L}_{(a,i)}(D) = \{\ a(t_1, \ldots, t_n) \mid t_j \in \mathcal{L}_{q_j}(D) \text{ pour } 1 \leqslant j \leqslant n, \\ q_1 \cdot \cdots \cdot q_n \in \mathcal{L}(e) \wedge (a, i) \rightarrow_D e\ \}$$

Le langage d'une EDTD D sur Σ est :

$$\mathcal{L}(D) = \bigcup_{(a,i) \in init(D)} \mathcal{L}_{(a,i)}(D)$$

La taille de D est donnée par la taille totale des règles qui la composent, c'est-à-dire $|D| = \sum_{(a,i) \rightarrow_D e} |(a, i)| + |e| = |sta(D)| + \sum_{(a,i) \rightarrow_D e} |e|$. Une EDTD est déterministe si, comme dans le cas des DTD, elle a au plus un état initial et si toutes les expressions régulières qui la composent sont déterministes.

Par essence, les DTD étendues sont équivalentes aux automates *hedge* avec des expressions régulières pour définir les langages horizontaux. De ce fait, elles peuvent reconnaître tous les langages réguliers d'arbres d'arité non bornée. XDuce (Hosoya et Pierce, 2003) et Relax NG (van der Vlist, 2003) sont des exemples de langages de schémas XML permettant de capturer la classe des langages réguliers d'arbres.

EDTD *single-type* et *restrained competition*

L'ajout de types dans la définition des DTD étendue conduit au problème du typage des arbres XML : étant donnés un document et une EDTD, il s'agit d'assigner un type valide aux étiquettes du document relativement à la DTD étendue. Calculer le type d'une étiquette avec une EDTD augmente le coût de traitement des documents XML par rapport

$$
\begin{array}{lcl}
\mathtt{country} & \rightarrow & \mathtt{name} \cdot \mathtt{city} \cdot \mathtt{regions}^{Admin} \cdot \mathtt{regions}^{Geog} \\
\mathtt{regions}^{Admin} & \rightarrow & \mathtt{region}^{Admin}* \\
\mathtt{regions}^{Geog} & \rightarrow & \mathtt{region}^{Geog}* \\
\mathtt{region}^{Admin} & \rightarrow & \mathtt{name} \cdot \mathtt{population} \cdot \mathtt{city}^* \\
\mathtt{region}^{Geog} & \rightarrow & \mathtt{name} \cdot \mathtt{city}^* \\
\mathtt{name} & \rightarrow & \epsilon \\
\mathtt{city} & \rightarrow & \epsilon \\
\mathtt{population} & \rightarrow & \epsilon
\end{array}
$$

Figure 24. *Une DTD étendue restrained competition pour décrire des pays et des régions.*

aux DTD standards. C'est pourquoi quelques restrictions ont été proposées pour les DTD étendues.

La première consiste simplement à empêcher la présence de deux états compétiteurs dans une même expression régulière. On dit de deux états qu'ils sont compétiteurs si tous deux se rapportent au même symbole de l'alphabet. Les DTD étendues respectant cette contrainte sont dites *single-type* (de type simple). Cette formalisation permet de capturer le langage de schéma W3C XML Schema (Murata *et al.*, 2005; Martens *et al.*, 2006).

La seconde restriction est similaire au déterminisme exigé dans les DTD standards, mais au niveau des types. Une DTD étendue est dite *restrained competition* si pour toute règle $q \rightarrow_D e$ il n'existe pas deux états compétiteurs (a, n_1) et (a, n_2) et des mots $u, v_1, v_2 \in \mathtt{sta}(D)^*$ tels que $u \cdot (a, n_1) \cdot v_1 \in \mathcal{L}(e)$ et $u \cdot (a, n_2) \cdot v_2 \in \mathcal{L}(e)$. Autrement dit, le type de l'étiquette d'un nœud dépend uniquement du type de ses prédécesseurs.

Par définition, toute EDTD *single-type* est aussi *restrained competition*, et toute EDTD *restrained competition* est déterministe (Murata *et al.*, 2005; Martens *et al.*, 2006). Pour le voir, reconsidérons par exemple la DTD étendue de la figure 23. Celle-ci est clairement déterministe mais n'est pas *restrained competition*. En effet, $\mathtt{region}^{Admin}$ et $\mathtt{region}^{Geog}$ sont compétiteurs et il est possible de former les mots $\mathtt{name} \cdot \mathtt{city} \cdot \mathtt{region}^{Admin}$ et $\mathtt{name} \cdot \mathtt{city} \cdot \mathtt{region}^{Geog}$. Le type de $\mathtt{region}$ ne dépend donc pas dans ce cas de ses prédécesseurs mais de ses descendants. Une DTD étendue *restrained competition* est par exemple celle de la figure 24. On a introduit un nouveau symbole $\mathtt{regions}$ afin de former deux ensembles de sous-arbres distincts pour les régions administratives d'une part et géographiques d'autre part, dans cet ordre, afin de lever l'ambiguïté. Cette EDTD n'est pas *single-type* car $\mathtt{regions}^{Admin}$ et $\mathtt{regions}^{Geog}$ sont compétiteurs et dans la même expression régulière.

Par ailleurs, les DTD étendues *restrained competition* sont strictement plus expressives que les DTD déterministes. Néanmoins, elles demeurent plus restrictives que la classe des langages réguliers afin de permettre le typage de tous les nœuds d'un document XML en une seule traversée (Martens *et al.*, 2005). Prenons par exemple le langage régulier d'arbres d'arité non bornée $L_1 = \{a(a)\}$. Celui-ci ne peut être reconnu par aucune DTD car il contient un arbre ayant deux types de nœuds étiquetés par a qu'il est nécessaire de distinguer. Mais L_1 peut être reconnu par une EDTD *restrained competition* avec deux états $(a, 1)$ et $(a, 2)$ (un pour chaque type de nœuds a). Le symbole initial est $(a, 1)$ et les règles $(a, 1) \to (a, 2)$ et $(a, 2) \to \epsilon$.

L'exemple précédent illustre en outre le fait qu'il n'est pas possible de transformer une EDTD *restrained competition* en un automate d'arbres *stepwise* déterministe ascendant en temps linéaire, contrairement au cas des DTD déterministes. L'approche naïve consiste à produire des règles $a \to (a, 1)$ et $a \to (a, 2)$, ce qui viole le déterminisme. Le problème vient du fait que le type d'un nœud étiqueté par a est déterminé seulement une fois celui de son père connu, si bien qu'il faut tester toutes les possibilités dans un automate ascendant déterministe.

EDTD *restrained competition* vers automates descendants déterministes

Nous montrons comment convertir une DTD étendue *restrained competition* D sur Σ en un automate descendant déterministe pour des arbres d'arité non bornée interprétés sur le codage frère-fils reconnaissant le même langage en temps $\mathcal{O}(|\Sigma| \times |D|)$.

On calcule d'abord la collection d'automates de Glushkov G_q pour chaque expression régulière e telle que $q \to_D e$. Comme D est *restrained competition*, toutes ses expressions régulières sont 1-non-ambiguës, donc les automates de Glushkov sont déterministes et la collection a une taille globale en $\mathcal{O}(|\Sigma| \times |D|)$ (cf. th. 1.10). L'alphabet pour l'ensemble des automates G_q est $\mathtt{sta}(D) \subseteq \Sigma \times \mathbb{N}$. Sans perte de généralité, on peut supposer G_q productif pour tout q. Pour un tel G_q, la propriété *restrained competition* de D implique qu'il n'existe pas deux règles $p \overset{(a,i)}{\to} p' \in \mathtt{rul}(G_q)$ et $p \overset{(a,j)}{\to} p'' \in \mathtt{rul}(G_q)$ avec $i \neq j$; le déterminisme implique que :

- $(*)$ pour tout état $p \in \mathtt{sta}(G_q)$ et tout symbole $a \in \Sigma$, il existe au plus un couple (i, p') tel que $p \overset{(a,i)}{\to} p' \in \mathtt{rul}(G_q)$.

À partir de l'ensemble des automates de Glushkov $(G_q)_{q \in \mathtt{sta}(D)}$, on construit un automate d'arbres B sur $\Sigma_\sharp$ tel que $\mathcal{L}^u(B) = \mathcal{L}(D)$. Les états de B sont les éléments de

l'ensemble $\biguplus_{q\in sta(D)} sta(G_q) \uplus \{f,h\}$, où f est l'unique état final, c'est-à-dire l'état qui évalue la racine, et h l'état qui évalue le second fils de la racine (nécessairement étiqueté par le symbole $\sharp$ par construction). Les règles de B sont définies comme suit, où (a_0, i_0) est l'unique état initial de D et s l'unique état initial de $G_{(a_0,i_0)}$:

$$\frac{p \xrightarrow{(a,i)} p' \in rul(G_q) \qquad init(G_{(a,i)}) = \{p''\}}{p \xrightarrow{a} (p'', p')} \qquad \frac{p \in fin(G_q)}{p \xrightarrow{\sharp} ()} \qquad \frac{}{\begin{array}{c} f \xrightarrow{a_0} (s,h) \\ h \xrightarrow{\sharp} () \end{array}}$$

L'automate B est déterministe descendant par (∗) et reconnaît $\mathcal{L}(D)$. On l'obtient en temps linéaire dans la taille de la collection d'automates de Glushkov de telle sorte que la construction globale s'effectue en temps $\mathcal{O}(|\Sigma| \times |D|)$.

Proposition 1.12
Pour toute DTD étendue restrained competition D *sur* Σ*, on peut calculer un automate d'arbres déterministe descendant* B *sur* $\Sigma_\sharp$ *tel que* $\mathcal{L}^u(B) = \mathcal{L}(D)$ *en temps* $\mathcal{O}(|\Sigma| \times |D|)$.

Notons que le résultat s'applique également aux DTD déterministes, celles-ci étant strictement moins expressives que les EDTD *restrained competition*.

Mémorandum

Ce chapitre a présenté plusieurs modèles d'automates déterministes pour les arbres d'arité non bornée et les schémas XML. La notion de déterminisme est un élément capital pour l'ensemble des résultats présentés dans la suite de ce mémoire. Il est source de nombreux pièges car il recouvre des notions différentes selon le modèle considéré.

Les automates frère-fils déterministes descendants, interprétés sur les arbres d'arité non bornée *via* le codage frère-fils, sont restreints à la classe des langages clos par chemin (prop. 1.3). Leur expressivité est certes suffisante pour représenter les DTD déterministes ou les DTD étendues *restrained competition*, et traiter les problèmes d'inclusion dans les schémas déterministes du chapitre 2, mais elle demeure trop limitée pour représenter les requêtes monadiques que nous envisageons dans les chapitres qui suivent.

Les automates *hedge*, pour leur part, ne disposent pas d'une bonne notion de déterminisme. En particulier, il n'existe pas un unique automate *hedge* déterministe minimal pour tout

langage régulier d'arbres d'arité non bornée (Martens et Niehren, 2007). Ce modèle d'automates est par conséquent inadapté dans le cadre de l'apprentissage par inférence grammaticale, basée sur le théorème de Myhill-Nerode (Comon *et al.*, 2007), dans lequel se situent nos travaux au chapitre 4. Toutefois, comme ils peuvent être transformés en temps linéaire en automates frères-fils ou *stepwise* (prop. 1.6), il est possible d'utiliser les automates *hedge* en partie gauche dans les algorithmes d'inclusion asymétrique présentés dans le prochain chapitre.

Les automates *stepwise* déterministes ascendants, interprétés sur les arbres d'arité non bornée modulo le codage curryfié, capturent l'ensemble des langages réguliers. Ils disposent en outre d'un unique minimal déterministe. Pour ces deux raisons, ils sont privilégiés pour traiter l'ensemble des problèmes abordés dans la suite. Des arguments supplémentaires sont apportés au chapitre 3 sur leur adéquation à représenter des requêtes monadiques en vue de l'apprentissage guidé par schéma.

Nous avons par ailleurs introduit les automates factorisés déterministes, qui sont des *stepwise* avec ϵ-transitions, afin de représenter efficacement les DTD déterministes. Leur utilisation permet d'éviter un saut quadratique dans la transformation des DTD en automates *stepwise*. Ainsi nous obtenons au chapitre 2 un test d'inclusion efficace dans les schémas XML déterministes, que nous pouvons directement intégrer aux algorithmes d'induction de requêtes guidée par schéma exposés au chapitre 4.

2

Inclusion efficace dans les automates déterministes et les schémas XML

Ce chapitre présente des algorithmes pour vérifier l'inclusion $\mathcal{L}(A) \subseteq \mathcal{L}(B)$ entre automates d'arbres finis A et B en temps $\mathcal{O}(|A| \times |B|)$, où B est déterministe ascendant ou descendant. Les résultats sont étendus à l'inclusion d'automates pour des arbres d'arité non bornée A dans des schémas XML déterministes D en temps $\mathcal{O}(|A| \times |\Sigma| \times |D|)$, où Σ est l'alphabet de A et de D. Outre l'intérêt théorique évident de ces résultats, l'obtention d'un algorithme efficace en pratique est déterminant. En effet, ce type d'inclusion doit être vérifiée un grand nombre de fois au cours de l'induction de requêtes guidée par schéma discutée par la suite ; l'efficacité du test d'inclusion dans ce cadre d'apprentissage est renforcée par l'incrémentalité de l'algorithme relativement à l'ajout d'ϵ-règles dans l'automate A, qui modélisent la fusion de deux états. Une étude expérimentale, effectuée en fin de chapitre, permet de valider notre approche.

Sommaire

2.1 Introduction **59**
2.2 Calcul de complexité avec Datalog clos **62**
2.3 Inclusion dans les automates *stepwise* **65**
2.3.1 Caractérisation de l'inclusion 66
2.3.2 Test de la caractérisation avec Datalog clos 68
2.3.3 Test d'inclusion en temps $\mathcal{O}(|A| \times |B|)$ 71
2.3.4 Algorithme efficace 73
2.3.5 Détection "au plus tôt" de l'échec de l'inclusion 77
2.3.6 Incrémentalité 79
2.4 Inclusion dans les automates *stepwise* factorisés **79**
2.4.1 Test d'inclusion en temps $\mathcal{O}(|A| \times |F|)$ 80
2.4.2 Algorithme efficace 86
2.4.3 Détection "au plus tôt" de l'échec de l'inclusion 88
2.5 Inclusion dans les automates descendant **90**
2.6 Inclusion dans les schémas XML **91**
2.7 Expériences **92**
2.7.1 Données synthétiques 92
2.7.2 Données réalistes 95

2.1 Introduction

Le problème d'inclusion L'inclusion est un problème de décision classique en théorie des langages. Différents problèmes concrets ont été modélisés par l'inclusion de langages de mots, notamment en vérification formelle (Gupta, 1992). Dans le cas des arbres, on l'observe également dans diverses applications comme l'inférence de type inverse (Milo *et al.*, 2003) ou l'inclusion de schémas XML (Tozawa et Hagiya, 2003). La plupart des approches proposées se fondent sur une représentation du langage par automate, que ce soit des chaînes ou des arbres.

Il existe cependant quelques travaux récents qui abordent directement le problème de l'inclusion dans les expressions régulières. En effet, ces dernières sont à la base des modèles de contenu utilisés dans les schémas XML. Martens *et al.* (2004) ont étudié la complexité du problème d'inclusion pour des expressions régulières "simples", tandis que Colazzo *et al.* (2009) proposent un algorithme pour la vérification de type sur une classe d'expressions régulières "sans conflits". Néanmoins, des restrictions sur l'expressivité des langages considérés sont nécessaires dans ces approches pour vérifier l'inclusion en temps polynomial.

Nous nous intéressons, dans ce chapitre, au problème d'inclusion pour les automates d'arbres finis [1]. Comme tous les problèmes de décision classiques de ces automates (*membership*, *emptiness*, etc.), l'inclusion est un problème décidable (Gécseg et Steinby, 1984, théorème 10.3). Ce résultat repose sur un lemme de pompage et des propriétés de clôture sur les opérations booléennes (union, complément, intersection) pour la classe des langages d'arbres reconnaissables (Comon *et al.*, 2007, chapitre 1).

Inclusion d'automates non déterministes Dans le cas général, décider l'inclusion $L(A) \subseteq L(B)$, où A et B sont des automates d'arbres finis non déterministes, est un problème DEXPTIME-complet. Cette complexité est la conséquence d'un résultat de Seidl (1990) qui a analysé la complexité du problème de l'équivalence des automates d'arbres finis, $L(A) = L(B)$. Seidl a en effet démontré que le problème dual d'*inequivalence* (non équivalence), c'est-à-dire décider $L(A) \neq L(B)$, est DEXPTIME-complet. Ce résultat s'applique naturellement à l'inclusion puisque $L(A) \subseteq L(B)$ si et seulement si $L(A) \cap L(B) = L(A)$.

1. Par un raccourci de langage, nous employons parfois l'expression "inclusion d'automates" alors qu'il s'agit de l'inclusion de leurs langages.

L'approche traditionnelle pour vérifier l'inclusion $L(A) \subseteq L(B)$ consiste à calculer l'automate complémentaire de B et à tester si l'intersection de son langage avec celui de A est vide, c'est-à-dire tester $L(A) \cap L(B)^c = \emptyset$. Or, pour complémenter l'automate B, il est nécessaire de le déterminiser, ce qui implique une explosion exponentielle de sa taille et donc du temps de calcul de l'inclusion. C'est pourquoi cette méthode n'est pas utilisée en pratique.

Hosoya *et al.* (2005) ont proposé un algorithme *top-down* (descendant), amélioré par Suda et Hosoya (2005), pour vérifier efficacement l'inclusion d'automates d'arbres non déterministes dans des cas typiques de vérification de type XML. Tozawa et Hagiya (2003), déjà cités plus haut, présentent une approche *bottom-up* (ascendante) du problème, motivée par l'inclusion de schémas XML. Enfin, plus récemment, Bouajjani *et al.* (2008) ont proposé un algorithme d'inclusion basé sur les antichaînes dans le cadre de la vérification de systèmes. Outre qu'elles traitent de l'inclusion d'automates d'arbres non déterministes, ces approches ont comme point commun d'éviter la déterminisation des automates.

Inclusion d'automates déterministes Une condition suffisante pour obtenir un test d'inclusion $L(A) \subseteq L(B)$ en temps polynomial est d'imposer que l'automate B soit déterministe (ascendant ou descendant). En effet, l'intersection $L(A) \cap L(B)^c$ s'obtient en calculant $A \times B^c$, l'automate produit de A et du complémentaire de B, en temps $\mathcal{O}(|A| \times |B^c|)$. Décider si le langage reconnu par cet automate est vide se fait en temps linéaire dans sa taille, en s'assurant simplement qu'il n'a pas d'état final accessible. Ainsi, vérifier $L(A) \cap L(B)^c = \emptyset$, où B est déterministe, se fait en temps $\mathcal{O}(|A| \times |B^c|)$.

L'inconvénient majeur de la précédente approche réside dans le calcul de l'automate complémentaire de B. Cette opération nécessite d'abord de compléter l'automate B avant de complémenter ses états finaux. Or, compléter B requiert l'ajout de règles pour toutes les parties gauche possibles. Dans le pire des cas, la taille de B^c est donc $\mathcal{O}(|\Sigma| \times |B|^n)$, où n est l'arité maximale des symboles dans la signature Σ. En considérant les automates pour des codages d'arbres binaires, on obtient un test d'inclusion en temps $\mathcal{O}(|A| \times |\Sigma| \times |B|^2)$. Cette complexité demeure trop élevée en pratique pour l'inclusion dans les schémas XML car la taille de B peut être de l'ordre de 500 et l'alphabet Σ compter autour de 100 symboles [2].

Une autre approche, mentionnée plus haut, peut être envisagée pour décider l'inclusion $L(A) \subseteq L(B)$ avec B déterministe. Il s'agit résoudre le problème équivalent

2. Ces ordres de grandeur ont été observés sur les DTD du corpus étudié par Bex *et al.* (2004).

$L(A) \cap L(B) = L(A)$. Mais il faut imposer comme condition supplémentaire que l'automate A soit déterministe pour le décider en temps polynomial. Cette restriction n'est pas envisageable pour le test d'inclusion asymétrique dont nous aurons besoin dans le cadre de l'induction de requêtes guidée par schéma. C'est pourquoi nous abandonnons les approches traditionnelles qui supposent, pour être polynomiales, que les automates A et B sont tous deux déterministes.

Résumé des contributions Nous présentons un algorithme efficace pour vérifier l'inclusion $L(A) \subseteq L(B)$, où A et B sont des automates d'arbres finis avec B déterministe ascendant (pas nécessairement A), en temps $O(|A| \times |B|)$, indépendamment de la taille de la signature Σ. Ce résultat est établi en premier lieu avec les automates *stepwise* pour les arbres binaires, puis étendu aux automates d'arbres d'arité bornée et aux automates *stepwise* pour les arbres d'arité non bornée grâce à la curryfication.

Dans le contexte plus spécifique de l'inclusion dans les schémas XML, nous montrons comment tester efficacement l'inclusion des automates *stepwise* pour les arbres d'arité non bornée A dans les DTD déterministes D en temps $O(|A| \times |\Sigma| \times |D|)$. Pour obtenir ce résultat, nous utilisons la notion d'automates d'arbres factorisés (déterministes) introduite au chapitre précédent. L'apparition du facteur $|\Sigma|$ est due à la transformation des DTD en automates factorisés et ne peut être évitée. À l'heure actuelle, nous ne connaissons pas d'algorithme avec une meilleure complexité pour ce problème.

Lorsque A est un automate qui opère sur le codage frère-fils des arbres d'arité non bornée et D une DTD déterministe, une légère adaptation de notre approche est nécessaire. En particulier, les DTD sont transformées en automates d'arbres descendants déterministes qui reconnaissent des arbres d'arité non bornée sur le codage frère-fils et le problème d'inclusion réduit à l'inclusion dans les automates de mots déterministes. De cette manière, vérifier l'inclusion se fait aussi en temps $O(|A| \times |\Sigma| \times |D|)$. Ce résultat s'applique également aux EDTD *restrained competition* déterministes, qui correspondent à une formalisation du standard XML Schema du W3C. Il généralise ainsi le test d'inclusion de Martens *et al.* (2006, section 10.3) qui suppose à la fois A et D déterministes et procède directement sur l'inclusion des automates de mots.

L'examen de la complexité des algorithmes d'inclusion en jeu est effectué en même temps que leur présentation. Cette analyse précise repose essentiellement sur une caractérisation du problème par l'intermédiaire d'un programme Datalog clos, outil que nous présentons

ci-après. Les résultats de complexité obtenus ne sont pas complètement évidents car ils nécessitent quelques astuces algorithmiques non triviales.

2.2 Calcul de complexité avec Datalog clos

Datalog est, à l'origine, un langage de requête pour les bases de données déductives dérivé de la logique du premier ordre (voir par exemple Abiteboul *et al.* (1995, chap. 12) et Ceri *et al.* (1989)). Dans le cadre de cette thèse, nous ne l'utilisons pas en tant que tel mais comme outil pour définir des relations sur des objets en restreignant sa syntaxe aux formules closes.

Un programme Datalog clos (Gottlob *et al.*, 2002) est un programme Datalog, c'est-à-dire un ensemble de clauses de Horn, sans symboles de fonction ni variables (et sans négation).

Plus formellement, un programme Datalog clos est défini sur un alphabet d'arité bornée (Γ, ar) composé de *constantes* c et de *prédicats* p d'arité $\mathrm{ar}(p) \geqslant 0$. Un *littéral* est un terme (clos) de la forme $p(c_1, \ldots, c_{\mathrm{ar}(p)})$ et l'on dénote par $\mathrm{lit}(\Gamma)$ l'ensemble de tous les littéraux de Γ. Une *clause* (de Horn), dénotée par $L :- L_1, \ldots, L_k.$, est un couple dans $\mathrm{lit}(\Gamma) \times \mathrm{lit}(\Gamma)^k$ avec $k \geqslant 0$; lorsque $k = 0$, on note "L." plutôt que "$L :- .$", comme c'en est l'usage.

Un *programme Datalog clos* $\mathcal{P}$ sur Γ est un ensemble fini de clauses de Horn sur Γ. La taille $|\mathcal{P}|$ d'un programme Datalog clos $\mathcal{P}$ est le nombre total d'occurrences de symboles apparaissant dans ses clauses.

Le *plus petit point fixe* [3] d'un programme Datalog clos $\mathcal{P}$ sur Γ, noté $\mathrm{lfp}(\mathcal{P})$, est le plus petit ensemble de littéraux sur Γ satisfaisant la propriété suivante : pour toute clause $L :- L_1, \ldots, L_k.$ de $\mathcal{P}$, si $L_1, \ldots, L_k \in \mathrm{lfp}(\mathcal{P})$ alors $L \in \mathrm{lfp}(\mathcal{P})$. Comme il n'y a pas de négation dans $\mathcal{P}$, son plus petit point fixe est unique et celui-ci est fini en raison de l'absence de symboles de fonction. Enfin, puisqu'un programme Datalog clos ne contient pas de variables, son plus petit point fixe peut être calculé efficacement (Ceri *et al.*, 1989; Dantsin *et al.*, 2001), ce qu'établit le théorème suivant.

Théorème 2.1 (Efficacité de Datalog clos)
Pour tout programme Datalog clos $\mathcal{P}$ sur Γ, son plus petit point fixe $\mathrm{lfp}(\mathcal{P})$ peut être calculé en temps linéaire $\mathcal{O}(|\mathcal{P}|)$.

3. Parfois abrégé en *point fixe* dans la suite.

Nous fournissons une démonstration de ce résultat qui, à notre connaissance, n'a jamais été formulé de la sorte dans la littérature.

Preuve
Un programme $\mathcal{P}$ définit un hypergraphe dont les arêtes sont des tuples $(L, L_1, \dots, L_k)$ pour toute clause $L :- L_1, \dots, L_k.$ de $\mathcal{P}$. Le plus petit point fixe $\mathtt{lfp}(\mathcal{P})$ correspond à l'ensemble des littéraux accessibles dans cet hypergraphe. On peut calculer les éléments accessibles des graphes en temps linéaire. Le même résultat est valable pour les hypergraphes sous la condition (souvent implicite) que l'égalité $L = L'$ peut être testée en temps constant. Cette condition est clairement valide lorsque tous les prédicats de $\mathcal{P}$ sont d'arité 0 ; dans ce cas, $\mathcal{P}$ correspond simplement à un ensemble de clauses de Horn propositionnelles, dont l'évaluation peut être effectuée en temps linéaire $\mathcal{O}(|\mathcal{P}|)$ (Minoux, 1988; Itai et Makowsky, 1987).

Pour un ensemble de symboles quelconque Γ, on voit l'ensemble $\mathtt{lit}(\Gamma)$ comme un alphabet d'arité bornée sans constantes où tous les littéraux sont assimilés à des prédicats d'arité 0. Chaque littéral L sur Γ est associé à exactement un littéral $L()$ sur $\mathtt{lit}(\Gamma)$. Ainsi, tout programme Datalog clos sur Γ peut être transformé en un programme Datalog clos sur $\mathtt{lit}(\Gamma)$ en temps $\mathcal{O}(|\mathcal{P}|)$.

Pour vérifier $L = L'$ en temps constant, on remplace tous les littéraux de $\mathcal{P}$ par des entiers, de telle sorte que chaque occurrence d'un même littéral est associée au même entier. On peut réaliser cette opération en temps $\mathcal{O}(|\mathcal{P}|)$ en précalculant un arbre préfixe qui mémorise tous les entiers associés aux littéraux déjà traités. Par exemple, l'arbre $r(p_1(c_1(c_2(1), c_3(2)), p_0(c_1(3))))$ mémorise les affectations de $p_1(c_1, c_2)$ à 1, de $p_1(c_1, c_3)$ à 2 et de $p_0(c_1)$ à 3. $\square$

Multiplicité Nous étendons le calcul du plus petit point fixe d'un programme Datalog clos aux multiensembles en comptant la multiplicité des littéraux, c'est-à-dire le nombre de fois où ils sont ajoutés au point fixe. La *multiplicité d'un littéral* dans le plus petit point fixe d'un programme $\mathcal{P}$ est donnée par la fonction $\mathtt{lfp}^{\#}_{\mathcal{P}} : \mathtt{lit}(\Gamma) \to \mathbb{N} \cup \{0\}$ définie comme suit, où $\#S$ dénote la cardinalité de l'ensemble S :

$$\mathtt{lfp}^{\#}_{\mathcal{P}}(L) = \#\{R \in \mathtt{lfp}(\mathcal{P})^k \mid L :- R. \in \mathcal{P} \wedge k \geqslant 0\}$$

Par définition, $L \in \mathtt{lfp}(\mathcal{P})$ si et seulement si $\mathtt{lfp}^{\#}_{\mathcal{P}}(L) > 0$. Le corollaire suivant montre que les multiplicités des littéraux dans le point fixe peuvent être calculées efficacement.

Corollaire 2.2
Pour toute signature Γ *et tout programme Datalog* $\mathcal{P}$ *sur* Γ, *une représentation du plus petit point fixe avec multiplicités* $\mathtt{lfp}^{\#}_{\mathcal{P}}$ *peut être calculée en temps linéaire* $O(|\mathcal{P}|)$.

Nous représentons $\mathtt{lfp}^{\#}_{\mathcal{P}}$ par sa restriction à $\mathtt{lfp}(\mathcal{P})$, c'est-à-dire par la relation $\{(L, \mathtt{lfp}^{\#}_{P}(L)) \mid L \in \mathtt{lfp}(\mathcal{P})\}$ qui contient toutes les valeurs non nulles de $\mathtt{lfp}^{\#}_{P}$. Par ce biais, on évite d'énumérer les éléments du complémentaire du plus petit point fixe de $\mathcal{P}$.

Preuve
La relation $\{(L, \mathtt{lfp}^{\#}_{P}(L)) \mid L \in \mathtt{lfp}(\mathcal{P})\}$ peut être calculée à partir de $\mathcal{P}$ et de $\mathtt{lfp}(\mathcal{P})$ en temps $\mathcal{O}(|\mathcal{P}|)$ en inspectant toutes les clauses de $\mathcal{P}$ exactement une fois et en comptant, pour chaque littéral, le nombre de ses apparitions dans la partie gauche des clauses dont tous les littéraux de la partie droite appartiennent au point fixe de $\mathcal{P}$. Pour ce faire, il est suffisant de calculer $\mathtt{lfp}(\mathcal{P})$ en temps $\mathcal{O}(|\mathcal{P}|)$ (cf. th. 2.1). □

Application au calcul des états accessibles et coaccessibles d'un automate *stepwise* et à la complexité du problème du vide

Pour vérifier que le langage reconnu par un automate *stepwise* est vide, il suffit de vérifier qu'il n'a pas d'état final accessible. Nous montrons comment calculer les états accessibles d'un automate *stepwise* A sur Σ par un programme Datalog clos en temps $\mathcal{O}(|A|)$. De la même façon, nous montrons que tout *stepwise* A sur Σ peut être rendu productif en temps $\mathcal{O}(|A|)$.

Soit A un automate *stepwise* sur Σ. Un état $p \in \mathtt{sta}(A)$ est *accessible* s'il existe un arbre $t \in T^{\mathrm{bin}}_{\Sigma_@}$ tel que $p \in \mathtt{eval}_A(t)$; un état p est *coaccessible* s'il existe un contexte $C \in T^{\mathrm{bin}}_{\Sigma_@ \uplus \{p\}}$ avec une unique occurrence de p tel que $\mathtt{eval}_A(C) \cap \mathtt{fin}(A) \neq \emptyset$, où $\mathtt{eval}_A(p) = p$. L'automate A est dit *productif* si tous ses états sont à la fois accessibles et coaccessibles. Notons qu'un automate productif complété peut devenir non productif car les états puits ne sont pas coaccessibles par définition.

La figure 25 présente la transformation d'un automate *stepwise* A sur Σ en un programme Datalog clos permettant de calculer ses états accessibles et coaccessibles. Les règles de l'automate A sont transformées en clauses Datalog avec des prédicats (unaires) `acc` et `coacc`. Le nombre total de clauses produites est linéaire dans la taille de A de telle sorte que le plus petit point fixe du programme peut être calculé en temps linéaire (cf. th. 2.1).

$\frac{a \rightarrow p \in \mathrm{rul}(A)}{\mathtt{acc}(p).}$	$\frac{p_1@p_2 \rightarrow p \in \mathrm{rul}(A)}{\mathtt{acc}(p) :- \mathtt{acc}(p_1), \mathtt{acc}(p_2).}$	$\frac{p' \xrightarrow{\epsilon}_A p}{\mathtt{acc}(p) :- \mathtt{acc}(p').}$
$\frac{p \in \mathrm{fin}(A)}{\mathtt{coacc}(p).}$	$\frac{p_1@p_2 \rightarrow p}{\mathtt{coacc}(p_1) :- \mathtt{coacc}(p), \mathtt{acc}(p_2). \quad \mathtt{coacc}(p_2) :- \mathtt{coacc}(p), \mathtt{acc}(p_1).}$	$\frac{p' \xrightarrow{\epsilon}_A p}{\mathtt{coacc}(p') :- \mathtt{coacc}(p).}$

Figure 25. *Calcul des états accessibles et coaccessibles d'un automate stepwise* A *par un programme Datalog clos.*

Théorème 2.3
On peut décider en temps linéaire $\mathcal{O}(|A|)$ *si le langage reconnu par un automate stepwise* A *sur* Σ *est vide.*

Par ailleurs, les états qui ne sont ni accessibles ni coaccessibles, ainsi que les règles qui les utilisent, peuvent être supprimés sans risque car ils sont inutiles pour évaluer un arbre. Cela donne donc une procédure effective pour rendre tout automate *stepwise* A productif en temps linéaire sans affecter son langage.

Proposition 2.4
Tout automate stepwise A *sur* Σ *peut être rendu productif en temps linéaire* $\mathcal{O}(|A|)$.

2.3 Inclusion dans les automates *stepwise*

Nous étudions le *problème d'inclusion déterministe* pour les automates d'arbres. Son entrée consiste en un alphabet d'arité bornée Σ, un automate d'arbres A potentiellement non déterministe avec ϵ-règles et un automate d'arbres déterministe (descendant) B, tous deux définis sur Σ ; sa sortie est la valeur de vérité du test $\mathcal{L}(A) \subseteq \mathcal{L}(B)$. La proposition suivante est une conséquence du lemme 1.4 (p. 36).

Proposition 2.5
Le problème d'inclusion déterministe pour les automates standards sur les arbres d'arité bornée peut être réduit en temps linéaire au problème d'inclusion déterministe pour les automates stepwise interprétés sur les arbres d'arité bornée.

Dans la suite, nous examinons le problème d'inclusion déterministe de deux automates *stepwise* A et B sur Σ. Pour obtenir un test d'inclusion $\mathcal{L}(A) \subseteq \mathcal{L}(B)$ efficace, il faut avant

tout éviter de calculer l'automate complémentaire de B. Le principe général de notre algorithme est de détecter l'échec de l'inclusion $\mathcal{L}(A) \nsubseteq \mathcal{L}(B)$ par le biais d'un automate produit $A \times B$. Autrement dit, nous cherchons à mettre en évidence l'existence d'un arbre reconnu par A qui n'appartienne pas au langage de B, indépendamment de son complémentaire. L'efficacité de l'algorithme auquel nous aboutissons repose sur le fait que l'on ne matérialise de $A \times B$ que sa partie accessible.

L'automate produit $A \times B$ défini sur Σ que nous considérons a pour états l'ensemble $\mathtt{sta}(A) \times \mathtt{sta}(B)$ et ses règles sont obtenues à partir de celles de A et de B comme suit :

$$\frac{a \to p \in \mathtt{rul}(A) \quad a \to q \in \mathtt{rul}(B)}{a \to (p, q)} \qquad \frac{p_1@p_2 \to p \in \mathtt{rul}(A) \quad q_1@q_2 \to q \in \mathtt{rul}(B)}{(p_1, q_1)@(p_2, q_2) \to (p, q)} \qquad \frac{p' \xrightarrow{\epsilon}_A p \quad q \in \mathtt{sta}(B)}{(p', q) \xrightarrow{\epsilon}_{A \times B} (p, q)}$$

Les états finaux de $A \times B$ sont donnés par l'ensemble $\mathtt{sta}(A) \times \mathtt{sta}(B) \setminus \mathtt{fin}(B)$ mais ils ne seront pas utilisés dans notre caractérisation de l'inclusion.

Nous adoptons dans cette section l'ensemble de notations et de notions suivant.

Propriété sémantique. Dans la suite, une propriété sémantique Φ satisfaite par l'automate A est notée $A \models \Phi$, sa négation $A \not\models \Phi$. On note $A, B \models \Phi$ si la propriété porte sur A et sur B. Les propriétés peuvent être composées suivant les connecteurs usuels de la logique du premier ordre. Par exemple, on écrit $A, B \models \Phi_1 \vee \Phi_2$ si et seulement si $A, B \models \Phi_1$ ou $A, B \models \Phi_2$.

Partie gauche productive. La propriété suivante pour $p_1, p_2 \in \mathtt{sta}(A)$ signifie que l'évaluation d'un arbre par A peut continuer *via* ces deux états :

$$A \models p_1 \underline{@} p_2 \overset{\text{déf.}}{\Leftrightarrow} \exists p \in \mathtt{sta}(A) \text{ tq. } p_1@p_2 \to p \in \mathtt{rul}(A)$$

Accessibilité dans le produit. Soit $p \in \mathtt{sta}(A)$ et $q \in \mathtt{sta}(B)$, on dénote l'accessibilité de (p, q) dans le produit $A \times B$ comme suit :

$$A, B \models \mathsf{acc}(p, q) \overset{\text{déf.}}{\Leftrightarrow} (p, q) \text{ est accessible dans } A \times B$$

2.3.1 Caractérisation de l'inclusion

Nous présentons une caractérisation de l'inclusion par un ensemble de conditions qui permettent de vérifier son échec.

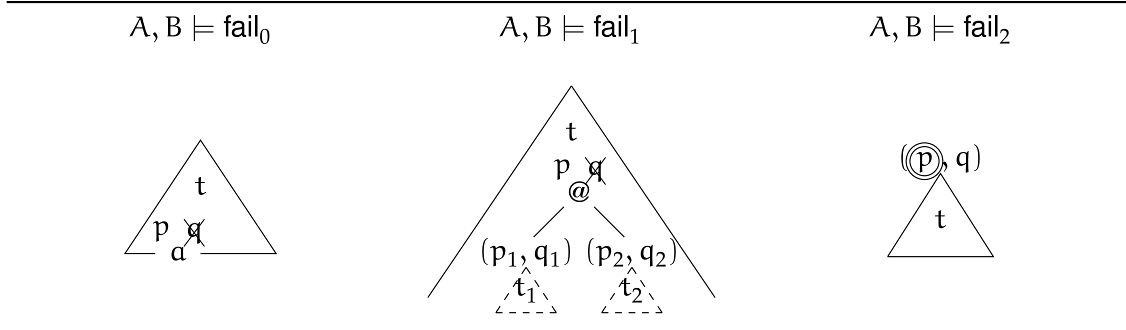


Figure 26. *Échec de l'inclusion* $\mathcal{L}(A) \subseteq \mathcal{L}(B)$.

Définition 2.6
L'inclusion $\mathcal{L}(A) \subseteq \mathcal{L}(B)$ échoue sous l'une des trois conditions suivantes :

$A, B \models \mathsf{fail}_0 \overset{\text{déf.}}{\Leftrightarrow}$ il existe une règle $a \to p \in rul(A)$ mais aucun état $q \in sta(B)$ tel que $a \to q \in rul(B)$;

$A, B \models \mathsf{fail}_1 \overset{\text{déf.}}{\Leftrightarrow}$ il existe des états $p_1, p_2 \in sta(A)$ et $q_1, q_2 \in sta(B)$ tels que $A, B \models \mathsf{acc}(p_1, q_1)$, $A, B \models \mathsf{acc}(p_2, q_2)$, $A \models p_1@p_2$ et $B \not\models q_1@q_2$;

$A, B \models \mathsf{fail}_2 \overset{\text{déf.}}{\Leftrightarrow}$ il existe des états $p \in fin(A)$ et $q \notin fin(B)$ tels que $A, B \models \mathsf{acc}(p, q)$. ◀

Proposition 2.7
Soient A *et* B *deux automates stepwise tels que* A *est productif (avec possiblement des* ϵ*-règles) et* B *déterministe. L'inclusion* $\mathcal{L}(A) \subseteq \mathcal{L}(B)$ *échoue si et seulement si* $A, B \models \mathsf{fail}_0 \vee \mathsf{fail}_1 \vee \mathsf{fail}_2$.

Preuve
Pour la correction, on suppose l'une des trois conditions d'échec réalisée et l'on démontre l'existence d'un arbre t reconnu par A mais pas par B, c'est-à-dire $t \in \mathcal{L}(A)$ et $t \notin \mathcal{L}(B)$. Ces situations sont illustrées à la fig. 26.

$A, B \models \mathsf{fail}_0$. On considère une règle $a \to p \in rul(A)$. Comme A est productif, l'état p est coaccessible. Autrement dit, il existe un contexte $C \in T^{bin}_{\Sigma_@ \uplus \{p\}}$ avec une unique occurrence de p tel que $eval_A(C) \cap fin(A) \neq \emptyset$. D'où $C[a] \in \mathcal{L}(A)$ mais $C[a] \notin \mathcal{L}(B)$ puisqu'il n'existe pas d'état q dans B tel que $a \to q \in rul(B)$.

$A, B \models \mathsf{fail}_1$. Du fait de l'accessibilité de (p_1, q_1) et (p_2, q_2), il existe $t_1, t_2 \in T_\Sigma$ tels que $(p_1, q_1) \in eval_{A \times B}(t_1)$ et $(p_2, q_2) \in eval_{A \times B}(t_2)$. Comme il existe $p \in sta(A)$ tel que $p_1@p_2 \to p \in rul(A)$, il s'ensuit $p \in eval_A(t_1@t_2)$ par définition de $eval_A$. De plus, comme A est productif, il existe un contexte $C \in T^{bin}_{\Sigma_@ \uplus \{p\}}$ avec une unique occurrence de p tel que $C[t_1@t_2] \in \mathcal{L}(A)$. Comme B est déterministe, $q_1 \in eval_B(t_1)$

et $q_2 \in eval_B(t_2)$ sont nécessairement uniques. Par hypothèse, il n'existe pas $q \in sta(B)$ tel que $q_1@q_2 \to q \in rul(B)$, de telle sorte que $C[t_1@t_2] \notin \mathcal{L}(B)$.

$A, B \models \mathsf{fail}_2$. Soient $p \in fin(A)$ et $q \notin fin(B)$ tels que (p, q) est accessible dans $A \times B$. Alors il existe $t \in T^{bin}_{\Sigma_@}$ tel que $(p, q) \in eval_{A\times B}(t)$. L'état p étant final dans A, il s'ensuit $t \in \mathcal{L}(A)$. Comme B est déterministe, $q \in eval_B(t)$ est nécessairement unique. Mais q n'étant pas un état final de B, il s'ensuit $t \notin \mathcal{L}(B)$.

Pour la complétude, on suppose l'existence d'un arbre t reconnu par A mais pas par B, c.-à-d. $t \in \mathcal{L}(A)$ et $t \notin \mathcal{L}(B)$, et l'on montre que l'une des trois conditions d'échec est réalisée. Deux cas doivent être considérés en fonction de $eval_B(t)$.

1. Supposons $eval_B(t) = \emptyset$. Il existe t', un plus petit sous-arbre de t, tel que $eval_B(t') = \emptyset$ également. On considère deux sous-cas selon t'.
   a) Si $t' = a$ est une feuille alors $eval_A(t') \neq \emptyset$ puisque $t \in \mathcal{L}(A)$. Comme A est productif, il existe $p \in sta(A)$ tel que $a \to p \in rul(A)$. Comme $eval_B(a) = \emptyset$, il n'existe pas d'état q de B tel que $a \to q \in rul(B)$. D'où $A, B \models \mathsf{fail}_0$.
   b) Si $t' = t_1@t_2$ alors il existe $p_1 \in eval_A(t_1)$, $p_2 \in eval_A(t_2)$ et $p_1@p_2 \to p \in rul(A)$ puisque $t \in \mathcal{L}(A)$. Comme t' est un sous-arbre minimal de t et B est déterministe, on a $eval_B(t_1) = \{q_1\}$ et $eval_B(t_2) = \{q_2\}$. Comme $eval_B(t_1@t_2) = \emptyset$, il n'existe pas d'état q de B tel que $q_1@q_2 \to q \in rul(B)$. D'où $A, B \models \mathsf{fail}_1$.
2. Supposons $eval_B(t) \neq \emptyset$. Alors il existe $q \in eval_B(t)$ nécessairement unique puisque B est déterministe. Comme $t \notin \mathcal{L}(B)$, il s'ensuit $q \notin fin(B)$. De plus, comme $t \in \mathcal{L}(A)$, il existe $p \in eval_A(t) \cap fin(A)$. D'où $A, B \models \mathsf{fail}_2$. □

2.3.2 Test de la caractérisation avec Datalog clos

La fig. 27 présente la transformation de deux automates *stepwise* A et B en un programme Datalog $\mathcal{D}_0(A, B)$ qui teste $A, B \models \mathsf{fail}_0$ et $A, B \models \mathsf{fail}_2$. Les clauses produites à partir de A et B par les règles de transformation (Acc_1), (Acc_2) et (Acc_3) calculent la relation d'accessibilité de l'automate produit $A \times B$. On a clairement $\mathtt{acc}(p, q) \in lfp(\mathcal{D}_0(A, B))$ si et seulement si $A, B \models \mathsf{acc}(p, q)$. Les clauses produites par les règles de transformation (Fail_0) et (Fail_2) servent au calcul des prédicats $\mathtt{fail}_0$ et $\mathtt{fail}_2$. Par construction, $\mathtt{fail}_0 \in lfp(\mathcal{D}_0(A, B))$ si et seulement si $A, B \models \mathsf{fail}_0$ et $\mathtt{fail}_2 \in lfp(\mathcal{D}_0(A, B))$ ssi $A, B \models \mathsf{fail}_2$.

$$(\mathrm{Acc}_1) \quad \frac{a \to p \in \mathrm{rul}(A) \qquad a \to q \in \mathrm{rul}(B)}{\mathtt{acc}(p, q).}$$

$$(\mathrm{Acc}_2) \quad \frac{p' \overset{\epsilon}{\to} p \in \mathrm{rul}(A) \qquad q \in \mathrm{sta}(B)}{\mathtt{acc}(p, q) :- \mathtt{acc}(p', q).}$$

$$(\mathrm{Acc}_3) \quad \frac{p_1@p_2 \to p \in \mathrm{rul}(A) \qquad q_1@q_2 \to q \in \mathrm{rul}(B)}{\mathtt{acc}(p, q) :- \mathtt{acc}(p_1, q_1), \mathtt{acc}(p_2, q_2).}$$

$$(\mathrm{Fail}_0) \quad \frac{a \to p \in \mathrm{rul}(A) \qquad \nexists q \in \mathrm{sta}(B) \text{ tq. } a \to q \in \mathrm{rul}(B)}{\mathtt{fail}_0.}$$

$$(\mathrm{Fail}_2) \quad \frac{p \in \mathrm{fin}(A) \qquad q \notin \mathrm{fin}(B)}{\mathtt{fail}_2 :- \mathtt{acc}(p, q).}$$

Figure 27. *Transformation d'automates* A, B *en un programme Datalog* $\mathcal{D}_0(A, B)$ *pour tester* $A, B \models \mathsf{fail}_0$ *et* $A, B \models \mathsf{fail}_2$.

Le programme $\mathcal{D}_0(A, B)$ est obtenu en temps $\mathcal{O}(|A| \times |B|)$ à partir de A et B, de telle sorte que sa taille est en $\mathcal{O}(|A| \times |B|)$. De plus, le plus petit point fixe de $\mathcal{D}_0(A, B)$ peut être calculé en temps $\mathcal{O}(|A| \times |B|)$ (cf. th. 2.1).

Tester $A, B \models \mathsf{fail}_1$ en temps $\mathcal{O}(|A| \times |B|)$ n'est pas trivial. À cet effet, nous introduisons pour les états $p \in \mathrm{sta}(A)$ et $q \in \mathrm{sta}(B)$ un prédicat d'*états interdits*, dénoté $\mathtt{frb}(p, q)$, équivalent à l'implication $\mathtt{acc}(p, q) \to \mathtt{fail}_1$ et dont la sémantique est donnée ci-dessous :

$$A, B \models \mathsf{frb}(p, q) \overset{\text{déf.}}{\Leftrightarrow} A, B \models \mathsf{acc}(p, q) \Rightarrow \mathsf{fail}_1$$

L'inclusion échoue si des états interdits sont accessibles. Le lemme suivant est une conséquence immédiate des définitions précédentes.

Lemme 2.8
$A, B \models \mathsf{frb}(p, q)$ *si et seulement s'il existe* p', q' *tels que l'une des deux conditions suivantes est vérifiée :*

1. $A \models p\underline{@}p' \wedge A, B \models \mathsf{acc}(p', q') \wedge B \not\models q\underline{@}q'$, *ou*
2. $A \models p'\underline{@}p \wedge A, B \models \mathsf{acc}(p', q') \wedge B \not\models q'\underline{@}q$.

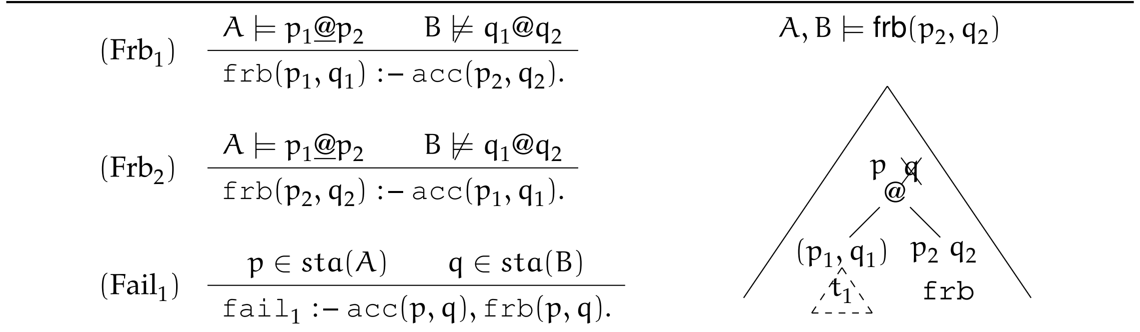


Figure 28. *Transformation d'automates* A, B *en un programme Datalog* $\mathcal{D}_1(A, B)$ *pour tester* $A, B \models \mathsf{fail}_1$. *Illustration de la situation pour la règle* (Frb$_2$).

La fig. 28 étend le programme $\mathcal{D}_0(A, B)$ à un programme $\mathcal{D}_1(A, B)$ avec des clauses pour le prédicat $\texttt{fail}_1$. Celles-ci sont produites par les règles de transformation (Frb$_1$) et (Frb$_2$) justifiées par le lemme 2.8. La règle (Fail$_1$) provient de la définition de $A, B \models \mathsf{frb}(p, q)$. On a clairement $\texttt{fail}_1 \in lfp(\mathcal{D}_1(A, B))$ si et seulement si $A, B \models \mathsf{fail}_1$.

Proposition 2.9
Soient A *et* B *deux automates. Si* A *est productif et* B *déterministe alors :*

$$\mathcal{L}(A) \subseteq \mathcal{L}(B) \Leftrightarrow lfp(\mathcal{D}_1(A, B)) \cap \{\texttt{fail}_0, \texttt{fail}_1, \texttt{fail}_2\} = \emptyset$$

Preuve
Le résultat se déduit de la proposition 2.7 et du lemme 2.8. □

Toutefois, le nombre de clauses produites par (Frb$_1$) et (Frb$_2$) peut atteindre $\mathcal{O}(|A| \times |sta(B)|^2)$ dans le pire des cas. La taille du programme $\mathcal{D}_1(A, B)$ est donc bornée par $\mathcal{O}(|A| \times |B|^2)$. Sur l'exemple de la fig. 29, explicité plus bas, la règle de transformation (Frb$_2$) produit n^2 clauses dans $\mathcal{D}_1(A, B)$ à partir des automates A et B.

En calculant le plus petit point fixe de $\mathcal{D}_1(A, B)$, on peut donc vérifier l'inclusion $\mathcal{L}(A) \subseteq \mathcal{L}(B)$ en temps $\mathcal{O}(|A| \times |B|^2)$. Malheureusement, bien qu'elle ne dépende plus de la taille de Σ, la complexité de cet algorithme d'inclusion n'est pas significativement moins élevée que celle du test naïf avec complémentation discuté dans la section 2.1.

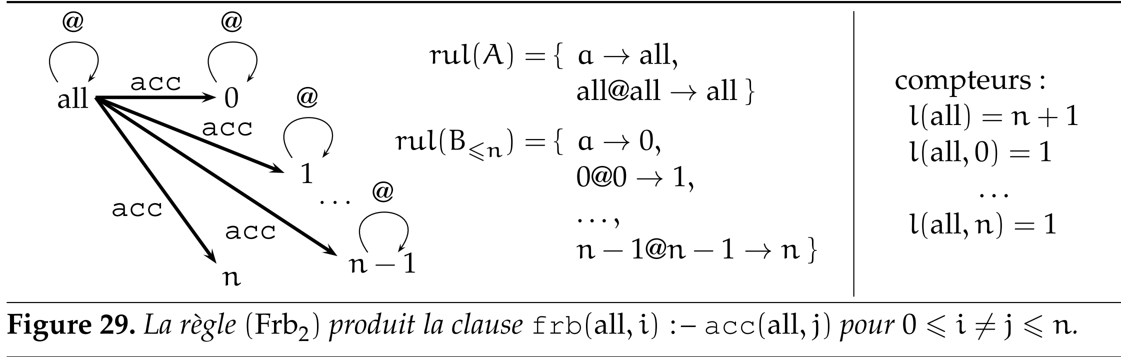


Figure 29. *La règle* (Frb$_2$) *produit la clause* $\mathtt{frb}(\text{all}, i)$:– $\mathtt{acc}(\text{all}, j)$ *pour* $0 \leqslant i \neq j \leqslant n$.

2.3.3 Test d'inclusion en temps $\mathcal{O}(|A| \times |B|)$

Le problème des règles de transformation (Frb$_1$) et (Frb$_2$) est l'énumération de clauses pour les états interdits. Cette opération rend explicite de l'information négative là où l'on souhaiterait qu'elle reste implicite.

Pour s'en rendre compte, posons $\Sigma = \{a\}$ et considérons l'exemple de la fig. 29. L'automate A possède un unique état (final) tel que $\mathtt{sta}(A) = \mathtt{fin}(A) = \{\text{all}\}$. Cet automate reconnaît l'ensemble des arbres binaires sur $\Sigma_@$ dont la longueur des chemins sur la gauche est arbitraire. Ces arbres représentent le codage curryfié des arbres d'arité non bornée dont les nœuds ont un nombre arbitraire de fils. L'automate $B_{\leqslant n}$ a pour états $\mathtt{sta}(B_{\leqslant n}) = \mathtt{fin}(B_{\leqslant n}) = \{0, 1, \ldots, n\}$. Il reconnaît l'ensemble des arbres binaires sur $\Sigma_@$ dont les chemins sur la gauche (resp. la droite) sont de longueur inférieure ou égale à n. Les nœuds de leur codage ont donc au plus n enfants.

Dans le dessin de la fig. 29, les règles telles que $A \models p@p$, resp. $B_{\leqslant n} \models q@q$, sont représentées par des boucles sur les états $p \in \mathtt{sta}(A)$, resp. $q \in B_{\leqslant n}$, étiquetées par le symbole @. L'accessibilité $A, B_{\leqslant n} \models \mathsf{acc}(\text{all}, j)$, représentée par une flèche de all vers j, est vérifiée pour $0 \leqslant j \leqslant n$ et implique les états interdits $A, B_{\leqslant n} \models \mathsf{frb}(\text{all}, i)$ pour $0 \leqslant i \neq j \leqslant n$. Il y a donc un nombre quadratique de clauses de la forme $\mathtt{frb}(\text{all}, i)$:– $\mathtt{acc}(\text{all}, j)$ dans le programme $\mathcal{D}_1(A, B_{\leqslant n})$. C'est cette explosion quadratique, revenant en fait à la complexité du calcul du complémentaire de $B_{\leqslant n}$, que l'on souhaite éviter.

L'idée est de compter l'information positive et de disposer d'un moyen de déduire l'information négative sans avoir à la matérialiser. Étant donné un état $p \in \mathtt{sta}(A)$, on compte le nombre de couples (p', q') tels que $A, B \models \mathsf{acc}(p', q')$ et $A \models p'@p$, ou *vice versa*, que l'on compare avec le nombre de couples qui impliquent $A, B \models \mathsf{frb}(p, q)$. Ces deux

```
// Initialisation des compteurs
for all p ∈ sta(A) do
  l(p) ← 0
  for all q ∈ sta(B) do
    l(p, q) ← 0
// Incrément des compteurs
for all p1, p2 ∈ sta(A) tq. A ⊨ p1@p2 do
  for all q ∈ sta(B) do
    if acc(p1, q) ∈ lfp(D0(A, B)) then
      l(p2) ← l(p2) + 1
    if acc(p2, q) ∈ lfp(D0(A, B)) then
      l(p1) ← l(p1) + 1
  for all q1, q2 ∈ sta(B) tq. B ⊨ q1@q2 do
    if acc(p1, q1) ∈ lfp(D0(A, B)) then
      l(p2, q2) ← l(p2, q2) + 1
    if acc(p2, q2) ∈ lfp(D0(A, B)) then
      l(p1, q1) ← l(p1, q1) + 1
```

Figure 30. *Compter en temps* $\mathcal{O}(|A| \times |B|)$.

nombres sont formalisés ci-dessous par les compteurs $l(p)$ et $l(p,q)$, respectivement (2.1) et (2.2) :

$$l(p) = \begin{array}{l} \#\{(p',q') \mid A \models p'\underline{@}p \wedge A,B \models \mathsf{acc}(p',q')\} \\ + \#\{(p',q') \mid A \models p\underline{@}p' \wedge A,B \models \mathsf{acc}(p',q')\} \end{array} \tag{2.1}$$

$$l(p,q) = \begin{array}{l} \#\{(p',q') \mid A \models p'\underline{@}p \wedge A,B \models \mathsf{acc}(p',q') \wedge B \models q'\underline{@}q\} \\ + \#\{(p',q') \mid A \models p\underline{@}p' \wedge A,B \models \mathsf{acc}(p',q') \wedge B \models q\underline{@}q'\} \end{array} \tag{2.2}$$

Lemme 2.10
$A,B \models \mathsf{frb}(p,q)$ *si et seulement si* $l(p) > l(p,q)$.

Preuve
Par définition, $l(p) \geqslant l(p,q)$ pour tout p,q. On a $l(p) > l(p,q)$ ssi il existe p',q' tels que $A \models p'\underline{@}p \wedge A,B \models \mathsf{acc}(p',q') \wedge B \not\models q'\underline{@}q$ ou, par symétrie, $A \models p\underline{@}p' \wedge A,B \models \mathsf{acc}(p',q') \wedge B \not\models q\underline{@}q'$. D'après le lemme 2.8, ceci équivaut à $A,B \models \mathsf{frb}(p,q)$. □

Il nous reste à voir comment calculer la collection de compteurs $l(p)$ et $l(p,q)$ pour tout $p \in \mathtt{sta}(A)$ et $q \in \mathtt{sta}(B)$ en temps $\mathcal{O}(|A| \times |B|)$. L'opération est illustrée par la procédure de la figure 30.

Théorème 2.11
Soient A *et* B *deux automates stepwise sur* Σ*. Si* B *est déterministe alors on peut vérifier l'inclusion* $\mathcal{L}(A) \subseteq \mathcal{L}(B)$ *en temps* $\mathcal{O}(|A| \times |B|)$*, indépendamment de la taille de* Σ*.*

Preuve
On calcule d'abord $\mathcal{D}_0(A, B)$ à partir de A et B en temps $\mathcal{O}(|A| \times |B|)$ puis le plus petit point fixe de ce programme, également en temps $\mathcal{O}(|A| \times |B|)$. Si $\mathtt{lfp}(\mathcal{D}_0(A, B))$ contient $\mathtt{fail}_0$ ou $\mathtt{fail}_2$ alors l'inclusion échoue. Sinon, on calcule tous les compteurs $l(p)$ et $l(p, q)$ en temps $\mathcal{O}(|A| \times |B|)$ *via* l'algorithme de la fig. 30 et, pour tout p, q tq. $\mathtt{acc}(p, q) \in \mathtt{lfp}(\mathcal{D}_0(A, B))$, on teste $A, B \models \mathsf{frb}(p, q)$. L'inclusion est vérifiée si ce test réussit. Celui-ci peut être effectué en temps $\mathcal{O}(|A| \times |B|)$ en inspectant la valeur des compteurs (lemme 2.10). □

2.3.4 Algorithme efficace

L'algorithme précédent a une complexité dans le pire des cas satisfaisante en $\mathcal{O}(|A| \times |B|)$. Cependant, il n'est pas optimal en moyenne.

Un premier problème est qu'il énumère toutes les paires de règles de A et de B lors du calcul des valeurs des compteurs. Pour y remédier, nous proposons un algorithme qui inspecte au plus la partie accessible de $A \times B$.

Un autre problème est que la détection de l'échec $A, B \models \mathsf{fail}_1$ est effectuée seulement après le calcul du plus petit point fixe de $\mathcal{D}_0(A, B)$. Nous avons comme perspective un algorithme qui détecte l'échec de l'inclusion $\mathcal{L}(A) \subseteq \mathcal{L}(B)$ au plus tôt (cf. sect. 2.3.5), de sorte à éviter de calculer complètement $\mathtt{lfp}(\mathcal{D}_0(A, B))$ lorsque cela est possible. Ce cas de figure intervient fréquemment en pratique, comme le montrent les expériences présentées dans la section 2.7, si bien que cet algorithme est très performant.

Nous introduisons des littéraux $\mathtt{frb}^c(p, Q)$ pour $p \in \mathtt{sta}(A)$ et $Q \subseteq \mathtt{sta}(B)$ avec la sémantique suivante :

$$A, B \models \mathsf{frb}^c(p, Q) \overset{\text{déf.}}{\Leftrightarrow} A, B \models \mathsf{frb}(p, q) \text{ pour tout } q \in \mathtt{sta}(B) \setminus Q$$

Dans la figure 29, par exemple, on a $A, B \models \mathsf{frb}^c(\text{all}, \{i\})$ pour $0 \leqslant i \leqslant n$. Donc chaque littéral $\mathtt{frb}(\text{all}, i)$ est impliqué par n littéraux $\mathtt{frb}^c(\text{all}, \{j\})$, $j \neq i$, c'est-à-dire $l(\text{all}) - l(\text{all}, i)$ littéraux. L'objectif est de calculer l'ensemble des littéraux satisfaisant $A, B \models \mathsf{frb}^c(p, q)$ par un programme Datalog afin d'en déduire les valeurs $l(p) - l(p, q)$ correspondantes.

$$(\mathrm{Frb}_1^c) \quad \frac{A \models p_1 @ p_2 \qquad q_2 \in \mathtt{sta}(B)}{\mathtt{frb}^c(p_1, Q_1^B(q_2)) :- \mathtt{acc}(p_2, q_2)} \quad Q_1^B(q_2) = \{q_1 \mid B \models q_1 @ q_2\}$$

$$(\mathrm{Frb}_2^c) \quad \frac{A \models p_1 @ p_2 \qquad q_1 \in \mathtt{sta}(B)}{\mathtt{frb}^c(p_2, Q_2^B(q_1)) :- \mathtt{acc}(p_1, q_1)} \quad Q_2^B(q_1) = \{q_2 \mid B \models q_1 @ q_2\}$$

Figure 31. *Grouper les clauses produites par* (Frb_1) *et* (Frb_2).

Notons que deux littéraux $\mathtt{frb}^c(p, Q)$ et $\mathtt{frb}^c(p, Q')$ sont syntaxiquement égaux si et seulement si $Q = Q'$. Pour réaliser cette égalité, on suppose fixé un ordre total $<$ sur $\mathtt{sta}(B)$ afin d'identifier $\mathtt{frb}^c(p, Q)$ avec l'unique littéral d'arité $n+1$ $\mathtt{frb}^c(p, q_1, \dots, q_n)$ où $Q = \{q_1, \dots, q_n\}$ et $q_1 < \dots < q_n$. Ainsi, les résultats sur Datalog clos s'appliquent encore.

Dans la figure 31 sont présentées les règles de transformation (Frb_1^c) et (Frb_2^c) qui produisent des clauses Datalog pour inférer des littéraux $\mathtt{frb}^c(p, Q)$. Ces dernières permettent de regrouper les clauses produites par les règles (Frb_1) et (Frb_2).

Considérons par exemple (Frb_2^c) qui présuppose $A \models p_1 @ p_2$ et $q_1 \in \mathtt{sta}(B)$. Cette règle calcule d'abord l'ensemble $Q_2^B(q_1)$ de tous les états $q_2 \in \mathtt{sta}(B)$ tels que $B \models q_1 @ q_2$ puis produit une clause $\mathtt{frb}^c(p_2, Q_2^B(q_1)) :- \mathtt{acc}(p_2, q_2)$. Cette définition est juste puisque (cf. lem. 2.8) si $A, B \models \mathsf{acc}(p_1, q_1)$ alors pour tout $q_2 \notin Q_2^B(q_1)$ on a $A, B \models \mathsf{frb}(p_2, q_2)$ et donc $A, B \models \mathsf{frb}^c(p_2, Q_2^B(q_1))$. Pour l'exemple de la figure 29, (Frb_1^c) produit les clauses $\mathtt{frb}^c(\mathrm{all}, \{i\}) :- \mathtt{acc}(\mathrm{all}, i)$ pour $0 \leqslant i \leqslant n$. La taille de l'ensemble de ces clauses est donc en $\mathcal{O}(n)$, alors que la règle (Frb_1) produit un ensemble de clauses en $\mathcal{O}(n^2)$.

Soit le programme Datalog $\mathcal{D}_2(A, B)$ qui étend $\mathcal{D}_0(A, B)$ avec les clauses obtenues par les règles (Frb_1^c) et (Frb_2^c) de la figure 31. Observons que $A, B \models \mathsf{frb}^c(p, Q)$ si et seulement si $\mathtt{frb}^c(p, Q) \in \mathtt{lfp}(\mathcal{D}_2(A, B))$. Ce programme demeure incomplet dans le sens où les littéraux $\mathtt{frb}^c(p, Q)$ ne sont pas utilisés pour inférer $\mathtt{fail}_1$.

Lemme 2.12
$\mathcal{D}_2(A, B)$ *peut être calculé en temps* $\mathcal{O}(|A| \times |B|)$ *à partir de* A *et* B.

Preuve
Le résultat pour $\mathcal{D}_0(A, B)$ a déjà été montré (cf. sect. 2.3.2). Pour $\mathcal{D}_2(A, B)$, le nombre de clauses produites par les règles (Frb_1^c) et (Frb_2^c) est en $\mathcal{O}(|A| \times |\mathtt{sta}(B)|)$ mais la taille de chaque clause est $n+1$, soit dans le pire des cas $|\mathtt{sta}(B)| + 1$. Néanmoins, la taille globale

$$\frac{\begin{array}{c} p_1@p_2 \to p \in \mathtt{rul}(A) \\ q_1 \in \mathtt{sta}(B) \end{array} \quad \left.\begin{array}{c} q_1@q_2^1 \to q^1 \in \mathtt{rul}(B) \\ \vdots \\ q_1@q_2^n \to q^n \in \mathtt{rul}(B) \end{array}\right\} \begin{array}{l} n \text{ règles} \\ \text{avec } q_1 \\ \text{à gauche} \end{array}}{\begin{array}{c} \mathtt{frb}^c(p_2, \{q_2^1, \dots, q_2^n\}) :- \mathtt{acc}(p_1, q_1). \\ \left.\begin{array}{c} \mathtt{acc}(p, q^1) :- \mathtt{acc}(p_1, q_1), \mathtt{acc}(p_2, q_2^1). \\ \vdots \\ \mathtt{acc}(p, q^n) :- \mathtt{acc}(p_1, q_1), \mathtt{acc}(p_2, q_2^n). \end{array}\right\} n \text{ clauses} \end{array}}$$

Figure 32. *Regroupement des règles* (Frb_2^c) *et* (Acc_3).

de l'ensemble des clauses de la forme $\mathtt{frb}^c(p, Q) :- \mathtt{acc}(p, q)$ est bornée par le nombre total de clauses de la forme $\mathtt{acc}(p, q) :- \mathtt{acc}(p_1, q_1), \mathtt{acc}(p_2, q_2)$ produites dans le même temps, qui est borné, lui, par $\mathcal{O}(|A| \times |B|)$. La fig. 32 permet de le constater. La règle (Frb_2^c) a été récrite de telle sorte que les clauses produites par (Acc_3) soient inférées en même temps.

Il reste à montrer comment inférer effectivement $\mathcal{D}_2(A, B)$ en temps $\mathcal{O}(|A| \times |B|)$. La procédure ci-dessous produit toutes les clauses à partir de la règle de transformation (Frb_2^c).

for all $p_1@p_2 \to p \in \mathtt{rul}(A)$ **do**
  for all $q_1 \in \mathtt{sta}(B)$ **do**
    $Q \leftarrow Q_2^B(q_1)$
    $\mathcal{D}_2(A, B) \leftarrow \mathcal{D}_2(A, B) \cup \{\mathtt{frb}^c(p_2, Q) :- \mathtt{acc}(p_1, q_1).\}$

L'ensemble Q peut être calculé en temps $\mathcal{O}(|Q|)$ à partir d'une structure de données précalculée qui, pour un état $q_1 \in \mathtt{sta}(B)$ donné, retourne l'ensemble des règles $q_1@q_2 \to q \in \mathtt{rul}(B)$ en temps linéaire dans la taille de la sortie (au plus $|\mathtt{sta}(B)|$ règles). On procède par symétrie pour la règle (Frb_1^c). La procédure s'exécute donc en temps $\mathcal{O}(|\mathcal{D}_2(A, B)|)$, c.-à-d. $\mathcal{O}(|A| \times |B|)$. □

Les programmes Datalog $\mathcal{D}_1(A, B)$ et $\mathcal{D}_2(A, B)$ possèdent le même ensemble de clauses pour les littéraux $\mathtt{acc}(p, q)$, $\mathtt{fail}_0$ et $\mathtt{fail}_2$, donc leurs plus petits points fixes coïncident. En particulier, on peut décider $A, B \models \mathsf{fail}_0 \vee \mathsf{fail}_2$ en temps $\mathcal{O}(|A| \times |B|)$ en testant l'appartenance de $\mathtt{fail}_0$ et $\mathtt{fail}_2$ à $\mathtt{lfp}(\mathcal{D}_2(A, B))$. Il nous reste à relier les deux programmes au niveau des états interdits et, partant, de $\mathtt{fail}_1$.

Lemme 2.13
Un littéral $\mathtt{frb}(p, q)$ *appartient à* $\mathrm{lfp}(\mathcal{D}_1(A, B))$ *si et seulement s'il existe un ensemble* $Q \subseteq \mathrm{sta}(B)$ *ne contenant pas* q *tel que* $\mathtt{frb}^c(p, Q) \in \mathrm{lfp}(\mathcal{D}_2(A, B))$.

Preuve
On suppose $\mathtt{frb}(p_1, q_1) \in \mathrm{lfp}(\mathcal{D}_1(A, B))$. Ce littéral provient d'une clause produite par (Frb_1) ou (Frb_2). Par symétrie, il est suffisant de ne considérer que le premier cas. La clause concernée est de la forme $\mathtt{frb}(p_1, q_1) :- \mathtt{acc}(p_2, q_2)$. La règle ($\mathrm{Frb}_1$) présuppose $A \models p_1@p_2$ et $B \not\models q_1@q_2$, d'où $q_1 \notin Q_1(q_2)$. La règle (Frb_1^c) produit la clause $\mathtt{frb}^c(p_1, Q_1^B(q_2)) :- \mathtt{acc}(p_2, q_2)$ dans $\mathcal{D}_2(A, B)$. Comme $\mathtt{acc}(p_2, q_2) \in \mathrm{lfp}(\mathcal{D}_1(A, B))$, on a aussi $\mathtt{acc}(p_2, q_2) \in \mathrm{lfp}(\mathcal{D}_2(A, B))$. De ce fait, la précédente clause de $\mathcal{D}_2(A, B)$ est applicable et l'on ajoute $\mathtt{frb}^c(p_1, Q_1^B(q_2))$ à $\mathrm{lfp}(\mathcal{D}_2(A, B))$. L'argument inverse est similaire.□

Lemme 2.14
Pour $\mathcal{D} = \mathcal{D}_2(A, B)$, $p \in \mathrm{sta}(A)$ *et* $q \in \mathrm{sta}(B)$, *on a :*

$$\begin{aligned} l(p) &= \sum_{Q \subseteq \mathrm{sta}(B)} \mathrm{lfp}^{\#}_{\mathcal{D}}(\mathtt{frb}^c(p, Q)) \\ l(p, q) &= \sum_{Q \subseteq \mathrm{sta}(B) \text{ tq. } q \in Q} \mathrm{lfp}^{\#}_{\mathcal{D}}(\mathtt{frb}^c(p, Q)) \end{aligned}$$

Preuve
Le résultat découle des définitions. Nous le détaillons seulement pour le premier cas.

$$\begin{aligned} l(p) &= \begin{array}{l} \#\{(p', q') \mid A \models p'@p \wedge A, B \models \mathsf{acc}(p', q')\} \\ + \; \#\{(p', q') \mid A \models p@p' \wedge A, B \models \mathsf{acc}(p', q')\} \end{array} \\ &= \begin{array}{l} \#\{(p', q') \mid \mathtt{frb}^c(p, Q_2^B(q')) :- \mathtt{acc}(p', q') \in \mathcal{D} \wedge \mathtt{acc}(p', q') \in \mathrm{lfp}(\mathcal{D})\} \\ + \; \#\{(p', q') \mid \mathtt{frb}^c(p, Q_1^B(q')) :- \mathtt{acc}(p', q') \in \mathcal{D} \wedge \mathtt{acc}(p', q') \in \mathrm{lfp}(\mathcal{D})\} \end{array} \\ &= \sum_{Q \subseteq \mathrm{sta}(B)} \mathrm{lfp}^{\#}_{\mathcal{D}}(\mathtt{frb}^c(p, Q)) \end{aligned}$$

□

On peut ainsi calculer tous les compteurs $l(p)$ et $l(p, q)$ en temps linéaire dans la taille de $\mathrm{lfp}(\mathcal{D}_2(A, B))$ d'après le corollaire 2.2.

L'inclusion $\mathcal{L}(A) \subseteq \mathcal{L}(B)$ peut être vérifiée de la même manière que précédemment, à la différence près que les compteurs $l(p)$ et $l(p, q)$ sont désormais calculés plus efficacement

à partir de $\mathtt{lfp}(\mathcal{D}_2(A, B))$. En effet, il suffit de considérer uniquement la partie accessible de $A \times B$ en ne produisant que les clauses nécessaires issues de (Acc_3). L'application des règles (Frb_1^c) et (Frb_2^c) peut se faire en temps $\mathcal{O}(|A| \times |\mathtt{sta}(B)| + |\mathtt{rul}(B)|)$, ce qui, en pratique, se trouve être beaucoup plus petit que $\mathcal{O}(|A \times B|)$.

2.3.5 Détection “au plus tôt” de l'échec de l'inclusion

Nous abordons maintenant le problème de la vérification de $A, B \models \mathsf{fail}_1$ qui est effectuée à la fin, c'est-à-dire après le calcul du point fixe de $\mathcal{D}_2(A, B)$, par la boucle qui suit :

for all $\mathtt{acc}(p, q) \in \mathtt{lfp}(\mathcal{D}_2(A, B))$ **do**
  if $l(p) > l(p, q)$ **then**
    return false
return true

À la place, nous testons $A, B \models \mathsf{fail}_1$ à la volée de la façon suivante. Lorsqu'un littéral $\mathtt{acc}(p, q)$ est inféré pendant le calcul du point fixe de $\mathcal{D}_2(A, B)$, on vérifie en temps constant si la valeur courante des compteurs satisfait $l(p) > l(p, q)$, autrement dit si $\mathtt{frb}(p, q)$ est impliqué par un littéral $\mathtt{frb}^c(p, Q)$ inféré avant avec $q \notin Q$. Cette approche demande une mise à jour des compteurs à la volée.

La difficulté principale survient lors de la dérivation d'un littéral $\mathtt{frb}^c(p, Q)$ *après* un littéral $\mathtt{acc}(p, q)$. Il faudrait dans ce cas vérifier pour tout $q \in \mathtt{sta}(B) \setminus Q$ si $\mathtt{acc}(p, q)$ a été inféré *avant*, ce qui reviendrait à énumérer le complémentaire de Q.

Il s'avère que cet obstacle peut être contourné en mettant en place la politique de priorité suivante. On suppose que les littéraux de la forme $\mathtt{acc}(p, q)$ sont toujours inférés avec la plus petite priorité ; dit autrement, dès qu'un autre littéral peut être inféré dans le même temps, il l'est avant.

Notre algorithme “au plus tôt” calcule donc le plus petit point fixe de $\mathcal{D}_2(A, B)$ en suivant les priorités décrites au-dessus. Les compteurs $l(p)$ et $l(p, q)$ sont mis à jour à la volée. Dès lors qu'un littéral $\mathtt{acc}(p, q)$ est inféré, on teste l'inégalité $l(p) > l(p, q)$. Si cette dernière est vérifiée alors l'inclusion échoue et l'algorithme retourne faux, autrement il continue et renvoie vrai à la toute fin.

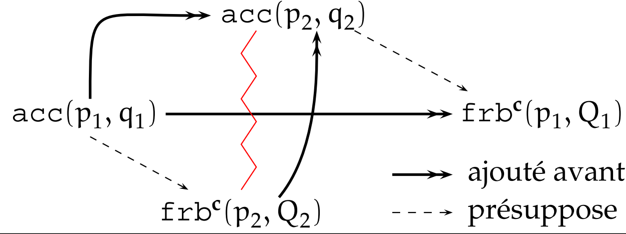


Figure 33. *Détection au plus tôt de l'échec de l'inclusion :* $\mathtt{frb}^c(p_2, Q_2)$ *a été ajouté à* $\mathtt{lfp}(\mathcal{D}_2(A, B))$ *avant* $\mathtt{acc}(p_2, q_2)$.

Lemme 2.15

L'algorithme "au plus tôt" détecte correctement $A, B \models \mathsf{fail}_1$ *si un littéral* $\mathtt{acc}(p, q)$ *est inféré avant un littéral* $\mathtt{frb}^c(p, Q)$ *avec* $q \notin Q$.

Preuve

Cette situation est illustrée à la fig. 33. Le littéral $\mathtt{frb}^c(p_1, Q_1)$ provient d'une clause produite par la règle (Frb_1^c) et d'un autre littéral $\mathtt{acc}(p_2, q_2)$ ajouté à $\mathtt{lfp}(\mathcal{D}_2(A, B))$ auparavant :

$$\frac{A \models p_1 @ p_2 \qquad Q_1 = Q_1^B(q_2)}{\mathtt{frb}^c(p_1, Q_1) :- \mathtt{acc}(p_2, q_2).}$$

On montre par contradiction que le littéral $\mathtt{acc}(p_1, q_1)$ a été ajouté à $\mathtt{lfp}(\mathcal{D}_2(A, B))$ avant $\mathtt{acc}(p_2, q_2)$. Si tel n'était pas le cas, le littéral $\mathtt{frb}^c(p_1, Q_1)$ aurait été ajouté avant $\mathtt{acc}(p_1, q_1)$, ce qui contreviendrait à notre politique de priorité.

Comme $\mathtt{acc}(p_1, q_1)$ appartient au point fixe de $\mathcal{D}_2(A, B)$, on peut appliquer la clause issue de la transformation (Frb_2^c) :

$$\frac{A \models p_1 @ p_2 \qquad Q_2 = Q_2^B(q_1)}{\mathtt{frb}^c(p_2, Q_2) :- \mathtt{acc}(p_1, q_1).}$$

Notons que $q_1 \in Q_1^B(q_2)$ ssi $q_2 \in Q_2^B(q_1)$, et donc $q_2 \notin Q_2$ puisque $q_1 \notin Q_1$. Par conséquent, $\mathtt{acc}(p_2, q_2), \mathtt{frb}^c(p_2, q_2) \in \mathtt{lfp}(\mathcal{D}_2(A, B))$ implique $A, B \models \mathsf{fail}_1$. L'échec de l'inclusion est bien détecté par l'algorithme "au plus tôt" car $\mathtt{frb}^c(p_2, Q_2)$ est ajouté à $\mathtt{lfp}(\mathcal{D}_2(A, B))$ avant $\mathtt{acc}(p_2, q_2)$. □

2.3.6 Incrémentalité

On s'intéresse dans cette section à l'incrémentalité du test d'inclusion. Il s'agit ici d'observer dans quelle mesure une modification ultérieure des automates A et/ou B, donc de leurs langages, implique ou non de recalculer les structures de données calculées auparavant afin de vérifier de nouveau l'inclusion. On étudie en particulier l'ajout d'ϵ-règles à l'automate A.

L'incrémentalité de l'algorithme d'inclusion est un point crucial pour nous. En effet, dans l'induction de requêtes guidée par schéma présentée plus loin, on ajoute ultérieurement des ϵ-règles à l'automate A. Ces règles modélisent les opérations de fusion d'états $p = p'$ par $p \overset{\epsilon}{\to} p'$ et $p' \overset{\epsilon}{\to} p$ au cours du processus d'apprentissage.

Les points fixes des programmes Datalog peuvent être calculés de manière incrémentale avec l'ajout de nouvelles clauses. Toutefois, la politique de priorité peut poser problème. Il ne serait pas correct d'appliquer des clauses prioritaires après alors qu'elles auraient dû l'être avant en raison de leur priorité. Cela entraînerait la réexécution de certains calculs.

Dans notre cas, l'ajout d'ϵ-règles à l'automate A est sans dommage. Celles-ci sont transformées par (Acc_2) en clauses qui ont une faible priorité et qui ne remettent pas en cause la validité des clauses déjà existantes [4]. Il n'est donc pas nécessaire de refaire des calculs après un ajout ultérieur d'ϵ-règles à l'automate A. Notons que l'incrémentalité présuppose naturellement la détection de l'échec au plus tôt.

2.4 Inclusion dans les automates *stepwise* factorisés

Dans cette section, nous assouplissons l'hypothèse de déterminisme sur l'automate B en vue de traiter efficacement l'inclusion dans les DTD déterministes. Pour cela, nous étendons le test d'inclusion précédent aux automates d'arbres *stepwise* factorisés déterministes.

4. Ce ne serait pas le cas, par exemple, si l'on ajoutait des règles de la forme $q_1@q_2 \to q$ à l'automate B, ce qui changerait la valeur des ensembles $Q_2^B(q_1)$.

2.4.1 Test d'inclusion en temps $\mathcal{O}(|A| \times |F|)$

L'idée générale du test d'inclusion $\mathcal{L}(A) \subseteq \mathcal{L}(F)$, où F est un automate factorisé déterministe, est de simuler l'inclusion $\mathcal{L}(A) \subseteq \mathcal{L}(\mathsf{b}(F))$ étudiée précédemment, mais sans expliciter $\mathsf{b}(F)$ puisque sa taille est en $\mathcal{O}(|sta(F)|^2)$ dans le pire des cas.

Validité de fail_0 et fail_2

Étant donnés un automate *stepwise* A et un automate factorisé déterministe F, nous caractérisons d'abord $A, \mathsf{b}(F) \models \mathsf{fail}_0$ et $A, \mathsf{b}(F) \models \mathsf{fail}_2$ en fonction de A et de F.

Lemme 2.16

1. $A, \mathsf{b}(F) \models \mathsf{fail}_0$ *si et seulement si* $\exists a \to p \in rul(A)$ *et* $\nexists q \in sta(F)$ *tq.* $a \to q \in rul(F)$;
2. $A, \mathsf{b}(F) \models \mathsf{fail}_2$ *si et seulement si* $\exists p \in fin(A), \exists q \in sta(F)$ *tq.* $A, B \models \mathsf{acc}(p, q)$ *et* $\forall r \in sta(F)\ \big(q \xrightarrow{\epsilon}{}_F^{\leqslant 1} r \Rightarrow r \notin fin(F)\big)$.

Preuve

La première assertion découle de la règle de construction (E_1) de $\mathsf{b}(F)$. Pour la seconde, remarquons que $q \notin fin(F)$ ssi $\forall r \in sta(F)\ \big(q \xrightarrow{\epsilon}{}_F^{\leqslant 1} r \Rightarrow r \notin fin(F)\big)$. La quantification universelle est sans conséquence puisque pour chaque état q il y a au plus un état r tq. $q \xrightarrow{\epsilon}{}_F^{\leqslant 1} r$. On peut donc conclure simplement :

$$\begin{aligned} A, B \models \mathsf{fail}_2 &\Leftrightarrow \exists p \in fin(A), \exists q \notin fin(B) \text{ tq. } A, B \models \mathsf{acc}(p, q) \\ &\Leftrightarrow \exists p \in fin(A), \exists q \in sta(F) \text{ tq.} \\ &\quad\ A, B \models \mathsf{acc}(p, q) \wedge \forall r \in sta(F)\ \big(q \xrightarrow{\epsilon}{}_F^{\leqslant 1} r \Rightarrow r \notin fin(F)\big) \end{aligned}$$

□

L'accessibilité dans $\mathsf{b}(F)$ implique l'accessibilité dans F, mais l'inverse n'est pas nécessairement vrai. Par exemple, dans la figure 17, l'état ok est accessible dans $F_{\leqslant n}$ mais pas dans $\mathsf{b}(F_{\leqslant n})$. Cela illustre un détail de la construction de $\mathsf{b}(F)$, essentiel pour la préservation du déterminisme (cf. prop. 1.9). Cette différence s'étend à l'accessibilité dans $A \times F$ et $A \times \mathsf{b}(F)$.

Afin d'éviter toute ambiguïté, pour $p \in sta(A)$ et $q \in sta(F)$, on dénote l'accessibilité dans le produit $A \times F$ comme suit :

$$A, F \models \mathsf{f.acc}(p, q) \overset{\text{déf.}}{\Leftrightarrow} (p, q) \text{ est accessible dans } A \times F$$

$$(\text{Acc}_{1a}) \quad \frac{a \to p \in \mathtt{rul}(A) \qquad a \to q \in \mathtt{rul}(F)}{\mathtt{acc}(p, q).}$$

$$(\text{Acc}_{2a}) \quad \frac{p' \xrightarrow{\epsilon} p \in \mathtt{rul}(A) \qquad q \in \mathtt{sta}(F)}{\mathtt{acc}(p, q) \text{ :- } \mathtt{acc}(p', q).}$$

$$(\text{Acc}_{3a}) \quad \frac{p_1@p_2 \to p \in \mathtt{rul}(A) \qquad q_1@q_2 \to q \in \mathtt{rul}(F)}{\mathtt{acc}(p, q) \text{ :- } \mathtt{f.acc}(p_1, q_1), \mathtt{f.acc}(p_2, q_2).}$$

$$(\text{F.Acc}) \quad \frac{p \in \mathtt{sta}(A) \qquad q \xrightarrow{\epsilon}{}_F^{\leqslant 1} r}{\mathtt{f.acc}(p, r) \text{ :- } \mathtt{acc}(p, q).}$$

$$(\text{Fail}_{0a}) \quad \frac{a \to p \in \mathtt{rul}(A) \qquad \nexists q \in \mathtt{sta}(F) \text{ tq. } a \to q \in \mathtt{rul}(F)}{\mathtt{fail}_0.}$$

$$(\text{Fail}_{2a}) \quad \frac{p \in \mathtt{fin}(A) \qquad \forall r \in \mathtt{sta}(F) \ \left(q \xrightarrow{\epsilon}{}_F^{\leqslant 1} \Rightarrow r \notin \mathtt{fin}(F)\right)}{\mathtt{fail}_2 \text{ :- } \mathtt{acc}(p, q).}$$

Figure 34. *Transformation d'automates* A, F *en un programme Datalog* $\mathcal{D}_0(A, F)$ *pour tester* $A, B \models \mathsf{fail}_0$ *et* $A, B \models \mathsf{fail}_2$ *avec* $B = \mathsf{b}(F)$.

Le lemme suivant est une conséquence directe de ce qui précède.

Lemme 2.17
$A, \mathsf{b}(F) \models \mathsf{acc}(p, q) \Rightarrow A, F \models \mathsf{f.acc}(p, q)$.

Les règles de transformation présentées dans la figure 34 définissent un programme Datalog $\mathcal{D}_0(A, F)$ dont le but est de calculer tous les littéraux `acc` et `f.acc` valides, c'est-à-dire qui induisent $A, \mathsf{b}(F) \models \mathsf{acc}(p, q)$ et $A, F \models \mathsf{f.acc}(p, q)$. De plus, les clauses produites par les règles de transformation (Fail_{0a}) et (Fail_{2a}) permettent d'inférer des littéraux $\mathtt{fail}_0$ et $\mathtt{fail}_2$ afin de vérifier $A, \mathsf{b}(F) \models \mathsf{fail}_0$ et $A, \mathsf{b}(F) \models \mathsf{fail}_2$ selon les termes du lemme 2.16.

Il reste à s'assurer que le programme $\mathcal{D}_0(A, F)$ permet effectivement d'inférer tous les littéraux `acc` et `f.acc` valides, ainsi que $\mathtt{fail}_0$ et $\mathtt{fail}_2$ le cas échéant. Ceci est montré par le lemme qui suit.

Lemme 2.18
Soit $B = \mathsf{b}(F)$ *et* $L = \mathtt{lfp}(\mathcal{D}_0(A, F))$*, alors :*

1. $A, B \models \mathsf{acc}(p, q)$ *si et seulement si* $\mathtt{acc}(p, q) \in L$ *;*
2. $A, F \models \mathsf{f.acc}(p, q)$ *si et seulement si* $\mathtt{f.acc}(p, q) \in L$ *;*
3. $A, B \models \mathsf{fail}_0$ *si et seulement si* $\mathtt{fail}_0 \in L$ *;*
4. $A, B \models \mathsf{fail}_2$ *si et seulement si* $\mathtt{fail}_2 \in L$.

Preuve
Les quatre implications de la droite vers la gauche peuvent être montrées par induction simultanée de la définition du plus petit point fixe. Pour les implications de la gauche vers la droite, on procède comme suit.

1. On montre que $(p, q) \in eval_{A\times B}(t)$ implique $\mathtt{acc}(p, q) \in L$ par induction sur la structure de t. Cela requiert l'utilisation de la règle (E_2) de $\mathsf{b}(F)$.
2. On montre que $(p, q) \in eval_{A\times B}(t)$ implique $\mathtt{f.acc}(p, q) \in L$ par induction sur la structure de t.
3. Déduit du lem. 2.16 et de la règle de transformation (Fail_{0a}).
4. Déduit du lem. 2.16, de la règle de transformation (Fail_{2a}) et de l'assertion 1 ci-dessus. □

Le programme $\mathcal{D}_0(A, F)$ est obtenu en temps $\mathcal{O}(|A| \times |F|)$ à partir des automates A et F, sa taille est donc en $\mathcal{O}(|A| \times |F|)$. Par conséquent (cf. th. 2.1), son plus petit point fixe peut être calculé en temps $\mathcal{O}(|A| \times |F|)$. La proposition suivante est une conséquence directe du lemme précédent.

Proposition 2.19
Vérifier $A, \mathsf{b}(F) \models \mathsf{fail}_0$ *et* $A, \mathsf{b}(F) \models \mathsf{fail}_2$ *peut être effectué en temps* $\mathcal{O}(A| \times |F|)$.

Preuve
Il suffit de tester la présence de $\mathtt{fail}_0$, resp. $\mathtt{fail}_2$, dans $\mathtt{lfp}(\mathcal{D}_0(A, F))$. □

Validité de fail_1

Il nous reste à caractériser $A, \mathsf{b}(F) \models \mathsf{fail}_1$ en fonction de A et de F.

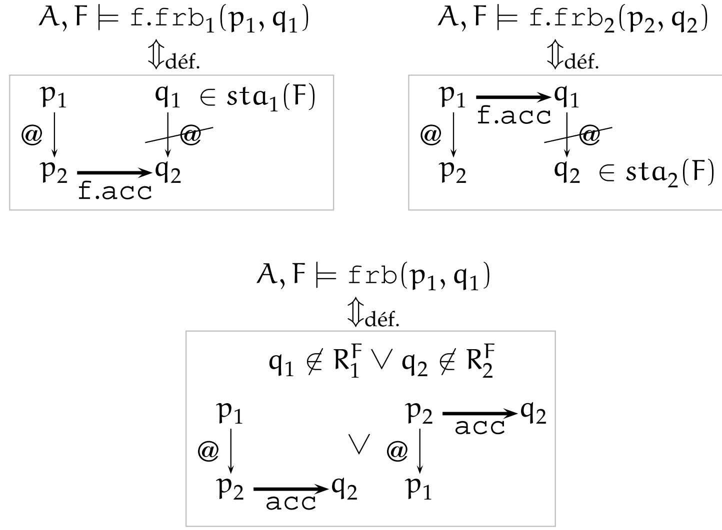


Figure 35. *Sémantique des prédicats* $\texttt{f.frb}_1(p, q)$*,* $\texttt{f.frb}_2(p, q)$ *et* $\texttt{frb}(p, q)$ *pour les automates* A *et* F*.*

Nous introduisons ci-dessous des prédicats $\texttt{f.frb}_1(p, q)$, $\texttt{f.frb}_2(p, q)$ et $\texttt{frb}(p, q)$ pour les automates A et F. Leur sémantique est illustrée à la figure 35.

$$A, F \models \mathsf{f.frb}_1(p_1, q_1) \overset{\text{déf.}}{\Leftrightarrow} \begin{cases} q_1 \in \mathtt{sta}_1(F) \wedge \exists p_2, q_2 \text{ tq.} \\ \quad A, F \models \mathsf{f.acc}(p_2, q_2) \wedge A \models p_1 \underline{@} p_2 \wedge F \not\models q_1 \underline{@} q_2 \end{cases}$$

Notons que $A, F \models \mathsf{f.frb}_1(p, q)$ n'implique pas toujours $A, \mathsf{b}(F) \models \mathsf{frb}(p, q)$ puisque $q_2 \in \mathtt{sta}_2(F)$ n'est pas une condition nécessaire dans la définition précédente. La sémantique de $A, F \models \mathsf{f.frb}_2(p, q)$ est symétrique.

Le troisième prédicat est doté de la sémantique suivante, où $R_i^F = \{q \mid \exists r \in \mathtt{sta}_i(F) \text{ tq. } q \xrightarrow{\epsilon}{}_F^{\leqslant 1} r\}$ pour $i \in \{1, 2\}$:

$$A, F \models \mathsf{frb}(p_1, q_1) \overset{\text{déf.}}{\Leftrightarrow} \begin{cases} \exists q_2 \text{ tq. } (q_1 \notin R_1^F \vee q_2 \notin R_2^F) \wedge \exists p_2 \text{ tq.} \\ \quad A, F \models (p_1 \underline{@} p_2 \vee p_2 \underline{@} p_1) \wedge A, F \models \mathsf{acc}(p_2, q_2) \end{cases}$$

Comme le montre la preuve du lemme suivant, $A, F \models \mathsf{frb}(p, q)$ implique $A, \mathsf{b}(F) \models \mathsf{frb}(p, q)$, mais pas inversement.

Lemme 2.20

$A, \mathsf{b}(F) \models \mathsf{frb}(p, q)$ *si et seulement si l'une des deux propriétés suivantes est vérifiée :*

1. *il existe* $i \in \{1, 2\}$ *tel que* $A, F \models \mathsf{f.frb}_i(p, r)$ *où* r *est l'unique état de sorte* i *avec* $q \xrightarrow{\epsilon}{}_F^{\leqslant 1} r$ *;*

2. $A, F \models \mathsf{frb}(p, q)$.

Preuve
Soit $B = \mathsf{b}(F)$. Pour l'implication de gauche à droite, on suppose $A, B \models \mathsf{frb}(p_1, q_1)$. Par définition (cf. lem. 2.8), il existe p_2, q_2 tels que soit (a) $A \models p_1@p_2$ et $B \not\models q_1@q_2$, soit (b) $A \models p_2@p_1$ et $B \not\models q_2@q_1$. Par symétrie, il suffit de traiter le cas (a). L'assertion 1 du lem. 2.18 montre que $A, B \models \mathsf{acc}(p_2, q_2)$ implique $A, F \models \mathsf{acc}(p_2, q_2)$. Nous considérons deux cas exhaustifs :

1. Cas où $q_1 \in R_1^F \wedge q_2 \in R_2^F$.
   Il existe un unique état $r_1 \in sta_1(F)$, resp. $r_2 \in sta_2(F)$, tq. $q_1 \xrightarrow{\epsilon}{}_F^{\leqslant 1} r_1$, resp. $q_2 \xrightarrow{\epsilon}{}_F^{\leqslant 1} r_2$. Dans ce cas, $B \not\models q_1@q_2$ équivaut à $F \not\models r_1@r_2$. De $A, B \models \mathsf{acc}(p_2, q_2)$, il s'ensuit $A, F \models \mathsf{f.acc}(p_2, r_2)$, et donc $A, F \models \mathsf{f.frb}_1(p_1, r_1)$.
2. Cas où $q_1 \notin R_1^F \vee q_2 \notin R_2^F$.
   Par définition, cela implique directement $A, F \models \mathsf{frb}(p_1, q_1)$.

Pour l'implication dans l'autre direction, on doit considérer chacune des deux assertions séparément.

1. Par symétrie, il suffit de traiter le cas $i = 1$. On peut donc supposer que l'unique $r_1 \in sta_1(F)$ avec $q_1 \xrightarrow{\epsilon}{}_F^{\leqslant 1} r_1$ satisfait $A, F \models \mathsf{f.frb}_1(p_1, r_1)$. Par définition, il existe p_2 et r_2 tels que $A \models p_1@p_2$ et $A, F \models \mathsf{f.acc}(p_2, r_2)$. D'après les assertions 2 et 1 du lem. 2.18, il existe q_2 tel que $A, B \models \mathsf{acc}(p_2, q_2)$ et $q_2 \xrightarrow{\epsilon}{}_F^{\leqslant 1} r_2$. Dans ce cas, $F \not\models r_1@r_2$ équivaut à $B \not\models q_1@q_2$. D'où $A, B \models \mathsf{frb}(p_1, q_1)$.
2. On suppose $A, F \models \mathsf{frb}(p_1, q_1)$ et on montre $A, B \models \mathsf{frb}(p_1, q_1)$. Par définition, il existe q_2 tq. $q_1 \notin R_1^F \vee q_2 \notin R_2^F$ et p_2 tq. $A \models p_1@p_2 \vee p_2@p_1$ et $A, B \models \mathsf{acc}(p_2, q_2)$. Par symétrie, il suffit de traiter le cas $A \models p_1@p_2$. De $q_1 \notin R_1^F \vee q_2 \notin R_2^F$, il s'ensuit $B \not\models q_1@q_2$, et donc $A, B \models \mathsf{frb}(p_1, q_1)$. □

Notre objectif est de vérifier l'échec $A, \mathsf{b}(F) \models \mathsf{fail}_1$, lorsqu'il est déclenché par $A, \mathsf{b}(F) \models \mathsf{acc}(p, q)$ et $A, F \models \mathsf{frb}(p, q)$, en temps $\mathcal{O}(|A| \times |F|)$. Pour ce faire, on pourrait déduire naïvement un programme Datalog de taille $\mathcal{O}(|A| \times |sta(F)|^2)$ à partir de la définition de $A, F \models \mathsf{frb}(p, q)$. Le programme Datalog $\mathcal{D}_1(A, F)$, présenté figure 36, étend $\mathcal{D}_0(A, F)$ afin de résoudre ce problème en temps $\mathcal{O}(|A| \times |F|)$. Pour éviter l'explosion quadratique, il s'appuie sur un nouveau prédicat $\mathtt{acc}(p, _)$, dont la sémantique est donnée ci-dessous :

$$A, \mathsf{b}(F) \models \mathsf{acc}(p, _) \overset{\text{déf.}}{\Leftrightarrow} A, F \models \exists q \text{ tq. } \mathsf{acc}(p, q)$$

$$(\mathrm{Acc}_4)\quad \frac{p \in \mathrm{sta}(A) \qquad q \in \mathrm{sta}(F)}{\texttt{acc}(p, _) :- \texttt{acc}(p, q).}$$

$$(\mathrm{Frb}_{1a})\quad \frac{A \models p_1 @ p_2 \qquad q_1 \notin R_1^F}{\texttt{frb}(p_1, q_1) :- \texttt{acc}(p_2, _).}$$

$$(\mathrm{Frb}_{2a})\quad \frac{A \models p_1 @ p_2 \qquad q_2 \notin R_2^F}{\texttt{frb}(p_2, q_2) :- \texttt{acc}(p_1, _).}$$

$$(\mathrm{Fail}_{1a})\quad \frac{p \in \mathrm{sta}(A) \qquad q \in \mathrm{sta}(F)}{\texttt{fail}_1 :- \texttt{acc}(p, q), \texttt{frb}(p, q).}$$

Figure 36. *Transformation d'automates* A *et* F *en un programme Datalog* $\mathcal{D}_1(A, F)$ *pour tester* $A, F \models \mathsf{fail}_1$.

Les littéraux $\texttt{acc}(p, _)$ valides sont inférés par le programme $\mathcal{D}_1(A, F)$ à partir de clauses produites par (Acc_4) et $\mathcal{D}_0(A, F)$. Les autres clauses de $\mathcal{D}_1(A, F)$ permettent de vérifier si l'on peut inférer $\texttt{fail}_1$ à partir de littéraux $\texttt{frb}(p, q)$ valides.

Lemme 2.21
$A, F \models \exists p \exists q$ *tq.* $\mathsf{acc}(p, q) \wedge \mathsf{frb}(p, q)$ *si et seulement si* $\texttt{fail}_1 \in \mathrm{lfp}(\mathcal{D}_1(A, F))$.

Preuve
La correction ("$\Leftarrow$") est évidente. Pour la complétude ("$\Rightarrow$"), on suppose $A, F \models \mathsf{acc}(p_1, q_1) \wedge \mathsf{frb}(p_1, q_1)$. Par définition de $A, F \models \mathsf{frb}(p_1, q_1)$, on a $A \models p_1 @ p_2$, $A, F \models \mathsf{acc}(p_2, q_2)$ et $q_1 \notin R_1^F \vee q_2 \notin R_2^F$. Par symétrie, il suffit de traiter le cas $q_1 \notin R_1^F$. Soit $L_1 = \mathrm{lfp}(\mathcal{D}_1(A, F))$. L'assertion 1 du lem. 2.18 montre $\texttt{acc}(p_2, q_2) \in L_1$, et donc $\texttt{acc}(p_2, _) \in L_1$ par la règle (Acc_4) ; de là, on déduit $\texttt{frb}(p_1, q_1) \in L_1$ par la règle (Frb_{1a}). On a aussi $\texttt{acc}(p_1, q_1) \in L_1$ par le lem. 2.18. D'où $\texttt{fail}_1 \in L_1$ par la règle (Fail_{1a}). □

Le programme $\mathcal{D}_1(A, F)$ est obtenu en temps $\mathcal{O}(|A| \times |F|)$ à partir de A et de F. En effet, $\mathcal{D}_0(A, F)$ est calculé en temps $\mathcal{O}(|A| \times |F|)$ et chacune des nouvelles clauses pour $\mathcal{D}_1(A, F)$ ne dépend à la fois que d'un élément de A et d'un élément de F. La taille de $\mathcal{D}_1(A, F)$ est donc en $\mathcal{O}(|A| \times |F|)$.

La prochaine étape consiste à tester $A, F \models \mathsf{frb}_i(p, q)$ pour tout p, q en temps $\mathcal{O}(|A| \times |F|)$. Nous ne considérons que le cas $i = 1$ du fait de la symétrie. De façon analogue au cas non

factorisé, nous définissons les compteurs suivants :

$$l_1(p) = \#\{(p', q') \mid A \models p'\underline{@}p \wedge A, F \models \mathsf{f.acc}(p', q')\} \tag{2.3}$$

$$l_1(p, q) = \#\{(p', q') \mid A \models p'\underline{@}p \wedge A, F \models \mathsf{acc}(p', q') \wedge F \models q'\underline{@}q\} \tag{2.4}$$

Lemme 2.22
Pour tout $q \in \mathtt{sta}_1(F)$, $A, F \models \mathsf{f.frb}_1(p, q)$ *si et seulement si* $l_1(p) > l_1(p, q)$.

Preuve
Par définition, $l_1(p) \geqslant l_1(p, q)$ pour tout p, q. On a $l_(p) > l_(p, q)$ ssi $\exists p', q'$ tq. $A \models p'\underline{@}p \wedge A, F \models \mathsf{acc}(p', q') \wedge F \models q'\underline{@}q$. Comme $q \in \mathtt{sta}_1(F)$ par hypothèse, cela équivaut à $A, F \models \mathsf{f.frb}_1(p, q)$. □

Théorème 2.23
L'inclusion $\mathcal{L}(A) \subseteq \mathcal{L}(F)$*, où* A *est un automate stepwise avec* ϵ*-règles et* F *un automate factorisé déterministe, peut être vérifiée en temps* $\mathcal{O}(|A| \times |F|)$.

Preuve
On calcule d'abord le plus petit point fixe de $\mathcal{D}_1(A, F)$ en temps $\mathcal{O}(|A| \times |F|)$. L'inclusion échoue si celui-ci contient $\mathtt{fail}_0$, $\mathtt{fail}_1$ ou $\mathtt{fail}_2$. Sinon, on calcule les valeurs de tous les compteurs $l_i(p)$ et $l_i(p, q)$ en temps $\mathcal{O}(|A| \times |F|)$, similairement au cas non factorisé (cf. fig. 30). S'il existe $\mathtt{acc}(p, q) \in \mathtt{lfp}(\mathcal{D}_1(A, F))$ pour lequel $l_i(p) > l_i(p, q)$ alors l'inclusion échoue également. Autrement, elle est vérifiée. □

2.4.2 Algorithme efficace

Nous présentons une méthode similaire au cas non factorisé pour calculer les valeurs des compteurs efficacement. Nous introduisons des nouveaux prédicats $\mathtt{f.frb}^c_i(p, Q)$, pour $p \in \mathtt{sta}(A)$, $Q \subseteq \mathtt{sta}_i(F)$ et $i \in \{1, 2\}$, avec la sémantique suivante :

$$A, F \models \mathsf{f.frb}^c_i(p, Q) \overset{\text{déf.}}{\Leftrightarrow} \forall q \in \mathtt{sta}_i(F) \setminus Q \; \big(A, F \models \mathsf{f.frb}_i(p, q)\big)$$

Le programme Datalog $\mathcal{D}_2(A, F)$ étend $\mathcal{D}_1(A, F)$ avec des clauses produites par les règles de transformation (F.Frb$^c_1$) et (F.Frb$^c_2$) présentées figure 37.

$$(\text{F.Frb}_1^c) \quad \frac{A \models p_1@p_2 \qquad q_2 \in \mathtt{sta}_2(F)}{\mathtt{f.frb}_1^c(p_1, Q_1^F(q_2)) \text{ :- } \mathtt{f.acc}(p_2, q_2)}$$

$$(\text{F.Frb}_2^c) \quad \frac{A \models p_1@p_2 \qquad q_1 \in \mathtt{sta}_1(F)}{\mathtt{f.frb}_2^c(p_2, Q_2^F(q_1)) \text{ :- } \mathtt{f.acc}(p_1, q_1)}$$

Figure 37. $\mathcal{D}_2(A, F)$ *étend* $\mathcal{D}_1(A, F)$ *avec des clauses pour les littéraux* $\mathtt{f.frb}_i^c(p, Q)$.

Lemme 2.24

Pour $\mathcal{D} = \mathcal{D}_2(A, F)$, $p \in \mathtt{sta}(A)$ *et* $q \in \mathtt{sta}_i(F)$, *on a :*

$$l_i(p) = \sum_{Q \subseteq \mathtt{sta}_i(F)} \mathtt{lfp}^\#_{\mathcal{D}}(\mathtt{f.frb}_i^c(p, Q))$$

$$l_i(p, q) = \sum_{Q \subseteq \mathtt{sta}_i(B) \text{ tq. } q \in Q} \mathtt{lfp}^\#_{\mathcal{D}}(\mathtt{f.frb}_i^c(p, Q))$$

Preuve

La preuve est similaire à celle du lemme 2.14. □

Lemme 2.25

$\mathcal{D}_2(A, F)$ *peut être calculé en temps* $\mathcal{O}(|A| \times |F|)$ *à partir de* A *et de* F.

Preuve

La preuve est analogue à celle du lem. 2.12. Les clauses produites par les deux règles (F.Frb_i^c) peuvent être regroupées de la même façon que les clauses produites par les règles (Frb_j^c). Les ensembles Q_i^F sont de taille $\mathcal{O}(|F|)$ et sont utilisés au plus $\mathcal{O}(|A|)$ fois par les règles (F.Frb_i^c). Donc la taille globale des clauses produites par ces règles est en $\mathcal{O}(|A| \times |F|)$. □

Un exemple est donné à la figure 38. L'automate A_u reconnaît tous les arbres (c.-à-d. $\mathcal{L}(A_u) = T_\Sigma^u$ avec $\Sigma = \{a, b, f\}$) et l'automate factorisé $F_{\leqslant n}$ l'ensemble des codages curryfiés des arbres d'arité non bornée ayant au plus n enfants par nœud. Le programme Datalog $\mathcal{D}_2(A_u, F_{\leqslant n})$ infère les littéraux $\mathtt{f.frb}_1^c(\text{all}, \{0, \ldots, n-1\})$ et $\mathtt{f.frb}_2^c(\text{all}, \{\text{ok}\})$. Le premier implique $A_u, \mathsf{b}(F_{\leqslant n}) \models \mathsf{f.frb}_1(\text{all}, n)$ et donc $A_u, \mathsf{b}(F_{\leqslant n}) \models \mathsf{fail}_1$. Le second est une tautologie car $\mathtt{sta}_2(F_{\leqslant n}) = \{\text{ok}\}$.

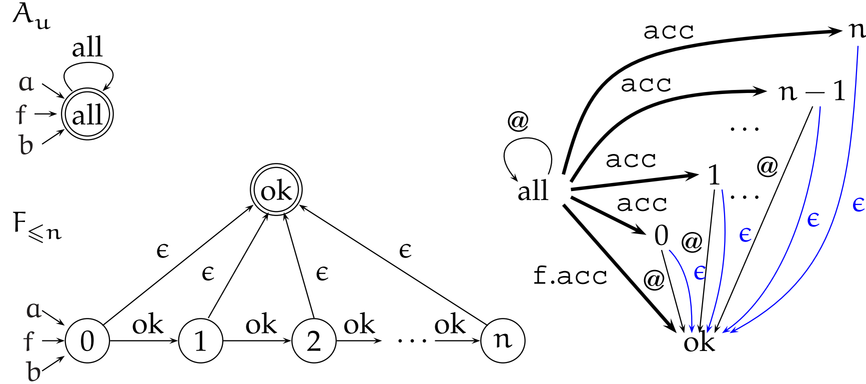


Figure 38. *Exemple d'exécution de l'algorithme avec* A_u *et* $F_{\leqslant n}$*. Les littéraux* $\mathtt{f.frb}^c_1(\text{all}, \{0, \dots, n-1\})$ *et* $\mathtt{f.frb}^c_2(\text{all}, \{\text{ok}\})$ *sont inférés.*

Par rapport au cas non factorisé, notre algorithme ne peut plus inférer des clauses pour les littéraux $\mathtt{frb}^c(p, Q)$ efficacement à cause des ϵ-règles. À la place, il infère des clauses pour des littéraux $\mathtt{f.frb}^c_i(p, Q)$ combinés avec les littéraux $\mathtt{f.acc}(p, q)$ (cf. fig. 37). Les ϵ-règles sont utilisées pour inférer les clauses pour ces littéraux $\mathtt{f.acc}(p, q)$ (fig. 34) et dans les règles de transformation (Frb_{ia}) (fig. 36) qui permettent de traiter les cas où l'évaluation s'arrête dans un état ne pouvant être converti dans la sorte voulue par aucune ϵ-règle.

2.4.3 Détection "au plus tôt" de l'échec de l'inclusion

Nous montrons comment vérifier $A, F \models \mathsf{fail}_1$ à la volée. On suppose que les compteurs $l_i(p)$ et $l_i(p, q)$ sont systématiquement à jour et, comme dans le cas non factorisé, que les littéraux $\mathtt{acc}(p, q)$ sont inférés avec la priorité la plus petite (également plus petite que $\mathtt{f.acc}(p, q)$).

L'algorithme "au plus tôt" calcule $\mathrm{lfp}(\mathcal{D}_2(A, F))$ en retournant faux dès que l'un des faits $\mathtt{fail}_0$, $\mathtt{fail}_1$ ou $\mathtt{fail}_2$ est produit. Les compteurs $l_i(p)$ et $l_i(p, q)$ sont incrémentés lorsqu'un littéral $\mathtt{f.frb}^c_i(p, Q)$ est ajouté au point fixe. Pour tout nouveau littéral $\mathtt{acc}(p, q)$ inféré, l'algorithme vérifie la condition $A, F \models \mathsf{f.frb}_i(p, q)$, où $i \in \{1, 2\}$ et $r \in \mathrm{sta}_i(F)$ avec $q \xrightarrow{\epsilon}{}^{\leqslant 1}_F r$, en testant les inégalités $l_1(p) > l_1(p, r)$ et $l_2(p) > l_2(p, r)$. Si l'une de celles-ci est vraie alors il retourne faux. Autrement, il poursuit le calcul du point fixe et renvoie vrai à la fin.

Nous montrons que tester les compteurs à la volée est une condition suffisante.

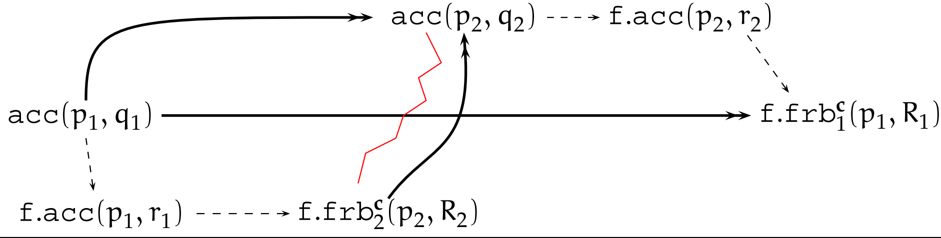


Figure 39. *Détection au plus tôt de l'échec de l'inclusion pour le cas* $i = 1$ *:* $\mathtt{f.frb}^{c}_{2}(p_2, R_2)$ *a été ajouté à* $\mathtt{lfp}(\mathcal{D}_2(A, F))$ *avant* $\mathtt{acc}(p_2, q_2)$.

Lemme 2.26

L'algorithme "au plus tôt" détecte la condition d'échec $A, F \models \mathsf{fail}_1$ *si un littéral* $\mathtt{acc}(p, q)$ *est ajouté dans* $\mathtt{lfp}(\mathcal{D}_2(A, F))$ *avant le littéral* $\mathtt{f.frb}^{c}_{i}(p, Q)$*, où* $r \notin Q$ *est l'unique état tel que* $q \xrightarrow{\epsilon}{}^{\leqslant 1}_{F} r$.

Preuve

Par symétrie, il suffit de considérer le cas $i = 1$. On suppose $\mathtt{f.frb}^{c}_{1}(p_1, R_1)$ ajouté au point fixe de $\mathcal{D}_2(A, F)$ après $\mathtt{acc}(p_1, q_1)$, où $q_1 \xrightarrow{\epsilon}{}^{\leqslant 1}_{F} r_1$ et $r_1 \in \mathtt{sta}_1(F) \setminus R_1$. La figure 39 illustre ce cas de figure. La présence du littéral $\mathtt{f.frb}^{c}_{1}(p_1, R_1)$ dans $\mathtt{lfp}(\mathcal{D}_2(A, F))$ est justifiée par un littéral $\mathtt{f.acc}(p_2, r_2)$ ajouté auparavant ; ce dernier provient lui-même d'un littéral $\mathtt{acc}(p_2, q_2)$. Les clauses permettant ces inférences, ainsi que leurs préconditions, sont rappelées ci-dessous, où $R_1 = Q^{F}_{1}(r_2)$.

$$\frac{A \models p_1@p_2 \qquad r_2 \in \mathtt{sta}_2(F)}{\mathtt{f.frb}^{c}_{1}(p_1, R_1) \coloneq \mathtt{f.acc}(p_2, r_2).} \qquad \frac{p_2 \in \mathtt{sta}(A) \qquad q_2 \xrightarrow{\epsilon}{}^{\leqslant 1}_{F} r_2}{\mathtt{f.acc}(p_2, r_2) \coloneq \mathtt{acc}(p_2, q_2)}$$

En raison de la politique de priorité, le littéral $\mathtt{acc}(p_1, q_1)$ est nécessairement inféré avant $\mathtt{acc}(p_2, q_2)$. De ce fait, on peut appliquer les clauses suivantes, où $R_2 = Q^{F}_{2}(r_1)$.

$$\frac{p_1 \in \mathtt{sta}(A) \qquad q_1 \xrightarrow{\epsilon}{}^{\leqslant 1}_{F} r_1}{\mathtt{f.acc}(p_1, r_1) \coloneq \mathtt{acc}(p_1, q_1)} \qquad \frac{A \models p_1@p_2 \qquad r_1 \in \mathtt{sta}_1(F)}{\mathtt{f.frb}^{c}_{2}(p_2, R_2) \coloneq \mathtt{f.acc}(p_1, r_1).}$$

Comme $r_1 \in Q^{F}_{1}(r_2)$ ssi $r_2 \in Q^{F}_{2}(r_1)$, il s'ensuit que $r_1 \notin R_1$ implique $r_2 \notin R_2$. D'où la présence des littéraux $\mathtt{acc}(p_2, q_2)$ et $\mathtt{f.frb}^{c}_{2}(p_2, R_2)$ provoque l'échec de l'inclusion $A, F \models \mathsf{fail}_1$. Du fait de la politique de priorité, le littéral $\mathtt{f.frb}^{c}_{2}(p_2, R_2)$ est inféré avant $\mathtt{acc}(p_2, q_2)$, de telle sorte que cette condition d'échec est correctement détectée par l'algorithme "au plus tôt". □

Comme dans le cas non factorisé, cet algorithme est incrémental relativement à l'ajout d'ϵ-règles à l'automate A. Notre test d'inclusion dans un automate factorisé déterministe est donc aussi efficace que dans le cas non factorisé.

2.5 Inclusion dans les automates descendant

Nous montrons comment vérifier l'inclusion dans les automates déterministes descendant en réduisant le problème au test d'inclusion dans les automates de mots déterministes. Pour en analyser finement la complexité, nous utilisons le théorème 2.11 dans le cas des mots, vus comme des arbres sur des signatures unaires.

Proposition 2.27
Soient Σ un alphabet d'arité bornée, A et B deux automates d'arbres sur Σ. Si B est déterministe descendant alors on peut décider l'inclusion $L(A) \subseteq L(B)$ en temps $\mathcal{O}(|A| \times |B|)$.

Nous basons l'algorithme sur le fait que les langages d'arbres reconnus par les automates déterministes descendant sont clos par chemin.

Lemme 2.28
Si L_2 est clos par chemin alors $L_1 \subseteq L_2$ si et seulement si $\mathrm{paths}(L_1) \subseteq \mathrm{paths}(L_2)$.

Preuve
L'implication de la gauche vers la droite est triviale : par définition, L_2 contient tous les chemins de L_1. Dans l'autre sens, on suppose $\mathrm{paths}(L_1) \subseteq \mathrm{paths}(L_2)$. Si $t_1 \in L_1$ alors $\mathrm{paths}(t_1) \subseteq \mathrm{paths}(L_1) \subseteq \mathrm{paths}(L_2)$. Donc $t_1 \in \mathrm{path\text{-}clos}(L_2)$ et $\mathrm{path\text{-}clos}(L_2) = L_2$ par hypothèse de clôture par chemin. □

Notons qu'il n'est pas nécessaire de supposer que L_1 soit clos par chemin.

Preuve (proposition 2.27)
Pour tout automate d'arbres A sur Σ, on construit un automate de séquences (possiblement non déterministe) P_A sur un sous-ensemble fini de $\Sigma \uplus \mathbb{N}$ tel que $L(P_A) = \mathrm{paths}(L(A))$. Les règles de P_A sont définies comme suit :

$$
\begin{aligned}
\mathrm{rul}(P_A) = \quad & \{q \xrightarrow{f \cdot i} q_i \mid f(q_1, \ldots, q_n) \in \mathrm{rul}(A) \text{ pour } 1 \leqslant i \leqslant n\} \\
\cup\ & \{a \to q \mid a \to q \in \mathrm{rul}(A)\} \\
\cup\ & \{q \xrightarrow{\epsilon} q' \mid q \xrightarrow{\epsilon} q' \in \mathrm{rul}(A)\}
\end{aligned}
$$

La première sorte de règles lit deux symboles à la fois mais peut facilement être divisée en deux règles lisant chacune un seul symbole. La construction de P_A est clairement en temps $\mathcal{O}(|A|)$. De plus, P_A est déterministe ssi A est déterministe descendant.

Étant donnés deux automates A et B sur Σ tels que B est déterministe descendant, on peut décider l'inclusion $L(A) \subseteq L(B)$ en testant l'inclusion $L(P_A) \subseteq L(P_B)$, d'après le lemme précédent. Comme P_B est déterministe, cela peut être effectué en temps $\mathcal{O}(|P_A| \times |P_B|)$ indépendamment de la taille de Σ (cf. th. 2.11) et donc en temps $\mathcal{O}(|A| \times |B|)$. □

2.6 Inclusion dans les schémas XML

On s'intéresse dans cette section à l'extension de notre test d'inclusion pour les automates d'arbres aux DTD déterministes et aux DTD étendues *restrained competition* (donc aux schémas définis avec le langage W3C XML Schema).

Nous avons montré dans le chapitre 1 comment transformer les premières en automates *stepwise* factorisés déterministes et les secondes en automates déterministes descendant. Partant, nous obtenons un algorithme efficace pour l'inclusion d'automates d'arbres dans chacun de ces deux formalismes de schéma, ce qu'établissent les deux corollaires suivants.

Corollaire 2.29 (inclusion dans les DTD déterministes)
Soient un automate pour les arbres d'arité non bornée A *stepwise, frère-fils ou hedge, et une DTD déterministe* D*, tous deux sur un alphabet* Σ*. L'inclusion* $\mathcal{L}(A) \subseteq \mathcal{L}(D)$ *peut être vérifiée en temps* $\mathcal{O}(|A| \times |\Sigma| \times |D|)$.

Preuve
Si A est un automate *stepwise*, on transforme D en un automate *stepwise* factorisé déterministe F reconnaissant le même langage en temps $\mathcal{O}(|\Sigma| \times |D|)$ (th. 1.11) ; la taille de F est donc également en $\mathcal{O}(|\Sigma| \times |D|)$. Comme F est déterministe, on peut tester l'inclusion $\mathcal{L}(A) \subseteq \mathcal{L}(F)$ en temps $\mathcal{O}(|A| \times |\Sigma| \times |D|)$ grâce au th. 2.23. Si A est un automate frère-fils, on transforme D en automate déterministe descendant B reconnaissant le même langage en temps $\mathcal{O}(|\Sigma| \times |D|)$ (prop. 1.12) et on applique la prop. 2.27. Si A est un automate *hedge*, on transforme A en automate *stepwise* ou en automate frère-fils en temps linéaire (cf. prop. 1.6) et l'on utilise le test d'inclusion approprié. □

Corollaire 2.30 (inclusion dans les EDTD *restrained competition*)
Soient un automate d'arbres frère-fils ou hedge A *et une DTD étendue restrained competition déterministe* D*, tous deux sur* Σ*. L'inclusion* $\mathcal{L}(A) \subseteq \mathcal{L}(D)$ *peut être vérifiée en temps* $\mathcal{O}(|A| \times |\Sigma| \times |D|)$.

Preuve
Si A est un automate frère-fils, on transforme D en un automate d'arbres descendant déterministe B sur $\Sigma_\sharp$ qui reconnaît le même langage d'arbres d'arité non bornée *via* la proposition 1.12 en temps $\mathcal{O}(|\Sigma| \times |D|)$; la taille de B est donc également en $\mathcal{O}(|\Sigma| \times |D|)$. Comme B est déterministe, on teste ensuite l'inclusion $\mathcal{L}(A) \subseteq \mathcal{L}(B)$ en temps $\mathcal{O}(|A| \times |\Sigma| \times |D|)$ par la proposition 2.27. Si A est un automate *hedge*, on le transforme en automate frère-fils reconnaissant le même langage modulo l'encodage en temps linéaire avec la prop. 1.6, puis on applique le test d'inclusion précédent. □

Notons que l'on ne peut pas choisir A *stepwise* dans la mesure où il faudrait le transformer en automate frère-fils. À l'heure actuelle, nous ne savons pas si cette opération peut s'effectuer en temps linéaire dans la taille de A.

Martens *et al.* (2006, cf. sect. 10) ont présenté un algorithme similaire pour l'inclusion de DTD étendues *restrained competition*, mais restreint au cas où à la fois A et D sont déterministes. De plus, leur approche ne s'appuie pas sur les automates d'arbres mais sur les automates de mots et ils ne fournissent pas une analyse de complexité aussi précise que la nôtre.

2.7 Expériences

Afin de vérifier l'efficacité de notre algorithme d'inclusion en pratique, nous avons mené diverses expériences à la fois sur des données synthétiques (simples) et sur des données réalistes issues de nos jeux de tests pour l'induction de requêtes guidée par schéma.

2.7.1 Données synthétiques

Nous faisons varier les tailles d'automates A et F lors du test d'inclusion $\mathcal{L}(A) \subseteq \mathcal{L}(F)$.

Dans cette optique, nous définissons dans un premier temps $\mathtt{Mult}_n$ l'automate déterministe minimal pour le langage d'arbres de la forme $f(a, \ldots, a)$ où le nombre de feuilles a est un multiple de n.

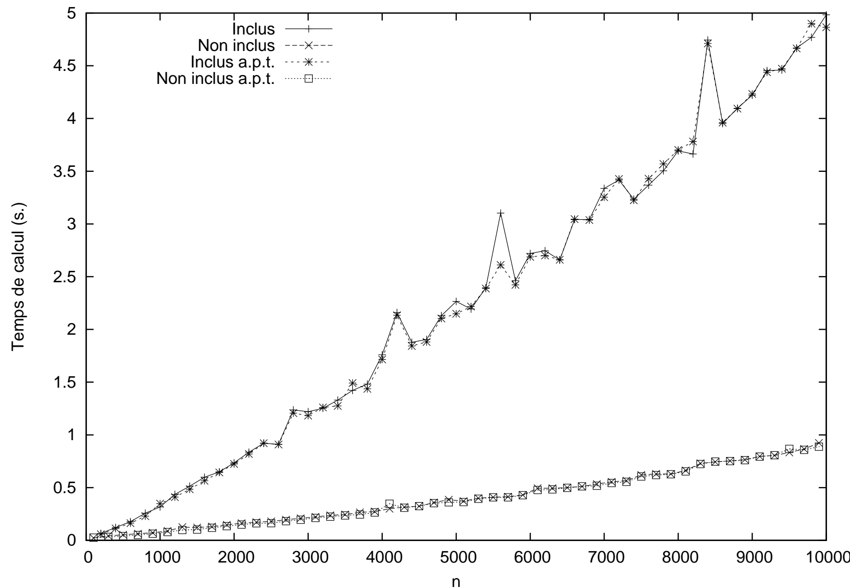


Figure 40. *Temps de calcul pour tester* $\mathcal{L}(\mathrm{Mult}_n) \subseteq \mathcal{L}(\mathrm{Mult}_{200})$.

Expérience 1

Le premier problème que nous étudions est de vérifier l'inclusion de Mult_n, pour n variant de 100 à 10 000 avec un incrément de 100, dans l'automate factorisé déterministe (minimal) Mult_{200}. Observons que cette inclusion est valide lorsque $n/100$ est un chiffre pair.

Le second problème est de vérifier l'inclusion de Mult_{400} dans Mult_n pour n variant entre 10 et 500 avec un incrément de 10. Ici, l'inclusion est valide lorsque $400/n$ est un nombre entier.

Nous estimons le temps de calcul pour des tests d'inclusion avec ou sans détection "au plus tôt" (noté a.p.t. dans les légendes des figures) de l'échec de l'inclusion. Nous distinguons le cas où l'inclusion est vérifiée du cas où elle ne l'est pas. Les résultats sont présentés figures 40 pour le premier problème et 41 pour le second.

Ces courbes permettent de vérifier que le temps de calcul pour tester l'inclusion $\mathcal{L}(A) \subseteq \mathcal{L}(F)$ est linéaire dans la taille de l'automate A (resp. F) une fois celle de F (resp. A) fixée. Les résultats de complexité théoriques sont ainsi confirmés [5].

5. Les irregularités observées dans le cas où l'inclusion est vérifiée (courbes fig. 40) sont dues à des cycles de récupération de mémoire.

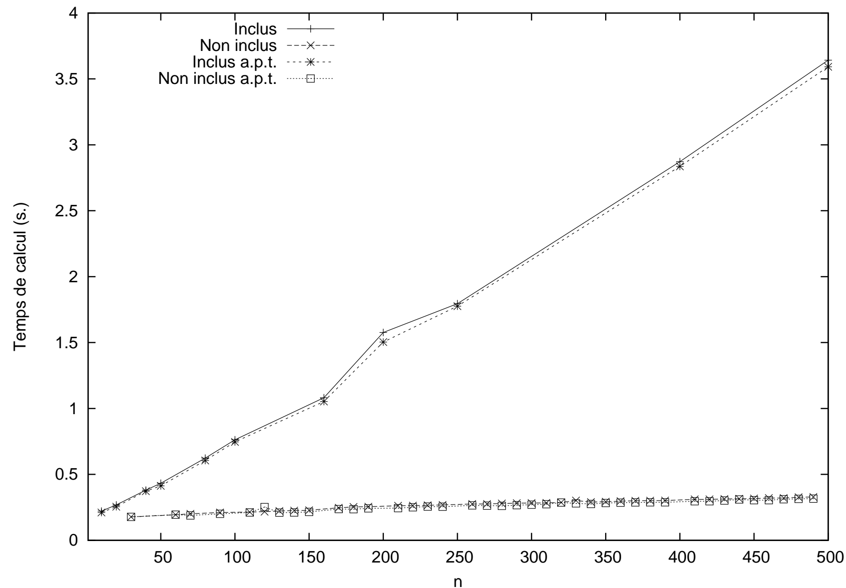


Figure 41. *Temps de calcul pour tester* $\mathcal{L}(\mathrm{Mult}_{400}) \subseteq \mathcal{L}(\mathrm{Mult}_n)$.

L'expérience permet également de constater que le temps de calcul est plus long lorsque l'inclusion est vraie que lorsqu'elle ne l'est pas. En effet, dans le second cas, l'échec de l'inclusion est détecté par l'algorithme sans qu'il soit nécessaire de calculer tous les états accessibles de l'automate produit. Toutefois, dans cette expérience, il n'y a pas d'échec de type $A, F \models \mathsf{fail}_1$; autrement dit, ici, le gain ne provient pas de l'effet de la détection "au plus tôt" mais des échecs du type $A, F \models \mathsf{fail}_2$ testés à la volée.

Expérience 2 : détection "au plus tôt"

Afin de montrer l'utilité de la détection "au plus tôt" de l'échec de l'inclusion (cas $A, F \models \mathsf{fail}_1$), nous considérons un autre exemple. Nous définissons l'automate déterministe minimal $\mathrm{Mult2}_n$ pour le langage des arbres de la forme $g(f(a, \ldots, a))$ où le nombre de feuilles a est un multiple de n.

Le problème est de tester l'inclusion de $\mathrm{Mult2}_n$, pour n variant de 100 à 10 000 avec un incrément de 100, dans l'automate factorisé déterministe (minimal) $\mathrm{Mult2}_{200}$. Les résultats sont présentés à la figure 42.

Dans le cas où l'inclusion échoue, le temps de calcul est approximativement cinq fois plus rapide que dans les autres cas. Il s'agit, ici, de l'effet de la détection "au plus tôt" de

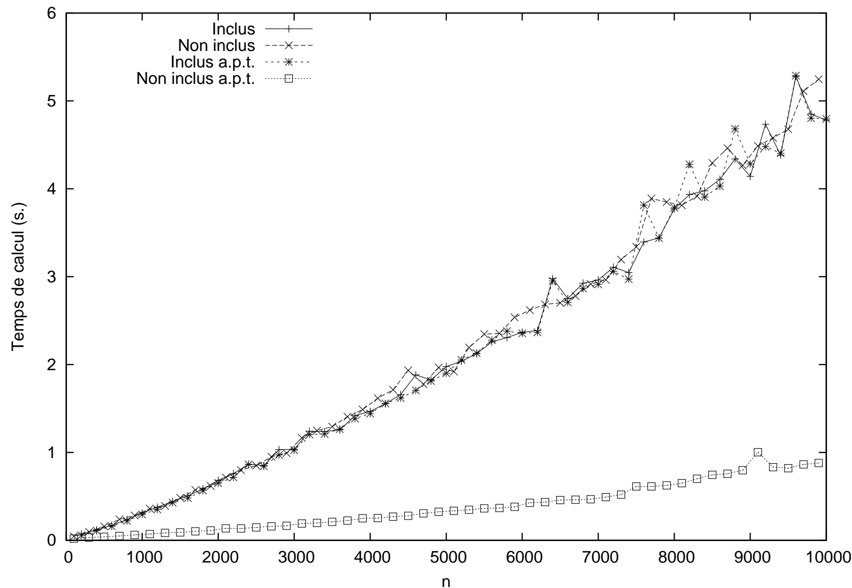


Figure 42. *Temps de calcul pour tester* $\mathcal{L}(\mathrm{Mult2}_n) \subseteq \mathcal{L}(\mathrm{Mult2}_{200})$.

l'échec de l'inclusion, démontrant ainsi l'intérêt de cette technique pour accélérer le test d'inclusion.

Notons que les temps de calcul pour le cas où l'inclusion est vérifiée sont similaires avec ou sans détection "au plus tôt" ; celle-ci n'induit donc pas de surcoût.

2.7.2 Données réalistes

Nous considérons maintenant des données réalistes issues de jeux de données pour le problème de l'induction de requêtes. Dans l'algorithme d'apprentissage, décrit en détail au chapitre suivant, un automate initial est calculé à partir d'un ensemble d'exemples et son langage est itérativement augmenté par fusion d'états. Une fusion est acceptée si le langage de l'automate qui en est issu satisfait un schéma (une DTD) donné. De ce fait, l'inclusion est fréquemment testée.

Nous nous intéressons au temps de calcul global de sessions d'apprentissage où les tests d'inclusion sont effectués avec ou sans détection "au plus tôt" (noté d.a.p.t. dans la légende). Nous utilisons la DTD du langage XHTML et les jeux de données *Okra*, *Bigbook*,

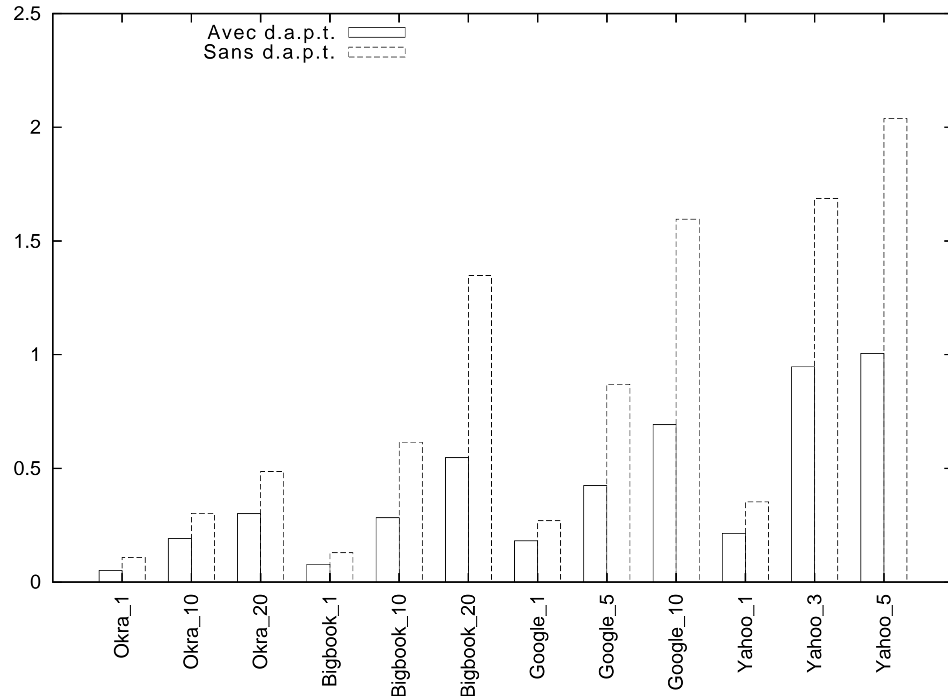


Figure 43. *Temps de calcul moyen (en s.) de sessions d'apprentissage avec ou sans détection "au plus tôt" de l'échec de l'inclusion* $A, F \models \mathsf{fail}_1$.

Google et *Yahoo* de Carme (2005) [6], en faisant varier le nombre d'arbres complètement annotés donnés en entrée de l'algorithme d'apprentissage. Les résultats sont présentés figure 43.

Il apparaît que l'algorithme d'inclusion s'exécute environ deux fois plus vite avec la détection "au plus tôt" de l'échec de l'inclusion (cas $A, F \models \mathsf{fail}_1$) pour tous les jeux de données. Cela signifie que cette situation survient fréquemment dans la pratique et que sa détection "au plus tôt" améliore amplement l'efficacité du test d'inclusion.

Plus généralement, cette expérience montre que l'algorithme d'inclusion présenté dans ce chapitre peut être utilisé pour des problèmes concrets. Comme nous le verrons plus loin, l'introduction des tests d'inclusion dans l'algorithme d'apprentissage n'augmente pas le temps de calcul ; au contraire, ils l'améliorent en empêchant des fusions d'états qui s'avèrent inutiles.

6. Ces jeux de données sont disponibles à l'adresse suivante : `http://www.grappa.univ-lille3.fr/~carme/WebWiki/DataSets.html`. Des détails sur le jeu de données *Yahoo* sont donnés au chapitre 5, sections 5.3.2 et suivante (p. 177).

3 Requêtes par automate

DIFFÉRENTS langages pour spécifier des requêtes de sélection de nœuds dans les arbres d'arité non bornée ont été suggérés depuis la fin des années 1960. Les plus répandus sont la logique du premier ordre ou, de façon équivalente, Core XPath 2.0, et la logique monadique du second ordre, ou Monadic Datalog, ou encore les automates d'arbres avec états de sélection. Dans ce chapitre, nous discutons des langages de requête basés sur les automates d'arbres déterministes, qui est le formalisme utilisé dans notre algorithme d'apprentissage. Cela nous conduira aux transducteurs de sélection de nœuds proposés par Carme *et al.* (2007).

Sommaire

3.1 Introduction **99**
3.2 Arbres annotés et projection **102**
3.3 Requêtes monadiques **103**
3.4 Transducteurs de sélection de nœuds (TSN) **103**
3.4.1 Définition d'une requête monadique avec un TSN 104
3.4.2 Calcul de la réponse d'une requête avec un TSN 106

3.1 Introduction

Dans le but de l'induction de requêtes guidée par schéma, nous souhaitons définir les requêtes par l'intermédiaire d'automates d'arbres déterministes, afin de pouvoir appliquer des algorithmes d'apprentissage existants. De plus, la classe d'automates d'arbres doit être suffisamment expressive pour capturer à la fois les requêtes XPath et les DTD déterministes.

XPath (Clark et DeRose, 1999; Berglund *et al.*, 2007) est le langage recommandé par le W3C pour définir des requêtes de sélection de nœuds dans les documents XML. Il est employé comme sous-langage dans plusieurs standards XML tels que XQuery (Boag *et al.*, 2007), XSLT (Clark, 1999) ou XProc (Walsh *et al.*, 2009). L'utilisation la plus simple de XPath consiste à définir des requêtes de sélection de nœuds monadiques, par exemple :

$$//\mathrm{region}[\mathrm{population}]/\mathrm{name}$$

Ici, on sélectionne tous les noms des régions pour lesquelles la population est connue. Dans l'arbre de la figure 44, seul un nœud est sélectionné par l'application de cette requête.

L'expressivité du langage XPath a été étudiée de façon approfondie. Marx (2005) a montré que le noyau logique de XPath version 2.0 (Core Xpath 2.0) est aussi expressif que la logique du premier ordre sur les arbres d'arité non bornée (FO). Il est assez facile de voir que les requêtes Core Xpath 2.0 peuvent être exprimées par des formules FO. La requête précédente, par exemple, peut être définie par une formule de logique du premier ordre

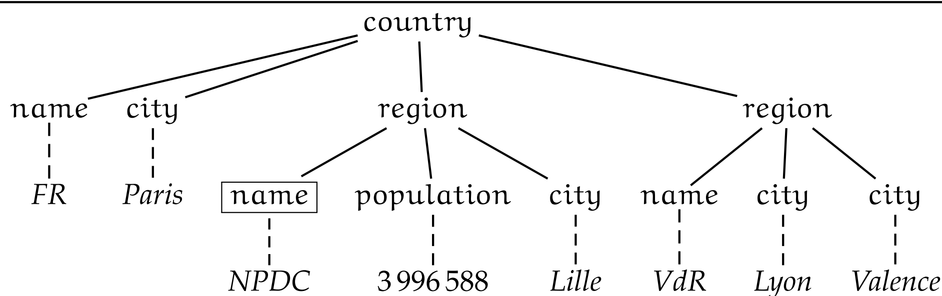


Figure 44. *Sélection des noms des régions pour lesquelles la population est connue.*

comme suit :

$$\exists y \; \text{child}^*(\text{root}, y) \wedge \text{lab}_{region}(y) \wedge \text{child}(y, x) \wedge \text{lab}_{name}(x) \wedge \\ \exists z \; \text{child}(y, z) \wedge \text{lab}_{population}(z)$$

Montrer l'inverse, à savoir que les formules FO peuvent être exprimées avec Core XPath 2.0, est nettement moins évident.

Toutefois, dans notre perspective, la logique du premier ordre n'est pas suffisamment expressive. En effet, certaines DTD ne peuvent être exprimées par aucune formule FO close, par exemple la DTD composée des deux règles $f \to (aa)^*$ et $a \to \epsilon$, où le nombre d'enfants de tous les nœuds étiquetés par f doit être pair. L'expressivité additionnelle requise peut être obtenue en utilisant la logique monadique du second ordre (MSO), comme le relèvent par exemple Gottlob et Koch (2002).

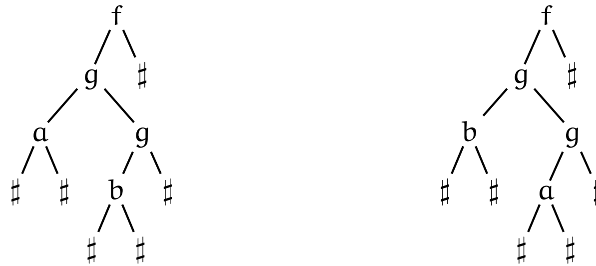
Le célèbre théorème de Thatcher et Wright (1968) montre, dans le cas des arbres d'arité bornée, que toute requête définissable par une formule de logique MSO peut aussi être définie par automate d'arbres et donc par Monadic Datalog. Le même résultat peut s'appliquer aux automates pour les arbres d'arité non bornée (Gottlob et Koch, 2004), par exemple en utilisant des codages binaires (Niehren *et al.*, 2005).

La question qui demeure est de savoir quel codage binaire est le plus adapté pour notre objectif. Comme nous l'avons montré dans la section 1.4, il est possible d'exprimer les DTD déterministes avec des automates déterministes descendants modulo le codage frère-fils, ou bien avec des automates déterministes ascendants (factorisés) interprétés sur le codage curryfié des arbres d'arité non bornée. Cependant, on ne peut pas choisir le déterminisme descendant car cela aurait pour effet de trop restreindre l'expressivité, si bien que certaines requêtes XPath ne pourraient plus être exprimées. À titre de contre-exemple, considérons la requête XPath booléenne suivante :

$$[f[g[a] \text{ and } g[b]]]$$

Les deux arbres ci-dessous sont acceptés par ce filtre :

Leurs codages frère-fils sont comme suit :



Mais tout automate déterministe descendant acceptant ces deux arbres binaires doit aussi, par définition, accepter les arbres ci-dessous :

Autrement dit, le filtre XPath devrait accepter les deux arbres d'arité non bornée :

Or, la définition de la requête ne le permet pas : deux sous-arbres $\mathtt{g}(\mathtt{a})$ et $\mathtt{g}(\mathtt{b})$ doivent coexister parmi les enfants du nœud $\mathtt{f}$.

Dans la suite du chapitre, nous nous intéressons au moyen de définir des requêtes monadiques par l'intermédiaire d'automates pour les arbres d'arité non bornée. Pour cela, nous utilisons les transducteurs de sélection de nœuds (TSN) proposés par Carme *et al.* (2007) ; nous montrons également comment calculer efficacement la réponse à une requête définie par un TSN.

3.2 Arbres annotés et projection

En préambule, nous introduisons les notions d'arbre annoté, d'arbre projeté et de projection d'automate, que nous utiliserons afin de donner une définition formelle des requêtes monadiques et des automates d'arbres qui les représentent.

Arbres annotés On note $\mathbb{B}$ l'ensemble des booléens avec $\mathbb{B} = \{0, 1\}$. Un *arbre annoté* sur Σ est un arbre sur $\Sigma \times \mathbb{B}$. L'arbre peut être d'arité bornée ou non ; nous explicitons seulement le cas des arbres d'arité non bornée, sachant que le cas des arbres d'arité bornée est identique.

Tout arbre annoté $s \in T^u_{\Sigma \times \mathbb{B}}$ peut être décomposé de façon unique en deux arbres $t \in T^u_\Sigma$ et $\beta \in T^u_{\mathbb{B}}$, ayant le même ensemble de nœuds $nod(s) = nod(t) = nod(\beta)$ tels que pour tout nœud v dans cet ensemble, on a $s(v) = (t(v), \beta(v))$, où $\beta(v)$ est appelé *annotation* de $t(v)$; on note $s = t * \beta$.

Le codage curryfié d'un arbre annoté sur Σ est un arbre binaire sur l'alphabet d'arité bornée $(\Sigma \times \mathbb{B})_@ = (\Sigma \times \mathbb{B}) \uplus \{@\}$, où les symboles de $\Sigma \times \mathbb{B}$ sont des constantes et @ l'unique symbole binaire. Remarquons que la représentation curryfiée d'un arbre annoté autorise ainsi des annotations seulement au niveau de ses feuilles.

Projection On dénote par Π_Σ (resp. $\Pi_{\mathbb{B}}$) la *projection* sur Σ (resp. $\mathbb{B}$) d'un arbre annoté $s = t * \beta \in T^u_{\Sigma \times \mathbb{B}}$ avec $\Pi_\Sigma(s) = t$ (resp. $\Pi_{\mathbb{B}}(s) = \beta$) ; la projection sur Σ d'un arbre annoté s est également appelée *composante* Σ de s. Par extension, si L est un ensemble d'arbres sur $\Sigma \times \mathbb{B}$, alors $\Pi_\Sigma(L)$ désigne la projection de tous les arbres de L sur la composante Σ. Notons que si $t * \beta \in L$, alors $t \in \Pi_\Sigma(L)$.

Soit A un automate (standard) sur $\Sigma \times \mathbb{B}$, l'automate $\Pi_\Sigma(A)$ est l'*automate projeté* de A sur Σ tel que $\mathcal{L}(\Pi_\Sigma(A)) = \Pi_\Sigma(\mathcal{L}(A))$. Les états et les états finaux de $\Pi_\Sigma(A)$ sont les mêmes que ceux de A ; les ϵ-règles de A sont également conservées dans $\Pi_\Sigma(A)$. Les autres règles de $\Pi_\Sigma(A)$ sont obtenues simplement à partir de celles de A comme suit :

$$\frac{(f, b)(p_1, \ldots, p_n) \to p \in rul(A)}{f(p_1, \ldots, p_n) \to p \in rul(\Pi_\Sigma(A))}$$

L'automate projeté de A sur Σ peut être calculé en temps linéaire dans la taille de A. Notons que si A est productif, sa projection $\Pi_\Sigma(A)$ l'est aussi. En revanche, contrairement à l'union, à l'intersection et au complémentaire, la projection ne préserve pas le déterminisme.

3.3 Requêtes monadiques

Nous définissons formellement les requêtes monadiques dans les arbres d'arité non bornée. Soit Σ un alphabet d'arité non bornée. Une *requête monadique* (ou *unaire*) $\mathcal{Q}$ dans les arbres sur Σ est une fonction totale qui associe un arbre $t \in T^u_\Sigma$ à un sous-ensemble de ses nœuds $\mathcal{Q}(t) \subseteq \mathrm{nod}(t)$. Par exemple, si $\mathcal{Q}$ est la requête monadique qui sélectionne les noms des régions pour lesquelles la population est connue, et t est l'arbre de la figure 44, alors $\mathcal{Q}(t)$ est le singleton $\{3 \cdot 1\}$.

Soit $\mathcal{Q}$ une requête monadique, on note $t * \mathcal{Q}(t)$ l'*arbre annoté par* $\mathcal{Q}$, c'est-à-dire, pour tout nœud $v \in \mathrm{nod}(t)$, on a :

$$\big(t * \mathcal{Q}(t)\big)(v) = \begin{cases} (t(v), 1) & \text{si } v \in \mathcal{Q}(t)\,; \\ (t(v), 0) & \text{sinon.} \end{cases}$$

Par commodité, on pourra confondre $\mathcal{Q}(t)$ avec un arbre sur $T^u_{\mathbb{B}}$.

Le *domaine* de la requête $\mathcal{Q}$, noté $\mathrm{dom}(\mathcal{Q})$, est l'ensemble des arbres de T^u_Σ pour lesquels celle-ci sélectionne au moins un nœud, c'est-à-dire :

$$\mathrm{dom}(\mathcal{Q}) = \{t \in T^u_\Sigma \mid \mathcal{Q}(t) \neq \emptyset\}$$

Le *langage* de $\mathcal{Q}$, dénoté $\mathcal{L}(\mathcal{Q})$, est l'ensemble des arbres de $\mathrm{dom}(\mathcal{Q})$ annotés par $\mathcal{Q}$, à savoir :

$$\mathcal{L}(\mathcal{Q}) = \{t * \mathcal{Q}(t) \in T^u_{\Sigma\times\mathbb{B}} \mid t \in \mathrm{dom}(\mathcal{Q})\}$$

3.4 Transducteurs de sélection de nœuds (TSN)

Les transducteurs de sélection de nœuds (TSN) sont des automates d'arbres qui permettent de définir une requête (Carme *et al.*, 2007). Le calcul de la réponse d'une requête représentée par un TSN peut être réalisé efficacement.

$$\frac{(a,1) \to p \in \mathrm{rul}(A)}{\mathtt{sel}(p).} \qquad \frac{p' \xrightarrow{\epsilon}_A p}{\mathtt{sel}(p) :- \mathtt{sel}(p').} \qquad \frac{p_1@p_2 \to p \in \mathrm{rul}(A)}{\begin{array}{c}\mathtt{sel}(p) :- \mathtt{sel}(p_1).\\ \mathtt{sel}(p) :- \mathtt{sel}(p_2).\end{array}}$$

$$\frac{(a,0) \to p \in \mathrm{rul}(A)}{\mathtt{sel}^c(p).} \qquad \frac{p' \xrightarrow{\epsilon}_A p}{\mathtt{sel}^c(p) :- \mathtt{sel}^c(p').} \qquad \frac{p_1@p_2 \to p \in \mathrm{rul}(A)}{\mathtt{sel}^c(p) :- \mathtt{sel}^c(p_1), \mathtt{sel}^c(p_2).}$$

Figure 45. *Transformation d'un automate* A *sur* $\Sigma \times \mathbb{B}$ *en un programme Datalog* $\mathcal{D}(A)$ *pour vérifier la propriété de non-nullité. Le prédicat* $\mathtt{sel}(p)$ *signifie que l'état* p *est "sélectionnant", autrement dit tout sous-arbre évalué dans cet état par* A *contient au moins une feuille annotée positivement ; le prédicat* $\mathtt{sel}^c(p)$*, similaire à l'accessibilité, indique que* p *est "non-sélectionnant", c'est-à-dire qu'il existe au moins un sous-arbre évalué par* A *dans cet état qui ne contient aucune feuille annotée positivement. Le langage de* A *est non-nul dès lors que tous ses états finaux sont sélectionnants et qu'aucun d'entre eux n'est non-sélectionnant ou, dit autrement,* $\forall p \in \mathrm{fin}(A)\big(\mathtt{sel}(p) \in \mathrm{lfp}(\mathcal{D}(A)) \wedge \mathtt{sel}^c(p) \notin \mathrm{lfp}(\mathcal{D}(A))\big)$*. Cela peut être vérifié en temps linéaire dans la taille de* A *:* $\mathcal{D}(A)$ *est obtenu en temps* $\mathcal{O}(|A|)$*, sa taille est également en* $\mathcal{O}(|A|)$*, et* $\mathrm{lfp}(\mathcal{D}(A))$ *peut donc être calculé en temps* $\mathcal{O}(|A|)$ *(th. 2.1).*

3.4.1 Définition d'une requête monadique avec un TSN

Un langage d'arbres annotés $L \subseteq T^u_{\Sigma\times\mathbb{B}}$ définit une relation $\mathcal{R}_L \subseteq T^u_\Sigma \times T^u_\mathbb{B}$ entre des arbres de même structure telle que $(t,\beta) \in \mathcal{R}_L \overset{\text{déf.}}{\Leftrightarrow} t * \beta \in L$. La relation $\mathcal{R}_L$ est dite *fonctionnelle* ou, de façon équivalente, le langage L *fonctionnel* si pour tout $t \in T^u_\Sigma$ il existe au plus un arbre $\beta \in T^u_\mathbb{B}$ tel que $t * \beta \in L$, autrement dit à tout arbre sur Σ est associé à au plus un arbre sur $\mathbb{B}$ dans le langage. De plus, L est *non-nul*, si pour tout $t * \beta \in L$ il existe au moins un nœud $\nu \in \mathrm{nod}(\beta)$ tel que $\beta(\nu) = 1$, autrement dit tout arbre du langage comporte au moins un nœud annoté positivement.

Définition 3.1 (Transducteur de sélection de nœuds (TSN))
Un *transducteur de sélection de nœuds* (TSN) sur Σ est un automate *stepwise* sur $\Sigma \times \mathbb{B}$ qui reconnaît un langage fonctionnel et non-nul d'arbres annotés.

Tout transducteur de sélection de nœuds A sur Σ définit une requête monadique $\mathcal{Q}_A$ telle que pour tout t dans T^u_Σ, on a :

$$\mathcal{Q}_A(t) = \{\nu \in \mathrm{nod}(t) \mid \exists \beta \in T^u_\mathbb{B} \text{ tq. } t * \beta \in \mathcal{L}^u(A) \wedge \beta(\nu) = 1\}$$

Inversement, une requête $\mathcal{Q}$ sur Σ est représentée par un transducteur de sélection de nœuds A si A définit $\mathcal{Q}$, autrement dit si $\mathcal{L}(\mathcal{Q}) = \mathcal{L}(\mathcal{Q}_A)$. ◀

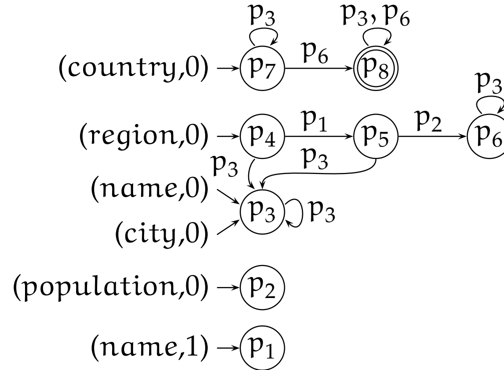


Figure 46. *Un transducteur de sélection de nœuds pour la requête qui sélectionne les noms des régions pour lesquelles la population est connue (cf. fig. 44).*

Dans la suite, nous considérons tout TSN A sur Σ comme un automate *stepwise* interprété sur les arbres d'arité non bornée [1] de $T^{u}_{\Sigma\times\mathbb{B}}$; de plus, nous ne faisons aucune hypothèse quant au déterminisme de A.

On peut vérifier en temps cubique $\mathcal{O}(|A|^3)$ si un tel automate A sur $\Sigma \times \mathbb{B}$ reconnaît un langage fonctionnel. Pour cela, il faut et il suffit que $\Pi_\Sigma(A)$ soit non ambiguë (cf. Carme *et al.* (2007)). La non-nullité de $\mathcal{L}(A)$ peut être vérifiée en temps linéaire dans la taille de A (cf. fig. 45). De ce fait, on peut décider en temps $\mathcal{O}(|A|^3 + |A|)$ si A est un transducteur de sélection de nœuds sur Σ. En terme d'expressivité, les TSN capturent la classe des requêtes définissables en logique monadique du second ordre, autrement dit les requêtes régulières (Niehren *et al.*, 2005).

La figure 46 propose un TSN pour représenter la requête monadique discutée page 99. Cet automate reconnaît le langage des arbres d'arité non bornée tels que les nœuds étiquetés par $\mathtt{name}$ suivis d'un nœud $\mathtt{population}$ et qui sont des enfants de $\mathtt{region}$ sont annotés positivement, tandis que tous les autres nœuds sont annotés négativement.

1. Le cas des arbres d'arité bornée peut être traité de façon analogue, c'est-à-dire *via* les automates *stepwise*, ou bien directement avec un automate standard. La différence entre les deux approches est que les annotations dans la représentation curryfiée d'un arbre annoté se situent nécessairement au niveau des feuilles, tandis qu'elles peuvent concerner des nœuds internes dans la représentation non curryfiée (interprétation directe).

3.4.2 Calcul de la réponse d'une requête avec un TSN

La réponse d'une requête monadique définie par un transducteur de sélection de nœuds peut être calculée efficacement en temps $\mathcal{O}(|A| \times |t|)$ pour un TSN A sur Σ et un arbre $t \in T^{u}_{\Sigma}$.

Soit $\mathcal{Q}$ sur Σ une requête monadique représentée par un TSN A. L'algorithme d'exécution procède en deux phases.

1. Dans une première phase ascendante, on évalue le codage curryfié de l'arbre t, dénoté t_c, avec l'automate projeté $\Pi_\Sigma(A)$ en conservant les états d'évaluation, appelés candidats dans la suite. Cette passe produit un arbre t'_c sur l'alphabet $\Sigma_@ \times 2^{sta(A)}$ tel que $nod(t'_c) = nod(t_c)$ et pour tout nœud $v \in nod(t'_c)$, on a :

$$t'_c(v) = \big(t_c(v), eval_{\Pi_\Sigma(A)}((t_c)_{|v})\big)$$

   Le calcul ici revient simplement à matérialiser le test d'appartenance de t au langage de $\Pi_\Sigma(A)$, d'où une complexité en $\mathcal{O}(|A| \times |t|)$.

2. Dans une seconde phase descendante, on sélectionne parmi les états candidats de t'_c ceux qui sont effectivement utilisés par A pour reconnaître un arbre annoté. Pour cela, on part des états finaux depuis la racine de t'_c et on regarde inductivement quelles règles de A peuvent s'appliquer pour chacun des deux sous-arbres. Au niveau d'une feuille, on peut déterminer si celle-ci est sélectionnée par la requête. Cette passe produit un arbre t''_c sur $(\Sigma \times \mathbb{B})_@$ tel que $nod(t''_c) = nod(t'_c)$ par l'intermédiaire d'une fonction $select^{\downarrow}_A : T_{\Sigma_@ \times 2^{sta(A)}} \times 2^{sta(A)} \to T^{bin}_{(\Sigma \times \mathbb{B})_@}$ définie récursivement comme suit :

$$select^{\downarrow}_A\big((a, Q), S\big) = \begin{cases} (a,1) & \text{si } (a,1) \to p \in rul(A) \text{ et } p \in Q \cap S \\ (a,0) & \text{sinon} \end{cases}$$

$$\begin{aligned} select^{\downarrow}_A\big((@(t'_1, t'_2), Q), S\big) = \ & @(t1, t2) \text{ avec} \\ & t_1 = select^{\downarrow}_A(t'_1, S_1), \\ & t_2 = select^{\downarrow}_A(t'_2, S_2), \\ & S_1 \times S_2 = \{(p_1, p_2) \in sta(A)^2 \mid \\ & \qquad p_1 @ p_2 \to p \in rul(A) \wedge p \in Q \cap S\} \end{aligned}$$

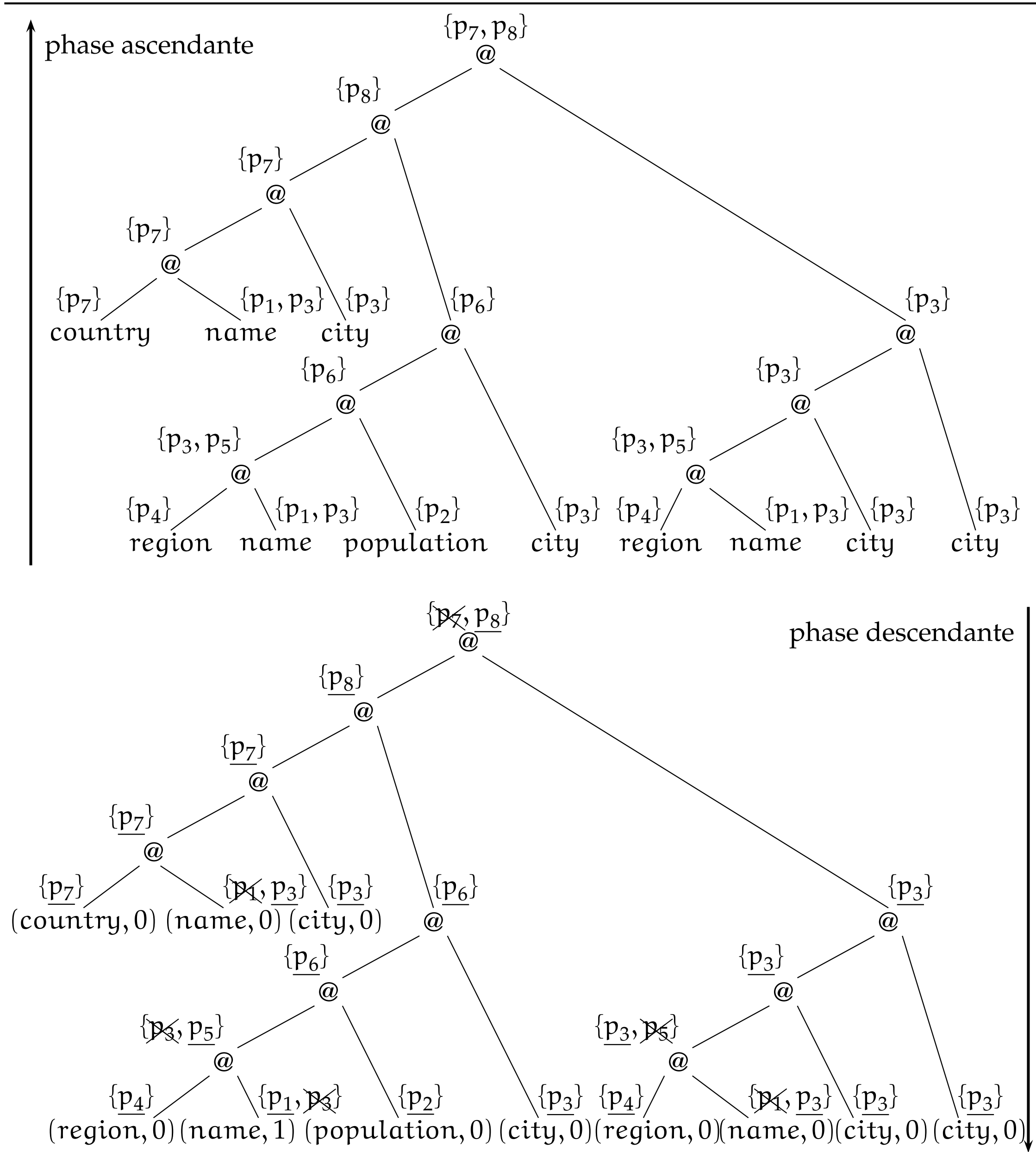


Figure 47. *Phases ascendante (en haut) et descendante (en bas) du calcul de la requête de la fig. 46 sur l'arbre de la fig. 44.*

L'appel initial à cette fonction est :

$$t''_c = \mathtt{select}^{\downarrow}_{A}(t'_c, \mathtt{fin}(A))$$

Ce calcul peut être réalisé en temps linéaire dans la taille de t'_c, qui est en $\mathcal{O}(|A| \times |t|)$ d'après l'étape précédente.

Notons que la fonction peut échouer si un ensemble $Q \cap S$ est vide ou s'il n'est pas possible d'associer une annotation à une feuille, auxquels cas la requête ne sélectionnera aucun nœud. Par ailleurs, la propriété de fonctionnalité garantit qu'il y a au plus une seule annotation booléenne par feuille.

On conclut le calcul en décurryfiant l'arbre binaire t''_c, ce qui donne un arbre annoté t''. L'ensemble des nœuds sélectionnés par la requête $\mathcal{Q}$ appliquée à l'arbre t est l'ensemble des nœuds de t'' avec une annotation positive.

La figure 47 montre les arbres curryfiés produits par les phases ascendantes et descendantes de la requête représentée par l'automate de la figure 46, appliquée à l'arbre d'arité non bornée de la figure 44. Dans la première phase, chaque nœud de l'arbre curryfié est annoté par un ensemble d'états candidats obtenus *via* l'évaluation de l'automate projeté. Dans la seconde phase, on souligne les états pouvant être utilisés par le TSN pour l'annotation et l'on raye les autres. Au niveau des feuilles, on retrouve les annotations positives ou négatives en regardant quelle règle constante s'applique.

4

Induction de requêtes guidée par schéma

DANS ce chapitre, nous étudions deux moyens d'introduire les connaissances offertes par le schéma dans les algorithmes d'induction de requêtes. Le premier consiste à assurer que la requête inférée est consistante avec le schéma. Pour cela, nous utilisons les algorithmes d'inclusion d'automates présentés au chapitre 2. Nous montrons que les requêtes consistantes avec un schéma sont apprenables dans une variante du modèle classique d'identification à la limite de Gold (1967). La seconde idée est d'utiliser des informations contenues dans le schéma pour enrichir les heuristiques d'élagage. Celles-ci permettent de réduire la taille des exemples donnés à l'algorithme et d'améliorer son efficacité. En contrepartie, l'élagage restreint la classe des requêtes pouvant être identifiées. C'est pourquoi nous introduisons la notion de requêtes stables par élagage, et prouvons que cette classe est apprenable.

Sommaire

4.1 Introduction **111**
4.2 Inférence de requêtes régulières **112**
4.2.1 Inférence d'automates d'arbres 113
4.2.2 Algorithme $\mathcal{RPNI}$ pour les langages réguliers d'arbres 115
4.2.3 Algorithme t$\mathcal{RPNI}$ pour les requêtes monadiques 117
4.3 Induction de requêtes consistantes avec un schéma **121**
4.3.1 Requêtes consistantes avec un schéma 122
4.3.2 Vérification de la consistance par un test d'inclusion 123
4.3.3 Vérification de la consistance par un typage des états 129
4.4 Élagage guidé par schéma **134**
4.4.1 Fonction élagage 135
4.4.2 Élaguer relativement au schéma 136
4.4.3 Transducteurs de sélection de nœuds élagueurs 137
4.4.4 Algorithme d'apprentissage avec élagage t$\mathcal{RPNI}^{\mathcal{P}}_{\mathrm{D}}$ 144
4.5 Apprenabilité des requêtes stables par élagage **148**
4.5.1 Stabilité des requêtes monadiques 150
4.5.2 Résultat d'apprenabilité des requêtes stables par élagage 152
4.5.3 Élagages utilisés en pratique 158

4.1 Introduction

L'induction supervisée de requêtes (ou *wrappers*) permet a un utilisateur de définir de nouvelles requêtes XML sans que celui-ci dispose de compétences techniques préalables. Kushmerick (2002) rapporte un ensemble de travaux sur l'induction de *wrappers* basés sur les chaînes. L'inconvénient de ces approches est qu'elles ignorent la structure arborescente des documents XML. D'autres techniques d'apprentissage ont été envisagées pour intégrer cette dimension : programmation logique inductive (Cohen *et al.*, 2002), classification (Gilleron *et al.*, 2006), champs aléatoires conditionnels (Jousse *et al.*, 2006), inférence d'automates d'arbres (Kosala *et al.*, 2003; Raeymaekers *et al.*, 2008; Carme *et al.*, 2007).

Nous nous intéressons, dans ce chapitre, à l'induction de requêtes *guidée par schéma*. Autrement dit, nous souhaitons exploiter les informations fournies par le schéma XML, lorsqu'il est disponible, afin d'aider l'algorithme d'apprentissage. Mitchell (1990), par exemple, explique que l'utilisation de biais appropriés est essentielle pour la réussite des algorithmes d'apprentissage par généralisations successives. Le schéma apporte naturellement des connaissances sur le domaine, permettant de ce fait de définir des contraintes sur le processus d'apprentissage. Cependant, dans les travaux cités plus haut, aucun n'a exploré cette possibilité.

Parmi les différentes approches pour l'induction de requêtes, l'inférence d'automates d'arbres apparaît comme l'une des rares — avec la programmation logique inductive (Muggleton et De Raedt, 1994) — pour laquelle il est assez aisé de mettre en œuvre des biais à partir du schéma. L'une des explications est que les schémas XML représentent des langages d'arbres et peuvent être facilement transformés en automates (cf. sect. 1.4).

Nous envisageons essentiellement deux façons d'intégrer des connaissances issues du schéma dans les algorithmes d'induction de requêtes par inférence d'automates. La première consiste à assurer que la requête apprise est consistante avec le schéma. Coste *et al.* (2004) ont formalisé cet apport de connaissances *a priori* dans le cas de l'inférence d'automates de mots ; ils distinguent d'une part un *biais de domaine*, où le schéma est utilisé comme une connaissance purement sémantique sur le langage de l'automate cible, et d'autre part un *biais de typage*, où la connaissance sémantique fournie par le schéma est embarquée dans la structure de l'automate inféré, donc traduite syntaxiquement. Nous généralisons les deux approches à l'inférence de requêtes représentées par des automates d'arbres annotés (les transducteurs de sélection de nœuds du chapitre précédent) et montrons que les requêtes consistantes avec un schéma sont apprenables dans une variante

du modèle classique d'identification à la limite de Gold (1967). À notre connaissance, ce résultat n'a pas été complètement démontré dans le cas des automates de mots.

La seconde utilisation du schéma que nous proposons ici s'inscrit dans le cadre des heuristiques d'élagage introduites par Carme *et al.* (2007) ; nous intégrons le schéma, représenté par un automate déterministe, à la définition de l'élagage. Ce dernier a pour but de réduire la taille des exemples donnés à l'algorithme d'induction et de permettre un apprentissage à partir d'exemples partiellement annotés. Cependant, élaguer des arbres a pour conséquence la perte de certaines informations, ce qui peut rendre impossible l'identification de requêtes pour lesquelles la sélection d'un nœud dépend d'une information éliminée par l'élagage. L'emploi du schéma dans l'élagage permet de limiter cette perte d'informations. Afin de caractériser précisément la classe de requêtes pouvant être identifiées par un algorithme d'induction utilisant un élagage, nous introduisons la notion de requêtes stables relativement à l'élagage. Nous donnons un résultat d'apprenabilité sur cette nouvelle classe des requêtes.

La suite du chapitre s'organise comme suit. Dans la seconde section, nous rappelons les bases de l'inférence de requêtes sur lesquelles s'appuient tous nos algorithmes d'apprentissage. La troisième section présente l'induction de requêtes consistantes avec le schéma. La quatrième section s'intéresse à l'élagage. Dans la dernière section, nous étudions la notion de stabilité.

4.2 Inférence de requêtes régulières

Nous définissons l'induction de requêtes comme un problème d'apprentissage supervisé (Mitchell, 1997). Dans ce cadre, induire une requête consiste à inférer une structure qui la représente (la cible), à partir d'un ensemble (fini) d'exemples qui constituent des "réponses" à la requête. Les requêtes monadiques étant ici représentées par des transducteurs de sélection de nœuds, notre problème d'induction de requêtes se ramène au problème plus général de l'inférence d'automates d'arbres ; les exemples sont ici des arbres annotés par la requête cible (cf. chap. 3).

L'inférence d'automates d'arbres comme technique d'apprentissage pour l'induction supervisée de requêtes a été employée notamment par Kosala (2003) et Raeymaekers (2008). Dans ces travaux, les requêtes sont représentées par des langages d'arbres locaux afin de permettre un apprentissage par exemples positifs seuls. Carme *et al.* (2007) ont

présenté un algorithme d'inférence permettant d'identifier la classe plus expressive des langages réguliers d'arbres complètement annotés, autrement dit les requêtes régulières. Notre approche se fonde sur cette technique d'apprentissage, que nous présentons dans la suite.

4.2.1 Inférence d'automates d'arbres

L'inférence grammaticale (de la Higuera, 2010), appelée aussi induction d'automates ou induction de grammaires, est une technique d'apprentissage supervisée qui consiste *grosso modo* à apprendre la représentation d'un langage (une grammaire ou un automate) à partir d'un ensemble fini d'éléments, appelés exemples, appartenant ou non à ce langage. Une grande variété de problèmes peuvent être modélisés comme des problèmes d'induction de langage, par exemple en reconnaissance de formes, en traitement de la langue naturelle ou encore en bio-informatique, et de nombreux algorithmes d'inférence grammaticale ont été élaborés pour les résoudre (voir les études exhaustives de Pitt (1989), Sakakibara (1997) et de la Higuera (2005)). Parmi ceux-ci, nous nous intéressons plus particulièrement à l'inférence d'automates d'arbres avec l'algorithme $\mathcal{RPNI}$ (pour *Regular Positive Negative Inference*) développé par Oncina et García, d'abord pour les langages réguliers de mots (Oncina et García, 1992), puis étendu aux ensembles réguliers d'arbres (García et Oncina, 1993).

Nous donnons plus bas une définition du modèle d'apprentissage en temps et données polynomiaux (Gold, 1978; de la Higuera, 1997), qui est une variante du modèle classique d'identification à la limite de Gold (1967), adaptée au problème de l'inférence d'automates d'arbres. Informellement, l'idée du modèle d'identification à la limite est la suivante. Un processus d'apprentissage reçoit continûment des informations sur un langage inconnu, sous forme d'exemples positifs et éventuellement négatifs, et produit en sortie la représentation d'un langage ; un langage est identifiable à la limite si, pour toute présentation admissible d'exemples du langage, le processus converge vers une représentation de ce langage au bout d'un temps fini. Gold (1967) a montré que la classe des langages réguliers (de mots) est apprenable dans ce modèle à partir d'exemples positifs et négatifs, mais pas à partir d'exemples positifs seuls. Le modèle en temps et données polynomiaux est un raffinement du précédent modèle qui introduit des contraintes de polynomialité sur le temps de réponse de l'algorithme d'apprentissage et la taille de l'échantillon caractéristique. Gold (1978) a montré que, sous ces conditions, les langages réguliers représentés par automates finis déterministes sont apprenables par exemples positifs et négatifs dans

ce modèle ; en revanche, ce résultat n'est plus vrai si les langages sont représentés par des automates non déterministes (de la Higuera, 1997). Dans le cas des langages réguliers d'arbres représentés par automates déterministes, nous verrons que la contrainte de polynomialité porte sur la cardinalité de l'échantillon caractéristique et non plus sur sa taille.

Nous nous intéressons maintenant aux aspects formels de l'apprentissage de langages réguliers d'arbres. Un *exemple* consiste en un couple (t, b) où t est un arbre et b une étiquette booléenne ; un *exemple positif* est étiqueté par 1 (ou `vrai`), un *exemple négatif* par 0 (ou `faux`). Un *échantillon* S est un ensemble fini d'exemples. La *taille* de l'échantillon, notée $|S|$, est le nombre total de nœuds dans les arbres qui le composent ; sa *cardinalité* (nombre d'éléments qu'il contient) est dénotée $\#S$. Un automate d'arbres A est *compatible* avec un exemple (t, b) si $t \in \mathcal{L}^u(A) \Leftrightarrow b$. Par extension, A est compatible avec un échantillon s'il est compatible avec tous ses exemples ; dans l'autre sens, on dit qu'un échantillon est *consistant* avec A.

Définition 4.1
Les langages d'arbres représentés par une classe d'automates $\mathscr{R}$ sont *identifiables en temps et données polynomiaux* [1] s'il existe deux polynômes ρ_1, ρ_2 et un algorithme $\mathcal{Learn}$ tels que :

1. étant donné un échantillon S, $\mathcal{Learn}$ retourne un automate $A \in \mathscr{R}$ compatible avec S en temps $\mathcal{O}(\rho_1(|S|))$;
2. pour tout automate $A \in \mathscr{R}$, il existe un échantillon caractéristique CS de cardinalité (∗) inférieure à $\rho_2(|A|)$ pour lequel étant donné un échantillon S contenant CS consistant avec A, $\mathcal{Learn}$ retourne un automate $A' \in \mathscr{R}$ équivalent à A. ◀

(∗) Le modèle d'apprentissage pour les langages d'arbres diffère du modèle pour les langages de mots sur un point important : il est demandé que la *cardinalité* de l'échantillon caractéristique soit polynomiale dans la taille de l'automate cible, tandis que pour les langages de mots il s'agit de sa *taille* ; le modèle pour les arbres est donc plus faible. Cela est dû au fait que la taille des arbres, mesurée en nombre de nœuds, peut être exponentielle dans la taille de la cible. En effet, si l'on considère par exemple le langage d'arbres contenant un unique arbre binaire équilibré de hauteur n sur un alphabet $\Sigma_@$ avec $\Sigma = \{a\}$, l'échantillon caractéristique doit contenir cet arbre ayant $2^{n+1} - 1$ nœuds alors que l'automate *stepwise* correspondant a seulement $n + 1$ états. Cette explosion est due à la nature intrinsèque des arbres. Ce problème pourrait être contourné en considérant une

1. Nous utiliserons également le terme *apprenable* pour signifier *identifiable en temps et données polynomiaux*.

représentation compacte de l'échantillon (par exemple avec des arbres "compressés"), ce que nous avons choisi de ne pas aborder afin de simplifier la présentation.

Théorème 4.2
Les langages réguliers d'arbres représentés par des automates déterministes ascendants sont identifiables en temps et données polynomiaux à partir d'exemples positifs et négatifs.

Nous ne donnons pas la preuve de ce résultat ici, mais discutons de l'algorithme d'apprentissage sur lequel il s'appuie dans la section suivante.

4.2.2 Algorithme $\mathcal{RPNI}$ pour les langages réguliers d'arbres

García et Oncina (1993) ont montré que la classe des langages réguliers d'arbres représentés par des automates d'arbres déterministes (ascendants) sont identifiables en temps et données polynomiaux avec l'algorithme $\mathcal{RPNI}$ (th. 4.2). Étant donné un échantillon $S = S^+ \uplus S^-$, où S^+ est un ensemble d'exemples positifs et S^- un ensemble d'exemples négatifs, l'algorithme $\mathcal{RPNI}$ retourne un automate d'arbres déterministe A compatible avec S en temps polynomial ; si S contient un échantillon caractéristique, l'automate retourné est l'automate déterministe minimal (au sens de Myhill-Nerode) qui reconnaît le langage cible. Cet algorithme correspond à une généralisation de l'algorithme d'inférence de langages réguliers de mots élaboré par les mêmes Oncina et García (1992) ; sa complétude repose sur le fait que l'échantillon caractéristique définit exactement l'automate déterministe minimal du langage cible.

Le principe général du fonctionnement de l'algorithme $\mathcal{RPNI}$, présenté figure 48 et détaillé ci-après, est le suivant. Il s'agit dans un premier temps de construire un automate d'arbres déterministe qui reconnaît exactement l'ensemble des exemples positifs de l'échantillon, puis de généraliser le langage de cet automate par fusions successives de ses états, tout en assurant qu'il reste compatible avec l'ensemble des exemples négatifs.

Plus précisément, la fonction $\mathtt{init}(S^+)$ (ligne 1) construit un automate déterministe A, appelé *automate initial*, en créant un état par sous-arbre de l'échantillon positif et en ajoutant les règles associées. Cet automate correspond au plus grand automate d'arbres déterministe (en nombre d'états) reconnaissant le langage S^+ ; il est productif par construction. Sans perte de généralité, on peut considérer que les arbres de S^+ sont curryfiés et que A est un automate *stepwise*. Soient $t_1, \ldots, t_n$ les sous-arbres des arbres de l'échantillon S^+.

Entrée: $S = S^+ \uplus S^-$, un échantillon composé d'exemples positifs (S^+) et négatifs (S^-)
Sortie: A, un automate d'arbres
1: $A \leftarrow \mathtt{init}(S^+)$
2: **let** $\mathtt{sta}(A) = \{p_1, \ldots, p_n\}$ // *on suppose les états de* A *convenablement ordonnés*
3: **for** $i = 2$ **to** n **do**
4:    **for** $j = 1$ **to** $i - 1$ **do**
5:       $A' \leftarrow \mathtt{det\text{-}merge}(A, p_i, p_j)$
6:       **if** A' est compatible avec S^- **then**
7:          $A \leftarrow A'$
8:          **exit inner loop**
9: **return** A

Figure 48. *Algorithme $\mathcal{RPNI}$.*

On pose $\mathtt{sta}(A) = \{p_1, \ldots, p_n\}$ de sorte que tout sous-arbre t_i est associé un unique état p_i. Les états finaux de A et ses transitions sont définis comme suit :

$$\begin{aligned} \mathtt{fin}(A) &= \{p_i \mid t_i \in S^+\} \\ \mathtt{rul}(A) &= \{f \to p_i \mid t_i = f\} \cup \{p_i @ p_j \to p_k \mid t_k = @(t_i, t_j)\} \end{aligned}$$

La fusion de deux états p_i et p_j consiste à remplacer p_j par p_i dans toutes les règles de A où p_j apparaît. Cette opération a pour but de généraliser le langage de l'hypothèse. La fonction $\mathtt{det\text{-}merge}(A, p_i, p_j)$ (ligne 5) réalise cette fusion en garantissant que la sortie est déterministe, autrement dit le non déterminisme possiblement induit par la fusion de p_i et p_j est éliminé à la volée par la fusion d'autres états en cascade. Par exemple, fusionner les états p_2 et p_3 dans un automate ayant deux règles $p_1 @ p_2 \to p_4$ et $p_1 @ p_3 \to p_5$ implique la fusion des états p_4 et p_5.

Les fusions déterministes sont tentées dans un ordre fixé au départ qui doit respecter l'ordre de hauteur des états dans l'automate initial. Pour cela, on associe à chaque état une hauteur correspondant à la hauteur de l'unique arbre qu'il reconnaît dans l'automate initial. La fusion de deux états p_i et p_j de hauteur h_i et h_j doit être précédée de l'ensemble des fusions possibles entre états de hauteur strictement inférieure à $\max\{h_i, h_j\}$. Cette contrainte sur l'ordre de fusion des états est liée à la construction de l'échantillon caractéristique et est nécessaire pour obtenir l'identification.

Pour finir, le test de la ligne 6 a pour objet d'assurer la compatibilité de l'automate obtenu après fusion déterministe avec l'échantillon ; si tel est le cas, l'automate A' devient la nouvelle hypothèse et la boucle interne est interrompue. Par construction, A' est

nécessairement compatible avec S^+. Vérifier la consistance de S^- avec A' revient à s'assurer qu'aucun arbre de S^- n'est reconnu par A'.

Nous examinons le temps de calcul de l'algorithme $\mathcal{RPNI}$ dans le pire des cas. La construction de l'automate initial s'effectue clairement en temps $\mathcal{O}(|S^+|)$, donc sa taille est également en $\mathcal{O}(|S^+|)$. L'opération de fusion déterministe s'effectue en temps linéaire dans la taille de l'automate, donc en temps $\mathcal{O}(|S^+|)$. Vérifier la compatibilité de l'échantillon négatif a la même complexité que vérifier l'appartenance de chacun de ses éléments au langage de l'automate, à savoir $\mathcal{O}(|S^-| + |S^+|)$ car l'automate est déterministe (cf. tab. 1 p. 33). Ces deux dernières opérations sont effectuées pour toutes les paires d'états de l'automate, donc on observe finalement une complexité de $\mathcal{O}\big((|S^-| + |S^+|) \times |S^+|^2\big)$.

Pour terminer, nous discutons informellement des conditions que doit vérifier un échantillon pour être caractéristique, relativement à un langage cible représenté par un automate d'arbres déterministe. Pour tout état de cet automate, un échantillon caractéristique doit contenir au moins le plus petit sous-arbre évalué dans cet état. Par ailleurs, toute transition de l'automate doit être représentée dans les exemples positifs de l'échantillon. Pour cela, on combine les sous-arbres obtenus précédemment pour construire des arbres du langage cible. Pour terminer, les exemples négatifs de l'échantillon doivent permettre de distinguer toutes les paires d'états de l'automate cible. L'ordre de fusion défini plus haut garantit l'identification de la cible par l'algorithme $\mathcal{RPNI}$, dans le cas où l'on dispose d'un échantillon caractéristique, avec comme invariant que pour toute fusion d'états déterministe tentée, il n'existe pas de plus petit état équivalent, autrement dit les états de l'automate déterministe minimal sont traités dans l'ordre croissant.

4.2.3 Algorithme $\mathrm{t}\mathcal{RPNI}$ pour les requêtes monadiques

Le résultat d'apprenabilité des langages réguliers d'arbres s'étend aux requêtes régulières représentées par des transducteurs de sélection de nœuds. Une légère adaptation du modèle et de l'algorithme d'apprentissage est toutefois nécessaire. En effet, les requêtes sont définies par des langages réguliers d'arbres sur un alphabet $\Sigma \times \mathbb{B}$ — plus exactement des langages fonctionnels et non-nuls (cf. sect. 3.4). Or, dans le cadre de l'induction de requêtes monadiques représentées par des transducteurs de sélection de nœuds, l'échantillon est constitué d'exemples positifs seuls, soit des arbres sur $\Sigma \times \mathbb{B}$ que nous appellerons *exemples complètement annotés* ; cependant, les exemples négatifs sont implicites : ils peuvent facilement être déduits à partir des exemples positifs, avec la propriété que pour tout exemple

complètement annoté $t * \beta$, tout autre arbre annoté $t * \beta'$ avec $\beta' \neq \beta$ est nécessairement un exemple négatif. De ce fait, la compatibilité de l'échantillon avec l'automate hypothèse peut être assurée par les propriétés de fonctionnalité et de non-nullité.

Définition 4.3
Les langages de requêtes représentés par une classe d'automates $\mathscr{R}$ sont *identifiables en temps et données polynomiaux* s'il existe deux polynômes ρ_1, ρ_2 et un algorithme *Learn* tels que :

1. étant donné un échantillon S, *Learn* retourne un automate $A \in \mathscr{R}$ en temps $\mathcal{O}(\rho_1(|S|))$;
2. pour tout automate $A \in \mathscr{R}$ représentant une requête $\mathcal{Q}_A$, il existe un échantillon caractéristique CS de cardinalité inférieure à $\rho_2(|A|)$ pour lequel étant donné un échantillon S contenant CS consistant avec A, *Learn* retourne un automate $A' \in \mathscr{R}$ représentant une requête $\mathcal{Q}_{A'}$ équivalente à $\mathcal{Q}_A$, c'est-à-dire $\mathcal{L}(\mathcal{Q}_A) = \mathcal{L}(\mathcal{Q}_{A'})$. ◄

Dans ce modèle, l'algorithme d'apprentissage retourne un automate déterministe représentant la requête cible seulement si l'échantillon d'entrée est caractéristique ; dans le cas contraire, il retourne un automate qui ne représente pas nécessairement une requête appartenant à la classe de la cible, et dont la compatibilité avec l'échantillon d'entrée n'est pas garantie. Un modèle d'apprentissage plus fort pourrait l'exiger ; les résultats d'apprenabilité de la présente section et de la suivante respectent d'ailleurs ces deux contraintes par nature. Nous relâchons la condition de correction en vue de l'apprentissage avec élagage présenté aux sections 4.4 et 4.5.

Théorème 4.4 (Carme *et al.* (2007))
La classe des requêtes monadiques représentées par des transducteurs de sélection de nœuds déterministes est identifiable en temps et données polynomiaux à partir d'exemples complètement annotés.

Le résultat est obtenu par réduction sur l'identification des langages réguliers représentés par des automates d'arbres déterministes. Il est généralisé à la section suivante par l'identification des requêtes consistantes avec un schéma. Ainsi la preuve du théorème 4.8 p. 123 permet de le démontrer à nouveau. Dans la suite, nous présentons l'algorithme t*RPNI*, une variante de *RPNI* pour l'inférence de requêtes monadiques représentées par des transducteurs de nœuds déterministes.

Entrée: S, un échantillon d'arbres complètement annotés
Sortie: A, un transducteur de sélection de nœuds
1: $\widehat{S} \leftarrow \{t * \beta \in S \mid \forall v \in \mathtt{nod}(\beta)\ (\beta(v) = 0)\}$
2: $A \leftarrow \mathtt{init}(S \setminus \widehat{S})$
3: **let** $\mathtt{sta}(A) = \{p_1, \dots, p_n\}$
4: **for** $i = 2$ **to** n **do**
5: **for** $j = 1$ **to** $i - 1$ **do**
6: $A' \leftarrow \mathtt{det\text{-}merge}(A, p_i, p_j)$
7: **if** $\mathcal{L}^u(A')$ est fonctionnel et non-nul
and A' est compatible avec $\widehat{S}$ **then**
8: $A \leftarrow A'$
9: **exit inner loop**
10: **return** A

Figure 49. *Algorithme* $\mathtt{t}\mathcal{RPNI}$.

Étant donné un échantillon S d'arbres complètement annotés, l'algorithme $\mathtt{t}\mathcal{RPNI}$ de la figure 49 retourne un TSN déterministe représentant une requête monadique en temps $\mathcal{O}(|S|^5)$. La complexité de $\mathtt{t}\mathcal{RPNI}$ repose sur le test de fonctionnalité (ligne 7) qui, après chaque fusion déterministe, garantit la consistance de l'échantillon avec l'automate hypothèse en temps $\mathcal{O}(|S|^3)$. Notons qu'une partie de l'échantillon, dénotée ici $\widehat{S}$, peut comporter des arbres dont toutes les annotations sont négatives. Ces arbres ne sont pas pris en compte pour la construction de l'automate initial (lignes 1 et 2). Le test de compatibilité de la ligne 7 garantit, en temps $\mathcal{O}(|S| \times |\widehat{S}|)$, que pour tout arbre $t * \beta$ de $\widehat{S}$ il n'existe pas $t * \beta' \in \mathcal{L}^u(A')$ tel que $\beta \neq \beta'$; cet artifice est nécessaire pour assurer la complétude de l'algorithme.

La figure 50 présente un exemple d'exécution de l'algorithme $\mathtt{t}\mathcal{RPNI}$ avec pour entrée un échantillon composé d'un unique arbre complètement annoté (a). À partir de celui-ci (ou plus exactement de son codage curryfié) est produit l'automate initial (b). L'algorithme fusionne successivement les états 1 et 2 (c), 1 et 3 (d), puis 1 et 4 (e) qui entraîne la fusion de 1 et 5 pour conserver un automate déterministe, et enfin 1 et 6 (f). L'algorithme tente ensuite de fusionner l'état 7 avec l'état 1, mais cette fusion échoue car elle produit un automate non fonctionnel. La fusion suivante entre 7 et 8 (g) est acceptée. Une dernière fusion, entre 1 et 9 (h), est effectuée, entraînant par déterminisme les fusions 10-13 et 11-14. Aucune autre fusion n'est possible par la suite en raison de la condition de fonctionnalité. La sortie de l'algorithme est donc le dernier automate calculé (h).

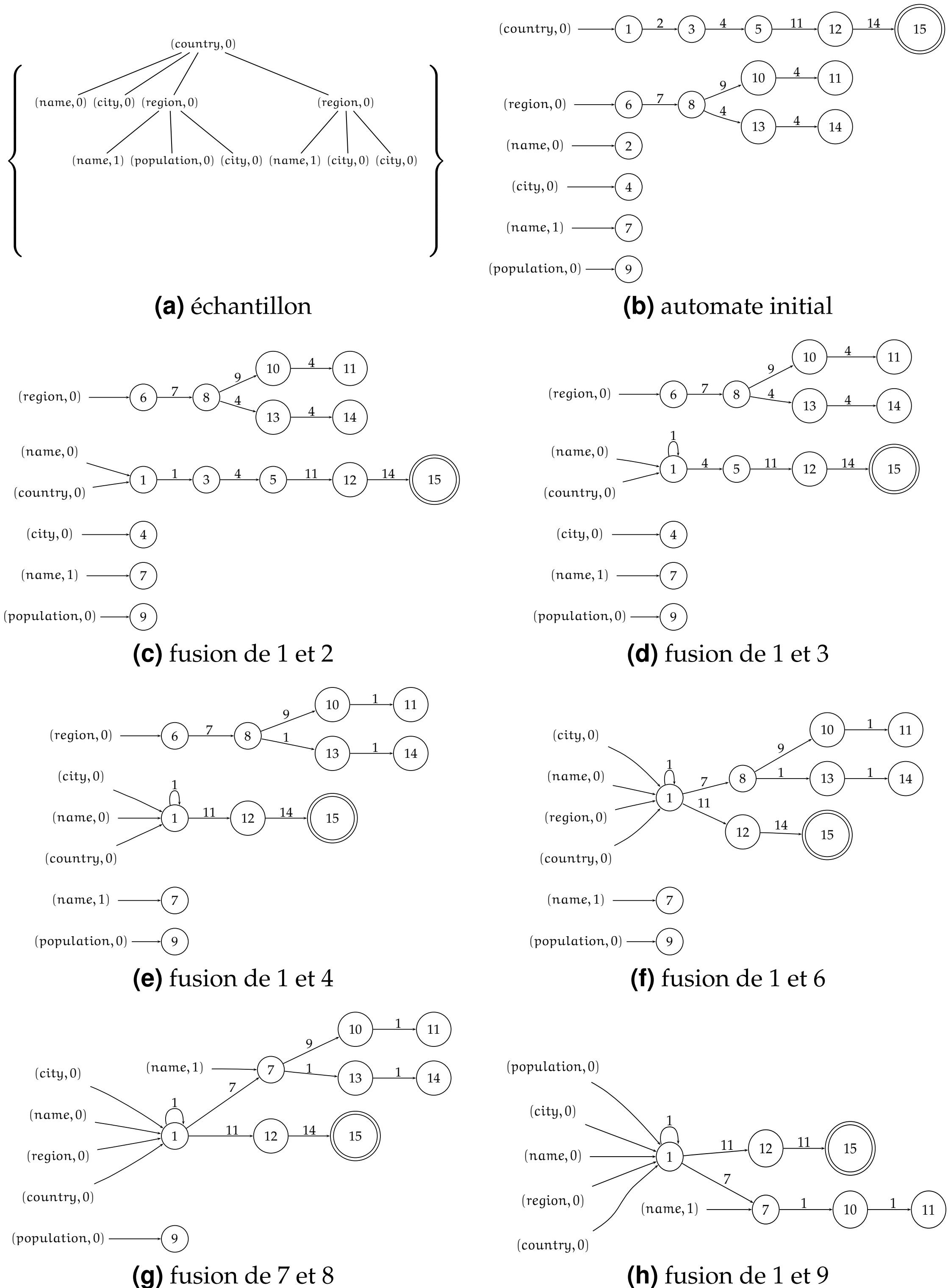


Figure 50. *Exemple d'exécution de l'algorithme* t$\mathcal{RPNI}$.

La suite du chapitre est consacrée à l'induction de requêtes guidée par schéma, d'abord avec la consistance, puis avec l'élagage.

4.3 Induction de requêtes consistantes avec un schéma

Une utilisation naturelle du schéma dans un algorithme d'inférence grammaticale consiste à assurer que le langage de l'automate appris soit inclus dans le langage défini par le schéma. L'idée sous-jacente est d'éviter que l'algorithme d'apprentissage réalise des généralisations erronées. Par rapport à d'autres heuristiques plus classiques basées sur des connaissances locales — p. ex. les k-testables (García, 1993; Raeymaekers *et al.*, 2008) —, le schéma peut apporter une connaissance globale sur le langage cible, qui dépendra du type de schéma considéré ; par exemple, une DTD déterministe apporte essentiellement de l'information sur les langages horizontaux. La consistance s'apparente donc à un critère sémantique plutôt que syntaxique. De plus, assurer la consistance avec le schéma n'affecte pas les résultats d'apprenabilité.

Coste *et al.* (2004) proposent un cadre générique pour intégrer des connaissances sur le domaine (dans le sens de langage), ou *biais de domaine*, dans les algorithmes d'inférence d'automates de mots par fusion d'états ; celui-ci s'étend naturellement au cas des arbres. Différentes interprétations sont possibles pour l'utilisation du schéma comme biais de domaine. Celui-ci peut être vu comme une hypothèse générale sur le domaine, obtenue par l'intermédiaire d'un expert ou *via* une première approximation. Par exemple, dans le cadre de l'inférence de DTD, on peut utiliser une DTD connue plus large si elle est disponible (Ahonen, 1995, 1996; Fernau, 2001) ; c'est le cas lorsque l'on souhaite apprendre une DTD plus spécifique que celle de HTML. Inversement, le schéma donne implicitement une information négative sur le langage cible : le complémentaire de son langage peut fournir un ensemble infini de contre-exemples en complément de l'échantillon négatif (Lambeau *et al.*, 2008). Ces diverses approches ne sont pas exclusives ; on peut même imaginer que l'on dispose de plusieurs schémas afin de les combiner.

Une autre méthode, basée sur un typage des états et qualifiée de *biais de typage* par les mêmes Coste *et al.* (2004), est envisageable pour assurer la consistance de l'automate cible avec un schéma. Elle a été employée, par exemple, par Oncina et Varó (1996) pour l'apprentissage de transducteurs subséquentiels. L'idée est d'utiliser les états de l'automate déterministe représentant le schéma afin de typer les états de l'automate hypothèse, puis d'interdire toute fusion élémentaire de deux états de types différents.

Dans la suite de la section, nous envisageons l'induction de requêtes consistantes avec un schéma comme un nouveau problème d'apprentissage. Nous précisons d'abord la notion de requête consistante avec un schéma. Puis nous présentons un algorithme d'induction de requêtes qui vérifie la consistance par un test d'inclusion d'automates, et démontrons l'apprenabilité des requêtes consistantes avec cet algorithme. Enfin, nous présentons un algorithme alternatif qui assure la consistance de façon statique par un typage des états de l'automate induit.

4.3.1 Requêtes consistantes avec un schéma

Dans le cadre de l'induction de requêtes où l'on se place, un schéma D est représenté par un automate déterministe sur un alphabet Σ, tandis que l'automate inféré A est défini sur $\Sigma \times \mathbb{B}$. En effet, l'algorithme d'apprentissage a en entrée un ensemble d'arbres (complètement) annotés, donc sur $\Sigma \times \mathbb{B}$, qui proviennent d'une collection de documents semi-structurés définis, eux, sur Σ et pour lesquels on dispose d'une DTD ou d'un autre type de schéma. De ce fait, la vérification de la consistance est effectuée uniquement sur la composante Σ du langage de l'automate A, autrement dit l'inclusion $\mathcal{L}^u(\Pi_\Sigma((A)) \subseteq \mathcal{L}(D)$ doit être un invariant d'un algorithme d'apprentissage qui intègre la consistance avec le schéma. Nous formalisons cette discussion ci-dessous.

Définition 4.5
Une requête (monadique) $\mathcal{Q}$ est dite *consistante avec un schéma* D (ou D-*consistante*) sur Σ si elle sélectionne des nœuds uniquement dans des arbres satisfaisant le schéma, autrement dit si $\mathcal{Q}(t) = \emptyset$ pour tout $t \notin \mathcal{L}(D)$. ◀

Une reformulation directe de la définition de consistance conduit à l'énoncé suivant : $\mathcal{Q}$ est consistante avec D si $\mathrm{dom}(\mathcal{Q}) \subseteq \mathcal{L}(D)$. Partant, on obtient la proposition qui suit.

Lemme 4.6
Soit une requête $\mathcal{Q}$ représentée par un transducteur de sélection de nœuds A sur Σ, la requête $\mathcal{Q}$ est consistante avec D si $\mathcal{L}^u(\Pi_\Sigma(A)) \subseteq \mathcal{L}(D)$.

Preuve
On montre que $\mathcal{L}^u(\Pi_\Sigma(A)) \subseteq \mathcal{L}(D)$ équivaut à $\mathcal{Q}(t) = \emptyset$ pour tout $t \notin \mathcal{L}(D)$. Par définition, on a $t \notin \mathcal{L}(D) \Rightarrow \mathcal{Q}_A(t) = \emptyset$, ce qui équivaut à $\mathcal{Q}_A(t) \neq \emptyset \Rightarrow t \in \mathcal{L}(D)$, puis, par définition de $\mathcal{Q}_A$, à $\mathrm{dom}(\mathcal{Q}_A) \subseteq \mathcal{L}(D)$. Or, $\mathrm{dom}(\mathcal{Q}_A) = \{t \in T^u_\Sigma \mid \exists \beta \in T^u_\mathbb{B} \text{ tq. } t * \beta \in \mathcal{L}^u(A)\}$, d'où $\mathrm{dom}(\mathcal{Q}_A) = \Pi_\Sigma(\mathcal{L}^u(A)) = \mathcal{L}^u(\Pi_\Sigma(A))$ (cf. sect. 3.2). □

La vérification de la consistance d'une requête $\mathcal{Q}$ avec un schéma D peut donc se faire en testant l'inclusion du domaine de $\mathcal{Q}$, qui correspond au langage de la projection sur la composante Σ de l'automate qui la représente, dans le langage de D. Par ailleurs, vérifier si une requête monadique est consistante avec un schéma représenté par un automate déterministe peut être réalisé efficacement grâce aux algorithmes du chapitre 2.

Proposition 4.7

Soient $\mathcal{Q}$ une requête monadique représentée par un transducteur de sélection de nœuds A sur Σ et D un schéma représenté par un automate stepwise (factorisé) déterministe sur Σ. On peut décider en temps $\mathcal{O}(|A| \times |\Sigma| \times |D|)$ si $\mathcal{Q}$ est une requête D-consistante.

Preuve

Par le lemme 4.6 et les théorèmes 2.23 et 1.11. □

Nous étudions dans la suite comment vérifier la consistance avec un schéma au cours de l'apprentissage, d'abord par un test d'inclusion, puis par un typage des états.

4.3.2 Vérification de la consistance par un test d'inclusion

Dans cette section, nous présentons l'algorithme $t\mathcal{RPNI}_D$, basé sur un test d'inclusion d'automates d'arbres, pour l'induction de requêtes D-consistantes, ainsi qu'une optimisation de cet algorithme qui tire parti de l'incrémentalité de l'inclusion.

Algorithme $t\mathcal{RPNI}_D$ pour les requêtes D-consistantes

L'apprentissage de requêtes monadiques consistantes avec un schéma constitue une extension de l'apprentissage de requêtes monadiques présenté précédemment. On peut donc se replacer dans le modèle de la définition 4.3 et montrer le théorème suivant.

Théorème 4.8

Soit D un schéma représenté par un automate (factorisé) déterministe. La classe des requêtes monadiques consistantes avec D représentées par des transducteurs de nœuds déterministes est identifiable en temps et données polynomiaux à partir d'exemples complètement annotés.

Notons que si $\mathcal{L}(D) = T_\Sigma^u$, autrement dit si le schéma représente le langage universel, ce théorème équivaut au théorème 4.4 sur l'apprenabilité des requêtes monadiques. La preuve qui suit, basée sur une réduction à l'apprentissage de langages réguliers d'arbres avec l'algorithme $\mathcal{RPNI}$, permet ainsi de démontrer les deux théorèmes à la fois.

Entrée: S, un échantillon d'arbres complètement annotés consistant avec le schéma D
Sortie: A, un transducteur de sélection de nœuds consistant avec D
1: $\widehat{S} \leftarrow \{t * \beta \in S \mid \forall \nu \in \mathtt{nod}(\beta)\ (\beta(\nu) = 0)\}$
2: $A \leftarrow \mathtt{init}(S \setminus \widehat{S})$
3: **let** $\mathtt{sta}(A) = \{p_1, \dots, p_n\}$
4: **for** $i = 2$ **to** n **do**
5: **for** $j = 1$ **to** $i - 1$ **do**
6: $A' \leftarrow \mathtt{det\text{-}merge}(A, p_i, p_j)$
7: **if** $\mathcal{L}^u(A')$ est fonctionnel et non-nul
and A' est compatible avec $\widehat{S}$
and A' est consistant avec D **then**
8: $A \leftarrow A'$
9: **exit inner loop**
10: **return** A

Figure 51. *Algorithme* $t\mathcal{RPNI}_D$.

Preuve

Considérons l'algorithme $t\mathcal{RPNI}_D$ de la figure 51. Il s'agit d'une variante de l'algorithme $t\mathcal{RPNI}$ qui intègre l'invariant discuté plus haut. On montre sa correction et sa complétude, à savoir : (1) $t\mathcal{RPNI}_D$ retourne un TSN déterministe qui représente une requête consistante avec D en temps polynomial dans la taille de l'échantillon (on considère ici une déclinaison plus exigeante du modèle d'apprentissage, cf. discussion p. 118) ; (2) pour tout TSN déterministe A représentant une requête D-consistante, il existe un échantillon caractéristique de cardinalité polynomiale dans la taille de A pour lequel $t\mathcal{RPNI}_D$ retourne un TSN déterministe équivalent.

1. Comme l'échantillon d'arbres complètement annotés S est consistant avec D (sur la composante Σ), le langage de l'automate initial l'est aussi nécessairement ($\mathcal{L}^u(\Pi_\Sigma(\mathtt{init}(S))) \subseteq \mathcal{L}(D)$) ; pour qu'une fusion déterministe soit acceptée, l'automate qui en est issu doit être consistant avec D ($\mathcal{L}^u(\Pi_\Sigma(\mathtt{det\text{-}merge}(A, q_i, q_j))) \subseteq \mathcal{L}(D)$), et son langage fonctionnel et non-nul. Si D est représenté par un automate (factorisé) déterministe sur Σ, l'inclusion $\mathcal{L}^u(\Pi_\Sigma(A)) \subseteq \mathcal{L}(D)$ peut être vérifiée efficacement en temps $\mathcal{O}(|A| \times |\Sigma| \times |D|)$ grâce aux algorithmes présentés dans le chapitre 2 [2]. Par conséquent, $t\mathcal{RPNI}_D$ retourne un TSN déterministe consistant avec D et compatible avec S en temps polynomial

2. L'automate $\Pi_\Sigma(A)$ n'est pas nécessairement déterministe mais cela n'affecte pas la complexité du calcul comme cela a été démontré. D'autre part, l'automate A et sa projection $\Pi_\Sigma(A)$ sont productifs par construction.

$\mathcal{O}(\rho_1(|S|)) = \mathcal{O}(|S|^5 + |S|^3 \times |\Sigma| \times |D|)$, qui correspond au temps du test de fonctionnalité auquel est additionné celui du test d'inclusion.

2. Soit A un TSN déterministe qui représente une requête D-consistante $\mathcal{Q}_A$. Par définition, A est un automate *stepwise* déterministe sur $\Sigma \times \mathbb{B}$, donc il existe un polynôme ρ_2 et un échantillon caractéristique $CS' = CS^+ \uplus CS^-$ de cardinalité inférieure à $\rho_2(|A|)$ pour lequel, étant donné un échantillon qui contient CS', l'algorithme $\mathcal{RPNI}$ retourne un automate *stepwise* déterministe (minimal) A' équivalent à A (cf. th. 4.2 et sect. 4.2.2). On définit un échantillon d'exemples complètement annotés CS à partir de CS' en conservant tous les exemples positifs, et en substituant un positif à chaque exemple négatif consistant avec D dont la composante Σ appartient au domaine de $\mathcal{Q}_A$:

$$CS = CS^+ \cup \{t * \mathcal{Q}_A(t) \in T^u_{\Sigma \times \mathbb{B}} \mid t * \beta \in CS^- \wedge \mathcal{Q}_A(t) \neq \emptyset \wedge t \in \mathcal{L}(D)\}$$

On définit également, à partir de l'échantillon d'exemples négatifs CS^-, un échantillon $\widehat{CS}$ d'exemples dont les annotations sont toutes négatives ; ces exemples permettent de capturer les négatifs D-consistants de $\mathcal{RPNI}$ qui ont au moins une annotation positive, mais pour lesquels la requête cible n'extrait en réalité aucun nœud :

$$\widehat{CS} = \{t * \mathcal{Q}_A(t) \in T^u_{\Sigma \times \mathbb{B}} \mid t * \beta \in CS^- \wedge \mathcal{Q}_A(t) = \emptyset \wedge t \in \mathcal{L}(D)\}$$

Notons que par définition $CS \subseteq \mathcal{L}^u(A)$. Par ailleurs, la cardinalité de $CS \uplus \widehat{CS}$ est par construction inférieure ou égale à celle de CS', et donc à $\rho_2(|A|)$. On montre que $CS \uplus \widehat{CS}$ est un échantillon caractéristique pour $t\mathcal{RPNI}_D$. Prenons un échantillon S d'arbres complètement annotés consistants avec D tel que $S \supseteq CS \uplus \widehat{CS}$. On doit vérifier que l'algorithme $t\mathcal{RPNI}_D$ avec l'échantillon S se comporte de façon identique à l'algorithme $\mathcal{RPNI}$ avec un échantillon $S' = S^+ \uplus S^-$, où $S^+ = S \setminus \widehat{CS}$ correspond à l'ensemble des exemples positifs (les exemples de $\widehat{CS}$ ont par définition une contrepartie explicite dans S^-) et $S^- = CS^- \cup \{t * \beta \in T^u_{\Sigma \times \mathbb{B}} \mid \forall v \in nod(t)\ (\beta(v) = 0)\}$ aux exemples négatifs (on complète CS^- avec l'ensemble des arbres annotés dont toutes les annotations sont négatives). Par définition de S, on a $CS^+ \subseteq S^+$, donc $S' \supseteq CS'$ est un échantillon caractéristique pour $\mathcal{RPNI}$; par conséquent, l'algorithme $\mathcal{RPNI}$ retourne l'automate *stepwise* déterministe A' défini plus haut. L'automate initial produit par $\mathcal{RPNI}$ est syntaxiquement équivalent à l'automate initial produit par $t\mathcal{RPNI}_D$; les deux algorithmes prennent en effet en compte les exemples

positifs, c'est-à-dire l'ensemble $S \backslash \widehat{CS}$, et seulement eux pour le construire. On montre ensuite que $t\mathcal{RPNI}_D$ simule $\mathcal{RPNI}$. Une fusion déterministe est rejetée par $\mathcal{RPNI}$ si et seulement si elle produit un automate reconnaissant un arbre annoté $t * \beta \in T^u_{\Sigma \times \mathbb{B}}$ appartenant à l'échantillon négatif S^- ; une telle fusion est nécessairement refusée par $t\mathcal{RPNI}_D$:

a) soit parce qu'elle produit un automate dont le langage n'est pas fonctionnel (il reconnaît un arbre $t * \beta$ qui est par définition en conflit avec $t * \mathcal{Q}_A(t) \in CS$) ;
b) soit parce qu'elle produit un automate dont le langage ne respecte pas la propriété de non-nullité (il reconnaît un arbre $t * \beta$ avec $\beta(\nu) = 0$ pour tout nœud ν, ce qui revient à $\mathcal{Q}_A(t) = \emptyset$) ;
c) soit parce qu'elle produit un automate qui n'est pas compatible avec $\widehat{CS}$ (il reconnaît un arbre $t * \beta'$ alors qu'il existe $t * \beta \in \widehat{CS}$ et $\beta \neq \beta'$) ;
d) soit parce qu'elle produit un automate qui n'est pas consistant avec D (il reconnaît un arbre $t * \beta$ or $t \notin \mathcal{L}(D)$).

Par ailleurs, toute fusion déterministe acceptée par $\mathcal{RPNI}$ est nécessairement acceptée par $t\mathcal{RPNI}_D$. En effet, une telle fusion est "bonne" par nature (c.-à-d. correspond à la fusion de deux états équivalents au sens de Myhill-Nerode) puisque l'algorithme bénéficie de l'échantillon caractéristique. Inversement, toute fusion déterministe testée par $t\mathcal{RPNI}_D$ est nécessairement testée par $\mathcal{RPNI}$: leurs automates initiaux sont syntaxiquement égaux et les fusions sont inductivement synchronisées selon l'argument précédent. Par conséquent, $t\mathcal{RPNI}_D$ retourne le même automate A' ; celui-ci représente par définition une requête D-consistante $\mathcal{Q}_{A'}$, qui équivaut à $\mathcal{Q}_A$ (puisque $A' \equiv A$). □

La figure 52 expose un exemple d'exécution de l'algorithme $t\mathcal{RPNI}_D$. Nous considérons le même échantillon que dans l'exemple sans D-consistance de la figure 50 (a). L'automate factorisé représentant le schéma D est montré à côté (b). La définition du schéma n'intervient pas dans la construction de l'automate initial (c) ; il est donc identique au cas précédent. En revanche, les fusions ne sont plus du tout les mêmes. Les premières échouent en raison du test de consistance avec le schéma. La fusion 2-7 est refusée par fonctionnalité. La première fusion acceptée est entre les états 10 et 11 (d). L'algorithme tente ensuite d'autres fusions (1-12, 2-12, etc.), qui échouent encore du fait de la D-consistance. La prochaine fusion entérinée est entre 5 et 12 (e). Les fusions 1-13, 2-13, etc., sont rejetées, toujours en raison de la consistance avec le schéma, jusqu'à 10-13 (f), qui entraîne par déterminisme les fusions 10-14 et 5-15. Il n'y a alors plus d'autre fusion à réaliser. Le résultat

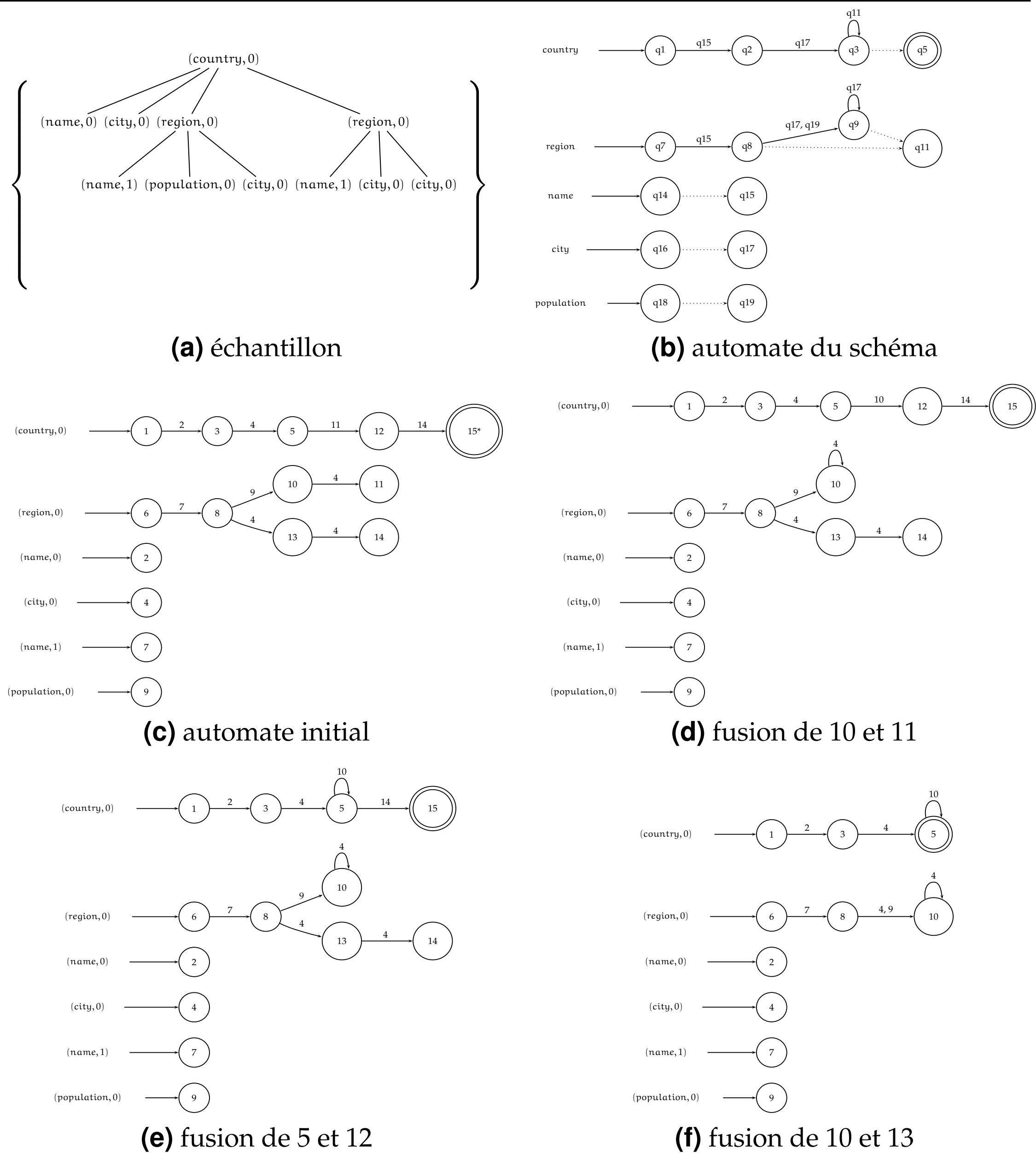


Figure 52. *Exemple d'exécution de l'algorithme* $\mathtt{t}\mathcal{RPNI}_{\mathrm{D}}$.

calculé par l'algorithme est donc ce dernier automate (f). On constate visuellement que la forme de l'automate inféré est proche de celle de l'automate pour le schéma. Notons enfin que dans cet exemple particulier, la requête apprise est quasiment celle visée ; il manque seulement le cas où la région n'a ni population, ni ville, qui n'apparaît pas dans l'échantillon.

Discussion Contrairement à $t\mathcal{RPNI}_D$, l'algorithme $t\mathcal{RPNI}$ (fig. 49) n'est pas complet pour la classe des requêtes D-consistantes ; il lui manque en effet les exemples négatifs induits par la consistance avec le schéma (il peut y en avoir un nombre infini). De ce fait, l'absence du test de consistance le conduit à apprendre des requêtes inconsistantes avec le schéma, ce qui signifie que celles-ci peuvent sélectionner des nœuds dans des arbres qui ne sont pas valides. En principe, on ne soumet pas de tels arbres à la requête apprise, donc garantir sa consistance avec le schéma n'apparaît pas déterminant tant qu'elle agit correctement pour les arbres valides. L'idée est que les généralisations qui violent la consistance sont potentiellement mauvaises, d'où l'intérêt d'ajouter ce test lorsqu'un schéma est disponible pour empêcher ces généralisations. L'objectif est avant tout d'améliorer la qualité de l'apprentissage.

Vérification incrémentale de la consistance

Dans cette section, on se place dans le cas où le schéma D est représenté par un automate factorisé déterministe F. Pour améliorer le temps d'exécution de l'algorithme $t\mathcal{RPNI}_D$, on se repose sur l'incrémentalité du test d'inclusion (cf. sect. 2.3.6 p. 79), plutôt que de l'employer après chaque fusion déterministe.

Après la construction de l'automate initial, on lance l'algorithme d'inclusion sur les automates $\Pi_\Sigma(\mathtt{init}(S))$ et F. Comme le test est vrai par hypothèse, cela a pour effet de calculer les états accessibles du produit $\Pi_\Sigma(\mathtt{init}(S)) \times F$ (relation $\mathsf{acc}(p, q)$), les états "interdits" (relation $\mathsf{frb}(p, q)$), ainsi que les compteurs pour tester l'échec de l'inclusion fail_1 ($l_i(p)$ et $l_i(p, q)$). Pour toutes ces relations, il est possible de précalculer des structures de données en temps $\mathcal{O}(|\mathtt{sta}(\Pi_\Sigma(\mathtt{init}(S)))| \times |\mathtt{sta}(F)|)$ qui permettent un accès et des mises à jour en temps constant. Les fusions sont ensuite simulées par l'ajout d'ϵ-règles dans l'automate A et les structures de données mises à jour en conséquence, en temps $\mathcal{O}(|\Pi_\Sigma(A)| \times |F|)$ dans le pire des cas, beaucoup moins en moyenne (la complexité dépend du nombre de paires d'états potentiellement fusionnés à chaque étape ; le pire des cas survient si tous les états

sont potentiellement fusionnés). Si l'inclusion échoue, un mécanisme de retour arrière (*backtrack*) permet de rétablir l'état antérieur des structures de données, dans le même temps. Les fusions sont matérialisées seulement à la fin de l'apprentissage. Cette méthode s'avère très efficace en temps de calcul dans la pratique.

La complexité dans le pire des cas de l'algorithme $t\mathcal{RPNI}_D$ avec la vérification incrémentale de la consistance reste donc principalement affectée par le test de fonctionnalité, qui est en $\mathcal{O}(|S|^3)$, à comparer avec le test de D-consistance, qui est en $\mathcal{O}(|S| \times |\Sigma| \times |D|)$ (si D est une DTD déterministe représentée par un automate factorisé), soit une complexité globale en $\mathcal{O}(|S|^5 + |S|^3 \times |\Sigma| \times |D|)$ dans le pire des cas. En pratique, le surcoût dû à l'inclusion est compensé par un nombre restreint de fusions déterministes.

4.3.3 Vérification de la consistance par un typage des états

La vérification incrémentale de la consistance discutée au-dessus s'effectue de façon *dynamique*, dans le sens où l'algorithme d'apprentissage intègre un test d'inclusion implicite qui garantit la consistance avec le schéma après chaque fusion déterministe. De plus, cette approche ne modifie pas le résultat d'apprenabilité des requêtes monadiques consistantes avec un schéma (th. 4.8). Nous discutons dans cette section d'une méthode alternative pour vérifier la consistance, que nous qualifions de *statique*, basée sur un typage des états de l'automate initial.

Algorithme $\mathtt{static}\text{-}t\mathcal{RPNI}_D$ pour les requêtes D-consistantes

L'idée de la vérification statique de la consistance est d'utiliser les états de l'automate déterministe représentant le schéma afin de typer les états de l'automate hypothèse. Pour cela, chaque état de l'automate initial est associé à un unique état (son type) de l'automate du schéma — celui-ci étant déterministe, la relation d'accessibilité dans l'automate produit permet cette association —, et toute fusion élémentaire de deux états de types différents est refusée. La vérification de la consistance grâce à un typage *a priori* des états est considérée comme statique dans le sens où les informations sont embarquées dans la structure de l'automate initial.

L'algorithme $\mathtt{static}\text{-}t\mathcal{RPNI}_D$ de la figure 53 est une variante de $t\mathcal{RPNI}$ qui intègre cette approche. Juste après la construction de l'automate initial, une fonction $\mathtt{type}$ (ligne 3) permet d'obtenir le type de chacun de ses états à l'aide du schéma en temps

Entrée: S, un échantillon d'arbres complètement annotés consistant avec le schéma D
Sortie: A, un transducteur de sélection de nœuds consistant avec D

1: $\widehat{S} \leftarrow \{t * \beta \mid t * \beta \in S \wedge \forall v \in \mathtt{nod}(\beta)\ (\beta(v) = 0)\}$
2: $A \leftarrow \mathtt{init}(S \setminus \widehat{S})$
3: $T \leftarrow \mathtt{type}(A, D)$
4: **let** $\mathtt{sta}(A) = \{p_1, \ldots, p_n\}$
5: **for** $i = 2$ **to** n **do**
6:    **for** $j = 1$ **to** $i - 1$ **do**
7:       $A' \leftarrow \mathtt{det\text{-}merge}(A, p_i, p_j, T)$
8:       **if** $\mathcal{L}^u(A')$ est fonctionnel et non-nul
         and A' est compatible avec $\widehat{S}$ **then**
9:          $A \leftarrow A'$
10:          **exit inner loop**
11: **return** A

Figure 53. *Algorithme* $\mathtt{static\text{-}t}\mathcal{RPNI}_D$.

$\mathcal{O}(|\mathtt{init}(S)| \times |\Sigma| \times |D|)$, où D est représenté par un automate (factorisé) déterministe. Les types peuvent être stockés dans un tableau de taille $\mathcal{O}(|\mathtt{sta}(\mathtt{init}(S))|)$. Celui-ci est passé en paramètre de la fonction de fusion déterministe $\mathtt{det\text{-}merge}$ afin d'assurer que toute fusion élémentaire est réalisée entre deux états de même type. Par rapport à l'algorithme $\mathtt{t}\mathcal{RPNI}$, une fusion déterministe peut échouer en raison du typage, auquel cas l'appel $\mathtt{det\text{-}merge}(A, p_i, p_j, T)$ retourne simplement l'automate A. Le reste de l'algorithme demeure inchangé.

Un tel algorithme n'effectue pas de tests d'inclusion. De ce fait, sa complexité dans le pire des cas est moindre que pour la vérification incrémentale (on calcule une seule fois un produit d'automates). En revanche, bien que garantissant en sortie une requête consistante avec le schéma, l'heuristique de typage qu'il intègre s'avère être un biais plus restrictif (en terme de fusions autorisées) que la vérification incrémentale présentée avant, ce qu'illustre la figure 54. L'algorithme $\mathtt{static\text{-}t}\mathcal{RPNI}_D$ conduit donc généralement, pour un même échantillon, à des automates de taille plus grande que $\mathtt{t}\mathcal{RPNI}_D$.

Cependant, un bon typage des états peut se révéler une information précieuse. En effet, prenons par exemple le cas où l'automate cible serait celui en haut à droite de la figure 54. Ici, les états p_1 et p_2 sont distingués par les types q_1 et q_2, qui proviennent de l'automate déterministe représentant le schéma. Un algorithme n'utilisant pas ce typage devra avoir comme exemple négatif explicite le mot $\mathtt{ba}$ dans son échantillon caractéristique afin d'opérer cette même distinction. Cet exemple montre ainsi que, étant donnée une requête

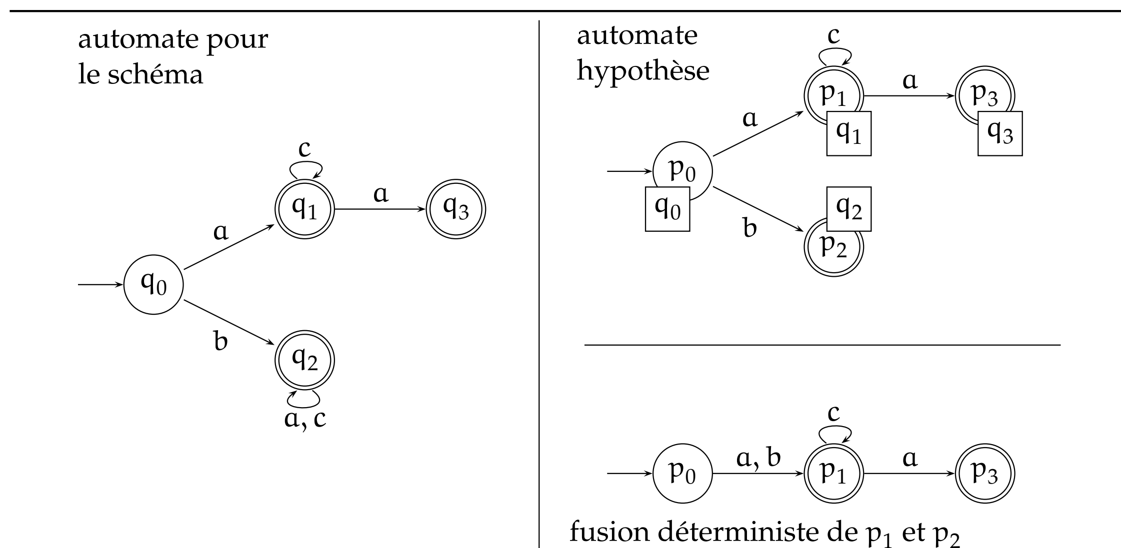


Figure 54. *Comparaison des vérifications statique et dynamique de la consistance dans le cas d'automates finis. Un automate pour un schéma est donné à gauche ; en haut à droite figure un automate hypothèse dont les états sont typés avec les états de l'automate du schéma. Une vérification statique de la consistance empêchera ici la fusion des états* p_1 *et* p_2 *puisqu'ils sont de types différents (resp.* q_1 *et* q_2*) ; au contraire, une vérification dynamique, basée sur un test d'inclusion, autorisera cette fusion déterministe puisque le langage de l'automate résultant demeure inclus dans celui de l'automate pour le schéma.*

cible, les échantillons caractéristiques pour l'identifier par les algorithmes static-t$\mathcal{RPNI}_D$ et t$\mathcal{RPNI}_D$ peuvent être différents.

Conjecture

Les requêtes D*-consistantes sont apprenables avec l'algorithme* static-t$\mathcal{RPNI}_D$.

L'idée générale de la preuve est de ramener l'identification d'un automate cible A à l'identification de l'automate produit $A \times B$, où B est l'automate déterministe qui représente le schéma D. On a en effet par définition $\mathcal{L}(A) \subseteq \mathcal{L}(B) \Leftrightarrow \mathcal{L}(A) = \mathcal{L}(A) \cap \mathcal{L}(B)$. Pour finir la démonstration, on procède ensuite comme pour le théorème 4.8, à savoir se réduire à l'apprentissage de langages réguliers d'arbres. La rédaction de cette preuve est actuellement en cours d'élaboration.

Heuristique des fusions horizontales et consistance

Dans cette section, nous nous intéressons à une heuristique propre à l'induction de requêtes XML, que nous désignons par *heuristique des fusions horizontales*, et la relions avec la vérification statique de la consistance avec un schéma discutée au-dessus.

On observe en pratique que les arbres XML (ou HTML) sont souvent peu profonds mais relativement larges ; cela est lié au fait que les schémas qui les définissent — souvent des DTD — autorisent de nombreuses récursions horizontales tout en limitant les récursions verticales. C'est pourquoi il peut être intéressant, pour un algorithme d'inférence grammaticale qui généralise un langage d'arbres par fusions d'états successives, d'encourager les fusions entre deux états correspondant à des sous-arbres frères, tout en décourageant dans le même temps celles qui concernent des sous-arbres ancêtres. Suivant ce principe, l'heuristique des fusions horizontales empêche les généralisations qui conduisent deux éléments (c.-à-d. étiquettes) distincts à partager le même langage de sous-arbres ; ce biais est qualifié ainsi car il interdit les fusions "verticales" et autorise toutes celles qui sont "horizontales". Notons qu'il peut être utilisé indépendamment de la disponibilité d'un schéma.

Cette heuristique peut être mise en œuvre par un mécanisme de typage d'états. Concrètement, chaque état de l'automate initial est associé à un unique symbole de l'alphabet, qui est donc son type — cela est possible car tout état est accessible à partir d'exactement un symbole dans la représentation graphique de l'automate *stepwise* initial —, et fusionner deux états de types différents est prohibé. La figure 55 apporte une illustration.

L'heuristique des fusions horizontales s'apparente en fait à la vérification statique de la consistance où l'automate déterministe qui représente le schéma a une forme particulière. Celui-ci n'est pas une donnée de l'algorithme mais peut être déduit en temps linéaire comme suit, où $(f, b)(t_1, \ldots, t_n)$ est un sous-arbre de l'échantillon d'arbres complètement annotés S et F l'automate de typage inféré (on rappelle que $t_i(\epsilon)$ correspond à l'étiquette de la racine de l'arbre t_i) :

$$\frac{(f, b)(t_1, \ldots, t_n) \in S}{\begin{array}{c} f \to q_{f,b} \in rul(F) \\ q_{f,b}@q_{t_1(\epsilon)} \to q_{f,b} \in rul(F) \\ \cdots \\ q_{f,b}@q_{t_n(\epsilon)} \to q_{f,b} \in rul(F) \end{array}}$$

Cela revient donc à utiliser un schéma simplifié qui donne l'ensemble des éléments pouvant apparaître sous un autre élément. Une différence avec la vérification statique de la consistance est que le schéma sous-jacent est défini ici sur l'alphabet des arbres annotés, à savoir $\Sigma \times \mathbb{B}$; l'algorithme d'apprentissage avec les fusions horizontales est similaire à celui qui vérifie statiquement la consistance, modulo la remarque qui précède.

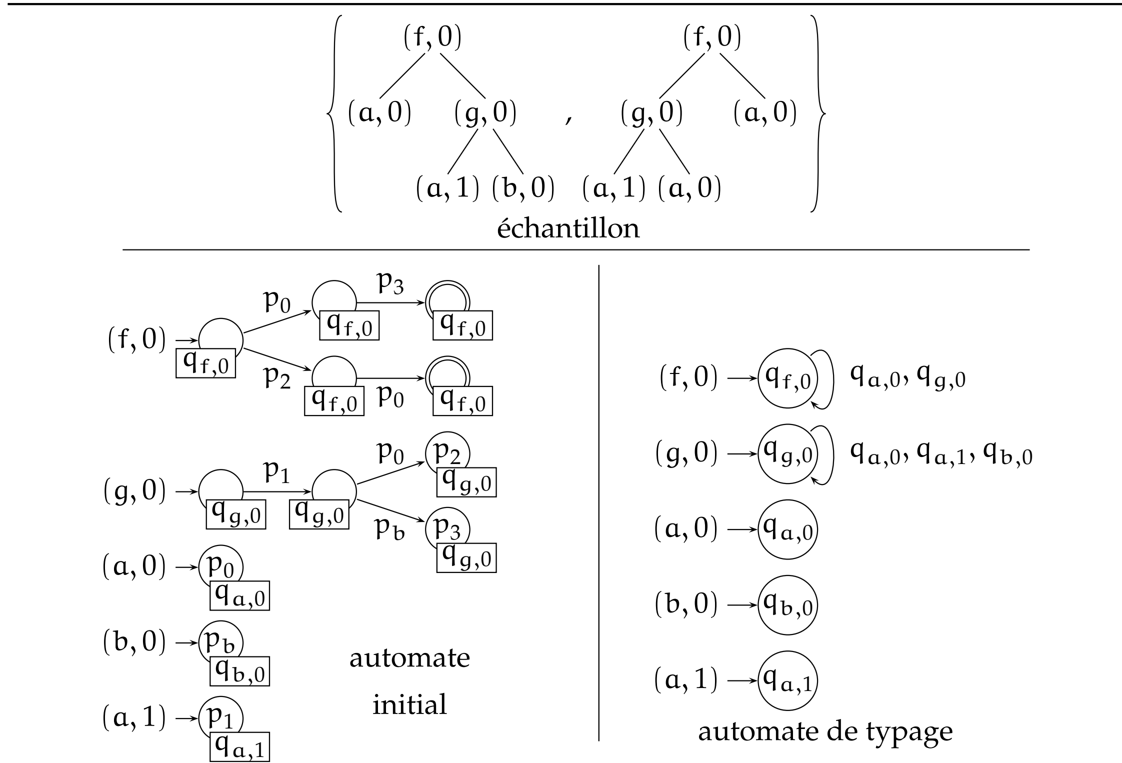


Figure 55. *Typage pour l'heuristique des fusions horizontales.*

Discussion La vérification de la consistance avec un schéma ou l'heuristique des fusions horizontales permettent de contrôler les fusions d'états dans un algorithme d'inférence basé sur $\mathcal{RPNI}$. Les fusions horizontales sont bien adaptées aux données XML et peuvent être employées en l'absence d'un schéma. La vérification de la consistance est indépendante de l'application et tire parti de la présence d'un schéma ; elle peut restreindre certaines fusions horizontales et autoriser des fusions verticales, contrairement à l'heuristique des fusions horizontales. Toutes ces approches apparaissent ainsi complémentaires et leur efficacité dépendra de la tâche d'apprentissage ; la théorie ne permet pas d'en dégager une "meilleure". Nous nous intéresserons aux questions pratiques dans le chapitre suivant, consacré à l'expérimentation.

4.4 Élagage guidé par schéma

L'induction de requêtes monadiques avec des arbres complètement annotés soulève quelques difficultés. D'abord, dans le résultat d'apprenabilité, la polynomialité de l'échantillon caractéristique est donnée par sa cardinalité, mais la taille de cet échantillon, qui dépend de la taille des arbres qui le composent, peut conduire dans certaines situations à un temps de calcul rédhibitoire. Or, dans un arbre annoté, seule une petite portion de l'arbre est généralement "utile" pour déterminer les nœuds sélectionnés et la (ou les) raison(s) de leur sélection par une requête monadique. En outre, les capacités de généralisation de l'algorithme peuvent être perturbées par la présence, dans différentes zones d'un même arbre, de motifs similaires dont les uns contiennent un ou plusieurs nœuds annotés positivement alors que les autres sont dépourvus d'annotations positives. Le temps de calcul ainsi que la qualité de l'apprentissage pourraient donc être considérablement améliorés si la partie "inutile" des arbres annotés n'était pas prise en compte pour l'induction.

Une autre difficulté, d'ordre pratique, peut survenir dans un certain nombre de cas. L'apprentissage de requêtes tel que nous l'avons envisagé jusqu'à présent suppose qu'un utilisateur ou un expert ait annoté des arbres (documents XML ou pages Web) de façon complète. Il n'est pas rare d'être confronté à des situations où un document nécessite beaucoup d'annotations ou bien qu'il soit nécessaire d'annoter beaucoup de documents, ce qui peut apparaître comme une tâche requérant un effort qu'un utilisateur n'est pas en mesure de fournir. Or, du fait de la structure régulière des arbres, on peut considérer dans de nombreux cas que seul un petit nombre d'annotations apportent une information vraiment pertinente à l'algorithme. Il paraîtrait raisonnable, pour alléger la tâche de l'utilisateur, de lui demander de fournir seulement quelques annotations jugées pertinentes et de faire en sorte que l'algorithme "devine" les annotations implicites.

Pour faire face à ces limitations, Carme *et al.* (2007) ont introduit une heuristique d'élagage simple qui permet d'obtenir des résultats relativement intéressants, que ce soit en terme de temps de calcul ou en qualité d'apprentissage. L'idée est à la fois de réduire la taille des arbres donnés en entrée de l'algorithme et de soulager l'utilisateur en lui permettant d'annoter les documents de façon partielle. Nous revisitons, dans la présente section, cette heuristique d'élagage à la lumière de la connaissance du schéma et caractériserons précisément, étant donné un élagage, la classe des requêtes apprenables dans la section suivante.

4.4.1 Fonction élagage

Informellement, un élagage est un mécanisme qui doit permettre de conserver les informations “pertinentes” dans un arbre complètement ou partiellement annoté, et d’en éliminer celles qui sont “inutiles”. Une information pertinente est évidemment un nœud annoté positivement puisqu’il s’agit d’un nœud à extraire ; il est donc judicieux de conserver au moins les nœuds situés sur le chemin qui mène de la racine à un nœud annoté positivement. Il est beaucoup plus difficile de déterminer une information inutile, précisément parce qu’elle n’est pas explicite. Nous verrons d’ailleurs plus loin que des élagages trop brutaux éliminent, à tort, des informations utiles. Quoi qu’il en soit, l’idée est de supprimer des sous-arbres dans lesquels aucun nœud n’est annoté positivement et de les remplacer par un (ou des) nœud(s) générique(s) dont le rôle est de signifier qu’à cette place peut s’enraciner un ensemble de sous-arbres. Les arbres annotés ainsi élagués sont destinés à remplacer les arbres complets dans l’algorithme d’induction de requêtes. Avant d’entrer dans les détails de cette approche heuristique de l’apprentissage, il est nécessaire d’introduire un ensemble de concepts.

Soient Σ un alphabet d’arité non bornée et Γ un ensemble fini de symboles, appelés *symboles d’élagage*. On pose $\Delta = \Sigma \uplus \Gamma$. Un *élagage* $\mathcal{P}$ est une fonction de $T^u_{\Sigma\times\mathbb{B}}$ dans $T^u_{\Delta\times\mathbb{B}}$ telle que pour tout $s \in T^u_{\Sigma\times\mathbb{B}}$, si $s' = \mathcal{P}(s)$ alors :

- $\mathrm{nod}(s') \subseteq \mathrm{nod}(s)$;
- si $s'(v) \in \Sigma \times \mathbb{B}$ alors $s'(v) = s(v)$;
- si $s'(v) \in \Gamma \times \mathbb{B}$ alors v est une feuille, et $s'(v) \in \Gamma \times \{0\}$;
- par ailleurs, si $s(v) \in \Sigma \times \{1\}$ alors $v \in \mathrm{nod}(s')$.

La première condition stipule simplement que tout nœud de l’arbre image est nécessairement défini dans l’arbre antécédent. Les deux points suivants constituent également des conditions sur la structure des arbres obtenus par élagage : le premier assure que les étiquettes des nœuds non élagués sont conservées ; le second indique que les symboles d’élagage sont toujours situés aux feuilles, et qu’ils sont nécessairement annotés négativement. La dernière condition spécifie que les nœuds annotés positivement sont nécessairement conservés ; comme $\mathrm{nod}(s')$ est clos par préfixe, cela implique aussi la conservation des chemins menant de la racine à ces nœuds discutée plus haut.

Pour tout élagage $\mathcal{P}$, nous formulons les deux hypothèses supplémentaires suivantes :

- $\mathcal{P}(s)$ peut être calculé en temps polynomial dans la taille de s pour tout $s \in T^u_{\Sigma\times\mathbb{B}}$;

– pour tout automate A sur $\Sigma \times \mathbb{B}$, on peut calculer, en temps polynomial dans la taille de A, un automate déterministe A' sur $\Delta \times \mathbb{B}$ tel que $\mathcal{L}(A') = \mathcal{P}(\mathcal{L}(A))$.

Cette dernière énonce que les langages d'arbres obtenus par $\mathcal{P}$ doivent être réguliers.

Nous adoptons par ailleurs la terminologie suivante. On appelle un arbre sur $\Sigma \times \mathbb{B}$ *arbre complet*, tandis qu'un arbre sur $\Delta \times \mathbb{B}$ obtenu par $\mathcal{P}$ est un *arbre élagué*. Par extension, si S désigne un ensemble d'arbres complets, on note $\mathcal{P}(S)$ l'ensemble des arbres élagués de S avec $\mathcal{P}(S) = \{s' \in T^{u}_{\Delta \times \mathbb{B}} \mid s \in S \wedge s' = \mathcal{P}(s)\}$. Si s est un arbre complet et $\mathcal{P}$ un élagage, on appelle *partie* (ou *zone*) *élaguée* de l'arbre élagué $s' = \mathcal{P}(s)$, et l'on note $e(s')$, l'ensemble de nœuds suivant :

$$e(s') = \left\{ \nu \in \mathrm{nod}(s) \mid \nu \notin \mathrm{nod}(s') \vee s'(\nu) \in \Gamma \times \{0\} \right\}$$

Autrement dit, la partie élaguée d'un arbre élagué est constituée de l'ensemble des nœuds définis dans l'arbre complet qui ne sont pas définis dans l'arbre élagué et des nœuds (feuilles) étiquetés par un symbole d'élagage. Les autres nœuds, communs aux deux arbres et dont l'étiquette est définie sur $\Sigma \times \mathbb{B}$, forment la *partie non élaguée*, notée $e^{c}(s')$:

$$e^{c}(s') = \left\{ \nu \in \mathrm{nod}(s') \mid s'(\nu) \notin \Gamma \times \{0\} \right\}$$

Notons que, par construction, la zone non élaguée définit un domaine d'arbres.

4.4.2 Élaguer relativement au schéma

Pour l'instant, cette définition générale de l'élagage ne tient pas compte de la disponibilité ou non d'un schéma D. Pour ce faire, l'idée est de relier le schéma aux symboles d'élagage en établissant très simplement $\Gamma = \mathrm{sta}(F)$, où F est l'automate (factorisé) déterministe qui le représente. Ainsi, un nœud étiqueté par un symbole d'élagage — rappelons que, par définition, il ne peut s'agir que d'une feuille dans un arbre d'arité non bornée — est un état q de l'automate F (annoté négativement). Sa sémantique est que tout arbre sur Σ évalué en q par l'automate F peut être "branché" à cette place, ce que nous traduisons ci-après.

Dans toute la suite, on considère un automate factorisé déterministe F représentant un schéma D sur Σ, et on pose $\Gamma = \mathrm{sta}(F)$. On rappelle par ailleurs que $\Delta = \Sigma \uplus \Gamma$. Un F-*élagage* est un élagage défini sur $\mathcal{L}^{u}(F)$. À partir de ce point, et jusqu'à la fin du présent manuscrit,

nous supposerons que tout élagage est un F-élagage. Un arbre élagué $t' * \beta' \in T^{u}_{\Delta \times \mathbb{B}}$ est dit *consistant* avec un arbre $t \in \mathcal{L}^{u}(F)$ si pour tout nœud $v \in nod(t')$, on a :

$$t'(v) = \begin{cases} t(v) & \text{si } t'(v) \in \Sigma\,; \\ q \in \Gamma \text{ tq. } eval^{u}_{b(F)}(t_{|v}) = \{q\} & \text{sinon.} \end{cases}$$

L'existence de q est justifiée par le fait que $t \in \mathcal{L}^{u}(F)$; son unicité par le déterminisme de $b(F)$ (cf. prop. 1.9). L'arbre t est appelé F-*complétion* (ou plus simplement *complétion*) de $t' * \beta'$. L'ensemble des complétions d'un arbre élagué est dénoté c_F, et défini comme la fonction de $T^{u}_{\Delta \times \mathbb{B}}$ dans $2^{\mathcal{L}^{u}(F)}$ telle que :

$$c_F(t' * \beta') = \{t'[t_1 \mapsto v_1, \ldots, t_n \mapsto v_n] \mid eval^{u}_{b(F)}(t_1) = t'(v_1), \ldots, eval^{u}_{b(F)}(t_n) = t'(v_n)\}$$

L'ensemble $c_F(t' * \beta')$ est l'ensemble des arbres de $\mathcal{L}^{u}(F)$ avec lesquels $t' * \beta'$ est consistant. La notation $t_i \mapsto v_i$ désigne ici la substitution d'une feuille v_i de t' dont l'étiquette comporte un état de F par un sous-arbre t_i sur Σ évalué dans cet état par F. Notez que si $\mathcal{Q}$ est une requête monadique et $\mathcal{P}$ un élagage, alors pour tout $t \in \mathcal{L}^{u}(F)$ on a par définition $t \in c_F(\mathcal{P}(t * \mathcal{Q}(t)))$. Dit autrement, un arbre annoté par une requête appartient à l'ensemble des F-complétions de son élagage.

Définition 4.9 (F-compatibilité)
Deux arbres élagués $t'_1 * \beta'_1$ et $t'_2 * \beta'_2$ dans $T^{u}_{\Delta \times \mathbb{B}}$ sont dits *compatibles relativement à* F (ou F-*compatibles*) si et seulement si, s'il existe $t \in \mathcal{L}^{u}(F)$ tel que $t'_1 * \beta'_1$ et $t'_2 * \beta'_2$ sont tous deux consistants avec t, alors pour tout nœud $v \in nod(t'_1 * \beta'_1) \cap nod(t'_2 * \beta'_2)$, si $t'_1(v) = t'_2(v)$ alors $\beta'_1(v) = \beta'_2(v)$. ◄

Autrement dit, deux arbres élagués sont F-compatibles si leurs annotations coïncident sur les nœuds qu'ils partagent avec toute F-complétion commune. La figure 56 permet d'illustrer les notions abordées jusqu'à présent. Notez que dans cette figure, ainsi que dans les suivantes, les symboles d'élagages sont présentés sans leur annotation, qui est toujours négative par définition.

4.4.3 Transducteurs de sélection de nœuds élagueurs

La fonctionnalité est une notion essentielle dans la définition des transducteurs de sélection de nœuds. Elle garantit, pour tout arbre, l'unicité de son annotation par une

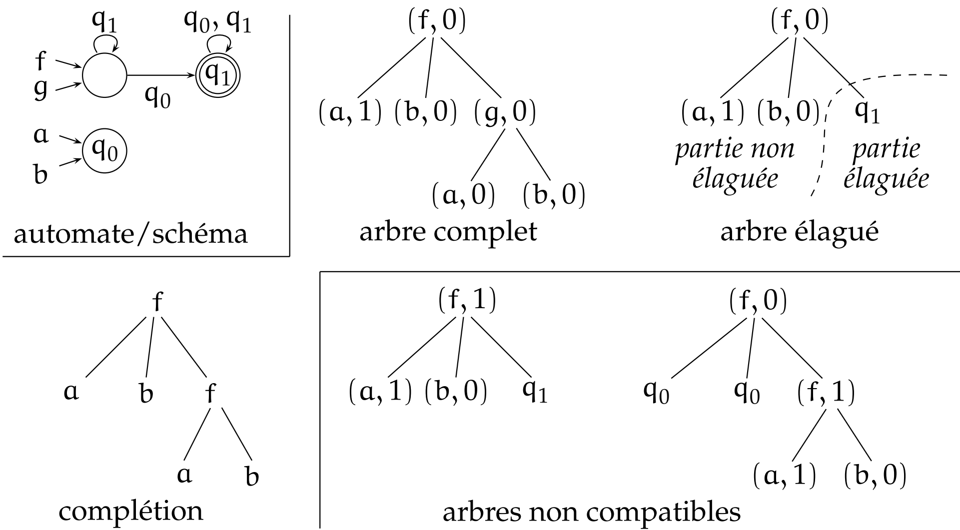


Figure 56. *Illustration de l'élagage avec un schéma. En haut, un automate stepwise représentant un schéma (déterministe, mais non factorisé pour simplifier), un arbre complet et un arbre élagué avec le schéma (selon un élagage arbitraire) consistant. En bas à droite, deux arbres élagués non compatibles (leurs racines sont annotées différemment), consistants avec la complétion en bas à gauche. Pour faciliter la lecture, les symboles d'élagage (*q_0 *et* q_1*) sont présentés sans leur annotation, qui est toujours négative par définition.*

requête (monadique). Cette propriété est définie pour des langages d'arbres complets. Nous définissons une notion de pseudo-fonctionnalité pour des langages d'arbres élagués, relativement à un schéma.

Définition 4.10 (F-pseudo-fonctionnalité)
Considérons un langage d'arbres élagués $L \subseteq T^{u}_{\Delta\times\mathbb{B}}$. On dit que L est *pseudo-fonctionnel relativement à* F (ou F-*pseudo-fonctionnel*) si pour toute paire d'arbres s'_1 et s'_2 de L, on a s'_1 et s'_2 F-*compatibles* entre eux. ◀

Notre objectif est d'apprendre un automate à partir d'un ensemble d'arbres élagués, autrement dit un automate d'arbres (déterministe) sur $\Delta \times \mathbb{B}$, qui reconnaît un langage F-pseudo-fonctionnel. Pour cela, nous introduisons les transducteurs de sélection de nœuds "élagueurs" relativement au schéma (abrégé en TSNé$_F$).

Définition 4.11 (Transducteur de sélection de nœuds F-élagueur)
Un *transducteur de sélection de nœuds* F-*élagueur* (TSNé$_F$) sur Σ est un automate *stepwise* sur $\Delta \times \mathbb{B}$ dont le langage est F-pseudo-fonctionnel et non-nul. ◀

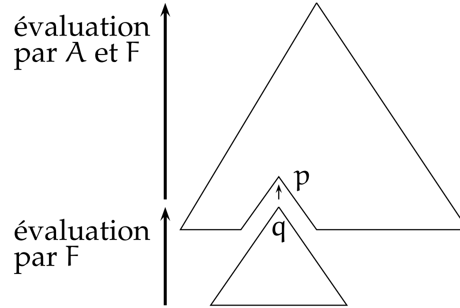


Figure 57. *Jonction entre parties élaguée et non élaguée avec un TSNé$_F$ A sur Σ et F l'automate factorisé déterministe qui représente le schéma.*

Notez qu'un TSNé A sur Σ est nécessairement un TSN sur Δ. En effet, supposons le contraire, à savoir que $\mathcal{L}^u(A)$ n'est pas fonctionnel. Dans ce cas, il existe $t' * \beta'_1$ et $t' * \beta'_2$ dans $\mathcal{L}^u(A)$ tels que $\beta'_1 \neq \beta'_2$. Par conséquent, il existe $v \in \mathtt{nod}(t')$ tel que $\beta'_1(v) \neq \beta'_2(v)$. Or ceci est impossible par définition de compatibilité. Donc $\mathcal{L}^u(A)$ est fonctionnel. Comme il est également non-nul par définition, on en déduit que A est un TSN.

Les TSNé opèrent similairement aux TSN hormis le fait qu'ils doivent d'abord simuler un élagage des arbres de façon non déterministe puis sélectionner des nœuds dans les arbres élagués. Nous détaillons leur fonctionnement avant de nous intéresser à un test de F-pseudo-fonctionnalité efficace.

Définition de requêtes par TSN élagueur

En tant qu'automate d'arbres sur $\Delta \times \mathbb{B}$, un TSNé A peut reconnaître seulement la partie non élaguée d'un arbre complet (c.-à-d. sur $\Sigma \times \mathbb{B}$) ; pour évaluer la zone élaguée de l'arbre, on recourt à l'automate représentant le schéma, dont les états sont utilisés comme symboles d'élagage par l'automate A. Ces derniers permettent d'effectuer la jonction entre parties élaguée et non élaguée, comme l'illustre la figure 57.

Pour opérer sur des arbres sur Σ avec un TSNé$_F$ A, on construit un automate *stepwise* (non déterministe) $j(A, F)$, appelé *automate de jonction*, où F est l'automate factorisé déterministe qui représente le schéma. La construction s'apparente à celle d'un produit d'automates pour reconnaître l'union de deux langages. L'automate $j(A, F)$ est défini sur l'alphabet $\Sigma \times \mathbb{B}$. Ses états sont $\mathtt{sta}(j(A, F)) = \mathtt{sta}(A) \uplus \mathtt{sta}(F)$ et ses états finaux $\mathtt{fin}(j(A, F)) = \mathtt{fin}(A) \uplus \mathtt{fin}(F)$. Les règles de $j(A, F)$ sont obtenues par l'ensemble des

$$\frac{(a,b) \to p \in rul(A) \qquad a \in \Sigma}{(a,b) \to p \in rul(j(A,F))} \qquad \frac{a \to q \in rul(F)}{(a,0) \to q \in rul(j(A,F))}$$

$$\frac{(q,0) \to p \in rul(A) \qquad q \in \Gamma}{q \xrightarrow{\epsilon}_{j(A,F)} p} \qquad \frac{q' \xrightarrow{\epsilon}_F q}{q' \xrightarrow{\epsilon}_{j(A,F)} q}$$

$$\frac{p_1@p_2 \to p \in rul(A)}{p_1@p_2 \to p \in rul(j(A,F))} \qquad \frac{q_1@q_2 \to q \in rul(F)}{q_1@q_2 \to q \in rul(j(A,F))}$$

Figure 58. *Transformation d'un TSNé$_F$ A sur Σ en automate de jonction $j(A,F)$ sur $\Sigma \times \mathbb{B}$, où F est l'automate factorisé déterministe qui représente le schéma.*

transformations présentées à la figure 58. Notons que $j(A,F)$ n'est pas nécessairement un TSN (non élagueur) sur Σ.

Lemme 4.12
Tout TSNé$_F$ A sur Σ peut être transformé en temps linéaire en un automate de jonction $j(A,F)$ sur $\Sigma \times \mathbb{B}$ tel que $j(A,F)$ définit la même requête que A.

La transformation s'effectue clairement en temps linéaire $\mathcal{O}(|A| + |F|)$. Par ailleurs, l'automate $j(A,F)$ définit la même requête que A. En effet, elle préserve les règles constantes de la forme $(a,1) \to p$ de A et n'ajoute pas de règles permettant la sélection d'autres nœuds. Un transducteur de sélection de nœuds élagueur permet de définir une requête monadique dans le sens suivant.

Définition 4.13
Un transducteur de sélection de nœuds F-élagueur A sur Σ définit une requête monadique $\mathcal{Q}_A$ pour tout arbre t de $\mathcal{L}^u(F)$ telle que :

$$\mathcal{Q}_A(t) = \{v \in nod(t) \mid \exists \beta \in T^u_{\mathbb{B}} \text{ tq. } t * \beta \in \mathcal{L}^u(j(A,F)) \wedge \beta(v) = 1\}$$ ◀

Notez que la même requête peut être définie alternativement, sans utiliser l'automate de jonction, comme $\mathcal{Q}_A(t) = \{v \in nod(t) \mid \exists t' * \beta' \in \mathcal{L}^u(A) \text{ consistant avec } t \wedge \beta'(v) = 1\}$.

Théorème 4.14
La réponse d'une requête $\mathcal{Q}_A(t)$ pour un transducteur de sélection de nœuds F-élagueur A et un arbre t de $\mathcal{L}^u(F)$ peut être calculée en temps $\mathcal{O}(|t| \times (|A| + |F|))$.

Le calcul de la réponse d'une requête avec un transducteur de sélection de nœuds élagueur fonctionne de façon analogue au cas non élagueur. Il est expliqué à l'annexe A.1.

Vérification efficace de la pseudo-fonctionnalité

Dans cette section, nous montrons comment vérifier la propriété de pseudo-fonctionnalité en temps polynomial. On considère toujours un schéma représenté par un automate factorisé déterministe F sur Σ. Pour décider si un automate *stepwise* déterministe A sur $\Delta \times \mathbb{B}$ est un TSNé$_F$, autrement dit si son langage est F-pseudo-fonctionnel (et non-nul), nous construisons un programme Datalog clos de taille polynomiale à partir de A et de F et caractérisons la propriété de F-pseudo-fonctionnalité par l'intermédiaire de ses prédicats.

L'idée du test de pseudo-fonctionnalité est de détecter si l'automate A reconnaît des arbres élagués non compatibles, ce qu'établit la caractérisation suivante.

Définition 4.15
Le langage $\mathcal{L}^u(A)$ d'un automate A sur $\Delta \times \mathbb{B}$ n'est pas F-pseudo-fonctionnel sous la condition suivante.

$A, F \models \mathsf{npf} \overset{\text{déf.}}{\Leftrightarrow}$ il existe deux arbres s_1 et s_2 dans $\mathcal{L}^u(A)$ tels que s_1 et s_2 ne sont pas compatibles relativement à F. ◀

Le lemme suivant est une conséquence directe des définitions précédentes.

Lemme 4.16
Soit A un automate stepwise déterministe sur $\Delta \times \mathbb{B}$ et F un automate factorisé déterministe sur Σ représentant un schéma. Le langage de A est pseudo-fonctionnel relativement à F si et seulement si $A, F \not\models \mathsf{npf}$.

Nous introduisons le prédicat $\texttt{inc}(p, p')$, défini entre deux états p et p' de A, dont le but est de montrer que l'automate A peut évaluer dans ces états deux arbres élagués consistants avec une même complétion mais non compatibles. Sa sémantique est donnée ci-dessous.

$A, F \models \mathsf{inc}(p, p') \overset{\text{déf.}}{\Leftrightarrow}$ il existe $t_0 \in \mathcal{L}^u(F)$, deux arbres élagués $t * \beta$ et $t' * \beta'$ de $T^u_{\Delta\times\mathbb{B}}$ consistants avec t_0 et une position ν avec $t(\nu) = t'(\nu)$ et $\beta(\nu) \neq \beta'(\nu)$ tels que $eval^u_A(t * \beta) = \{p\}$ et $eval^u_A(t' * \beta') = \{p'\}$.

Nous utilisons deux relations intermédiaires pour décomposer le problème. Le prédicat $\mathtt{sim}(p, p')$ permet de relier deux états de l'automate A qui jouent un rôle similaire dans deux arbres élagués consistants avec une même complétion.

$A, F \models \mathsf{sim}(p, p') \overset{\text{déf.}}{\Leftrightarrow}$ il existe $t_0 \in \mathcal{L}^u(F)$ et deux arbres élagués $t * \beta$ et $t' * \beta'$ de $T^u_{\Delta \times \mathbb{B}}$ consistants avec t_0 tels que $eval^u_A(t * \beta) = \{p\}$ et $eval^u_A(t' * \beta') = \{p'\}$.

Le prédicat $\mathtt{sch}(p, q)$ associe des états p de A à des états q de F. Son but est de permettre le branchement d'un arbre élagué à un autre arbre au niveau de ses symboles d'élagages, qui sont précisément des états de F.

$A, F \models \mathsf{sch}(p, q) \overset{\text{déf.}}{\Leftrightarrow}$ il existe $t_0 \in \mathcal{L}^u(F)$ et un arbre élagué $t * \beta \in T^u_{\Delta \times \mathbb{B}}$ consistant avec t_0 tels que $eval^u_A(t * \beta) = \{p\}$ et $eval^u_{\mathsf{b}(F)}(t_0) = \{q\}$.

La figure 59 présente un ensemble de règles pour construire un programme Datalog clos $\mathcal{D}(A, F)$ à partir de A et de F.

Lemme 4.17
Soit $L = \mathtt{lfp}(\mathcal{D}(A, F))$*, alors :*

1. $A, F \models \mathsf{sch}(p, q)$ *si et seulement si* $\mathtt{sch}(p, q) \in L$ *;*
2. $A, F \models \mathsf{sim}(p, p')$ *si et seulement si* $\mathtt{sim}(p, p') \in L$ *;*
3. $A, F \models \mathsf{inc}(p, p')$ *si et seulement si* $\mathtt{inc}(p, p') \in L$*.*
4. $A, F \models \mathsf{npf}$ *si et seulement si* $\mathtt{npf} \in L$*.*

Preuve
La preuve est analogue à celle du lemme 2.18 p. 82. Les implications de la droite vers la gauche se montrent par induction simultanée sur la définition du point fixe. Dans l'autre sens, on applique les définitions et on opère par induction sur la structure des arbres. □

Proposition 4.18
Soit A *un automate stepwise déterministe sur* $\Delta \times \mathbb{B}$ *et* F *un automate factorisé déterministe sur* Σ *représentant un schéma. Le langage de* A *est pseudo-fonctionnel relativement à* F *si et seulement si* $\mathtt{npf} \notin \mathtt{lfp}(A, F)$*.*

Preuve
Lemmes 4.16 et 4.17. □

Théorème 4.19
On peut décider en temps $\mathcal{O}(|A| \times |F| + |A|^2)$ *si le langage d'un automate stepwise déterministe* A *est pseudo-fonctionnel relativement à un schéma représenté par un automate factorisé déterministe* F*.*

$$(\mathrm{Sch}_1) \quad \frac{(a,b) \to p \in \mathrm{rul}(A) \qquad a \to q \in \mathrm{rul}(F)}{\mathtt{sch}(p,q).}$$

$$(\mathrm{Sch}_2) \quad \frac{(q,0) \to p \in \mathrm{rul}(A) \qquad q \in \mathrm{sta}(F)}{\mathtt{sch}(p,q).}$$

$$(\mathrm{Sch}_3) \quad \frac{p_1@p_2 \to p \in \mathrm{rul}(A) \qquad q_1@q_2 \to q \in \mathrm{rul}(F)}{\mathtt{sch}(p,q) :- \mathtt{sch}(p_1,q_1), \mathtt{sch}(p_2,q_2).}$$

$$(\mathrm{Sch}_4) \quad \frac{q' \xrightarrow{\epsilon}_F q}{\mathtt{sch}(p,q) :- \mathtt{sch}(p,q').}$$

$$(\mathrm{Sim}_1) \quad \frac{p \in \mathrm{sta}(A)}{\mathtt{sim}(p,p).}$$

$$(\mathrm{Sim}_2) \quad \frac{(q,0) \to p' \in \mathrm{rul}(A) \qquad q \in \mathrm{sta}(F)}{\mathtt{sim}(p,p') :- \mathtt{sch}(p,q).}$$

$$(\mathrm{Sim}_3) \quad \frac{p_1@p_2 \to p \in \mathrm{rul}(A) \qquad p_1'@p_2' \to p' \in \mathrm{rul}(A)}{\mathtt{sim}(p,p') :- \mathtt{sim}(p_1,p_1'), \mathtt{sim}(p_2,p_2').}$$

$$(\mathrm{Inc}_1) \quad \frac{(a,b) \to p \in \mathrm{rul}(A) \qquad (a,\neg b) \to p' \in \mathrm{rul}(A)}{\mathtt{inc}(p,p').}$$

$$(\mathrm{Inc}_2) \quad \frac{p_1@p_2 \to p \in \mathrm{rul}(A) \qquad p_1'@p_2' \to p' \in \mathrm{rul}(A)}{\begin{array}{c}\mathtt{inc}(p,p') :- \mathtt{sim}(p_1,p_1'), \mathtt{inc}(p_2,p_2'). \\ \mathtt{inc}(p,p') :- \mathtt{inc}(p_1,p_1'), \mathtt{inc}(p_2,p_2'). \\ \mathtt{inc}(p,p') :- \mathtt{inc}(p_1,p_1'), \mathtt{sim}(p_2,p_2').\end{array}}$$

$$(\mathrm{Npf}) \quad \frac{p, p' \in \mathrm{fin}(A)}{\mathtt{npf} :- \mathtt{inc}(p,p').}$$

Figure 59. *Ensemble de règles pour transformer un automate stepwise déterministe* A *sur* $\Delta \times \mathbb{B}$ *et un automate factorisé déterministe* F *sur* Σ *en un programme Datalog clos* $\mathcal{D}(A, F)$ *permettant de vérifier si le langage de* A *est pseudo-fonctionnel relativement au schéma représenté par* F.

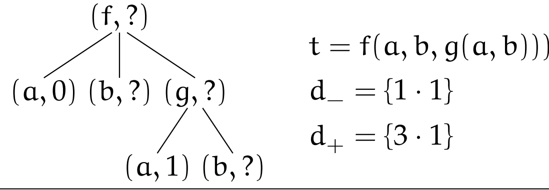


Figure 60. *Représentation graphique d'un arbre partiellement annoté. Le symbole "?" accompagne les nœuds dont l'annotation positive ou négative n'est pas spécifiée.*

Preuve
Le programme $\mathcal{D}(A, F)$ peut être obtenu en temps $\mathcal{O}(|A| \times |F| + |A|^2)$ à partir de A et de F avec les règles de transformation de la fig. 59 ; sa taille est également en $\mathcal{O}(|A| \times |F| + |A|^2)$. Par conséquent, le plus petit point fixe de $\mathcal{D}(A, F)$ peut être calculé en temps $\mathcal{O}(|A| \times |F| + |A|^2)$ (th. 2.1). On vérifie ensuite la présence de `npf` dans $\mathrm{lfp}(\mathcal{D}(A, F))$, auquel cas A n'est pas pseudo-fonctionnel relativement à F. □

En corollaire, on peut décider efficacement si un automate *stepwise* A sur $\Delta \times \mathbb{B}$ est un TSN élagueur (relativement à F), en vérifiant si son langage est F-pseudo-fonctionnel grâce au théorème 4.19, d'une part, et s'il est non-nul grâce au test présenté à la figure 45 p. 104, d'autre part.

4.4.4 Algorithme d'apprentissage avec élagage $t\mathcal{RPNI}_{D}^{\mathcal{P}}$

Nous adaptons l'algorithme d'apprentissage $t\mathcal{RPNI}$ à l'heuristique d'élagage relative à un schéma. L'élagage a pour but, d'une part, de supprimer des zones inutiles pour l'apprentissage et, d'autre part, de limiter le nombre d'annotations nécessaires de la part de l'utilisateur. Nous considérons donc un échantillon composé d'arbres annotés partiellement, dans le sens où l'information sur les nœuds sélectionnés par la requête cible est potentiellement incomplète. Nous définissons les arbres partiellement annotés et présentons un algorithme d'apprentissage avec élagage.

Formellement, un *arbre partiellement annoté* sur Σ est un triplet (t, d_+, d_-), où t est un arbre sur Σ, et d_+ (resp. d_-) un sous-ensemble de nœuds de t annotés positivement (resp. négativement) avec $d_+ \cap d_- = \emptyset$. Un arbre partiellement annoté (t, d_+, d_-) est *consistant avec une requête* (monadique) $\mathcal{Q}$ si $d_+ \subseteq \mathcal{Q}(t)$ et $d_- \cap \mathcal{Q}(t) = \emptyset$; il est *consistant avec un schéma* D si $t \in \mathcal{L}(D)$. Notons qu'un arbre complètement annoté $t * \beta$ est un arbre partiellement annoté (t, d_+, d_-) tel que $\mathrm{nod}(t * \beta) = d_+ \uplus d_-$ et pour tout nœud

Paramètres:
  D, un schéma représenté par un automate factorisé déterministe F
  $\mathcal{P}$, un F-élagage
Entrée: $\widetilde{S}$, un ensemble d'arbres partiellement annotés consistants avec D
Sortie: A, un transducteur de sélection de nœuds F-élagueur
1: $S' \leftarrow \{\mathcal{P}(t * d_+) \mid (t, d_+, d_-) \in \widetilde{S} \wedge d_+ \neq \emptyset\}$
2: $A \leftarrow \mathtt{init}(S')$
3: **let** $\mathtt{sta}(A) = \{p_1, \ldots, p_n\}$ *// on suppose les états de* A *convenablement ordonnés*
4: **for** $i = 2$ **to** n **do**
5:   **for** $j = 1$ **to** $i - 1$ **do**
6:     $A' \leftarrow \mathtt{det\text{-}merge}(A, p_i, p_j)$
7:     **if** $\mathcal{L}^u(A')$ est F-pseudo-fonctionnel et non-nul
       and A' est compatible avec $\widetilde{S}$
       and A' est consistant avec D **then**
8:       $A \leftarrow A'$
9:       **exit inner loop**
10: **return** A

Figure 61. *Algorithme* $t\mathcal{RPNI}_D^{\mathcal{P}}$.

$v \in \mathtt{nod}(\beta)$, on a $\beta(v) = 0 \Leftrightarrow v \in d_-$, et $\beta(v) = 1 \Leftrightarrow v \in d_+$. La figure 60 présente une représentation graphique pour les arbres partiellement annotés.

L'algorithme $t\mathcal{RPNI}_D^{\mathcal{P}}$ présenté à la figure 61 est paramétré par un élagage $\mathcal{P}$ et un schéma D. Comme toujours, on suppose D représenté par un automate factorisé déterministe F. De plus, $\mathcal{P}$ doit être un F-élagage. Par rapport à l'algorithme d'apprentissage traditionnel, $t\mathcal{RPNI}_D^{\mathcal{P}}$ prend en entrée un ensemble d'arbres *partiellement* annotés $\widetilde{S}$, et la première opération qu'il effectue (ligne 1) est de construire un échantillon d'arbres élagués S' à partir de $\widetilde{S}$ avec la fonction d'élagage $\mathcal{P}$. La suite de l'algorithme est similaire à l'algorithme de base, à savoir qu'un automate initial est construit puis que des fusions déterministes d'états sont effectuées.

Toutefois, par rapport au cas non élagué, le test de pseudo-fonctionnalité ne suffit pas pour garantir la consistance de l'échantillon $\widetilde{S}$ avec l'automate appris. Il est en effet possible que le TSNé en cours d'inférence autorise la sélection de nœuds explicitement négatifs dans les arbres de l'échantillon partiellement annoté $\widetilde{S}$. Cela est dû au fait que l'élagage ne conserve pas forcément cette information. Par conséquent, il est nécessaire d'assurer la compatibilité de l'automate A' avec tous les arbres de l'échantillon (ligne 7), en vérifiant

la propriété :

$$\mathcal{Q}_{A'}(t) \cap d_- = \emptyset \text{ pour tout } (t, d_+, d_-) \in \widetilde{S}$$

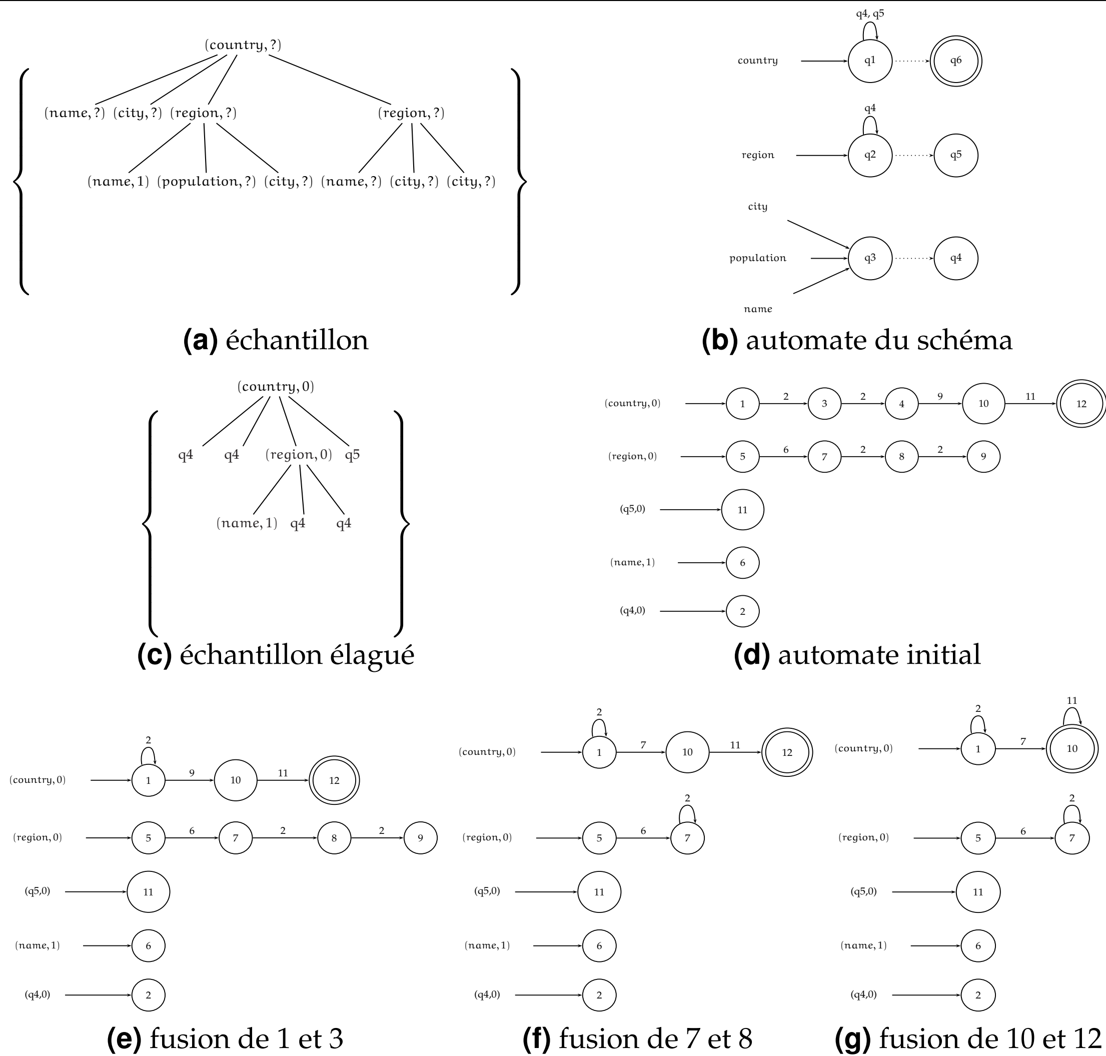
Autrement dit, la requête apprise ne doit sélectionner, pour chaque arbre partiellement annoté de l'échantillon d'entrée, aucun nœud dans l'ensemble de ceux qui sont explicitement négatifs. En revanche, vérifier $\mathcal{Q}_{A'}(t) \supseteq d_+$ s'avère inutile étant donné que l'élagage préserve par définition l'information sur les nœuds positivement annotés. La compatibilité de A' avec $\widetilde{S}$ peut être vérifiée en temps polynomial $\mathcal{O}(|\widetilde{S}| \times (|A'| + |F|))$ puisqu'il s'agit, pour chaque arbre t de l'échantillon d'arbres partiellement annotés $\widetilde{S}$, d'évaluer $\mathcal{Q}_{A'}(t)$, ce qui s'effectue en temps $\mathcal{O}(|t| \times (|A'| + |F|))$ d'après le théorème 4.14. Ce test de compatibilité est plus fort que le test de compatibilité de l'algorithme de base $t\mathcal{RPNI}$ car il s'applique à tout l'échantillon.

Par ailleurs, pour appliquer le test de consistance avec le schéma, il est nécessaire de prendre en compte le fait que l'automate A' reconnaît des arbres élagués. On peut utiliser l'automate de jonction $j(A', F)$ qui définit la même requête que A', et maintenir l'inclusion $\mathcal{L}^u(\Pi_\Sigma(j(A', F))) \subseteq \mathcal{L}^u(F)$ similairement au cas non élagué (cf. sect. 4.3.2). Le coût de cette approche pour tester l'inclusion est cependant en $\mathcal{O}((|\widetilde{S}| + |F|) \times |F|)$. Une implémentation plus efficace consiste à utiliser l'algorithme d'inclusion de la section 2.4 pour vérifier $\mathcal{L}^u(\Pi_\Delta(A')) \subseteq \mathcal{L}^u(F)$ en ajoutant une règle de transformation (Acc_5) qui produit des littéraux $\mathtt{acc}(p, q)$ dans le point fixe de $\mathcal{D}_1(A', F)$:

$$(\mathrm{Acc}_5) \quad \frac{q \to p \in rul(\Pi_\Delta(A'))}{\mathtt{acc}(p, q).}$$

Cette règle permet de traiter simplement le cas des symboles d'élagage. Par cet artifice, la consistance avec le schéma peut être vérifiée efficacement en temps $\mathcal{O}(|\widetilde{S}| \times |F|)$.

La figure 62 propose un exemple d'exécution de l'algorithme $t\mathcal{RPNI}^{\mathcal{P}}_{D}$. L'échantillon est composé d'un unique arbre partiellement annoté (a). Le schéma utilisé diffère de celui des exemples précédents (b). La phase d'élagage avec la fonction $\mathcal{P}$ — non explicitée ici — produit l'échantillon montré ensuite (c), avec deux symboles q4 et q5 provenant de l'automate du schéma. L'automate initial est construit à partir de cet échantillon (d). Étant données les contraintes de pseudo-fonctionnalité et de D-consistance, l'algorithme réalise les trois fusions déterministes 1-3 (e), 7-8 (f) et 10-12 (g). Le TSNé inféré est le dernier obtenu.



(a) échantillon **(b)** automate du schéma

(c) échantillon élagué **(d)** automate initial

(e) fusion de 1 et 3 **(f)** fusion de 7 et 8 **(g)** fusion de 10 et 12

Figure 62. *Exemple d'exécution de l'algorithme* $t\mathcal{RPNI}_{\mathrm{D}}^{\mathcal{P}}$.

Discussion L'élagage permet de pallier certaines limitations de l'inférence à partir d'exemples complètement annotés en réduisant la taille des arbres d'entrée et en concentrant l'effort d'apprentissage dans des zones estimées pertinentes. Son utilisation est particulièrement appropriée dans un cadre interactif où il est demandé à l'utilisateur de fournir quelques annotations afin de l'assister dans la définition d'une requête ; elle peut aussi se révéler judicieuse dans un contexte où l'on dispose de toutes les annotations puisque le cas complètement annoté peut être vu comme un cas particulier du partiellement annoté.

En contrepartie, la qualité de l'algorithme repose principalement sur la fonction d'élagage utilisée. Si celle-ci conserve trop d'information inutile, le résultat peut être médiocre ; si elle supprime l'information pertinente, l'identification de la requête n'est plus garantie. Plus généralement, l'inférence de transducteurs de sélection de nœuds élagueurs n'hérite pas "gratuitement" du résultat théorique d'apprenabilité des TSN non élagueurs. La caractérisation de la classe des requêtes apprenables par l'algorithme $t\mathcal{RPNI}_{\mathrm{D}}^{\mathcal{P}}$ nécessite un travail de formalisation, que nous menons dans la section qui suit.

4.5 Apprenabilité des requêtes stables par élagage

Pour introduire informellement le problème discuté à la fin de la section précédente, nous devisons ci-dessous un exemple qui montre que l'emploi d'un élagage qui ne conserve pas suffisamment d'information conduit à l'impossibilité d'identifier certaines requêtes monadiques par l'algorithme $t\mathcal{RPNI}_{\mathrm{D}}^{\mathcal{P}}$, même si l'information est explicitée dans l'échantillon non élagué.

Considérons pour cela les deux arbres t_1 et t_2 sur $\Sigma = \{f, a, b\}$ suivants :

On suppose t_1 et t_2 consistants avec un certain schéma D qu'il n'est pas nécessaire de préciser ici. On cherche à définir par un TSNé la requête monadique $\mathcal{Q}$ qui sélectionne tous les nœuds étiquetés par a dont le prédécesseur est un nœud étiqueté par b. Appliquée à t_1 et t_2, elle produirait $s_1 = t_1 * \mathcal{Q}(t_1)$ et $s_2 = t_2 * \mathcal{Q}(t_2)$ comme suit :

On considère maintenant l'échantillon d'arbres partiellement annotés constitué des deux arbres précédents [3], c'est-à-dire $\widetilde{S} = \{s_1, s_2\}$, et on suppose un élagage correctement défini $\mathcal{P}$ qui produit, à partir de s_1 et s_2, les deux arbres élagués $s_1' = \mathcal{P}(s_1)$ et $s_2' = \mathcal{P}(s_2)$ suivants, où $\Gamma = \{q\}$:

Il apparaît clairement ici qu'il est sans espoir d'identifier la requête cible $\mathcal{Q}$ avec l'algorithme $t\mathcal{RPNI}_D^{\mathcal{P}}$ en employant l'élagage $\mathcal{P}$. En effet, la sélection d'un nœud étiqueté par a dépend de la présence d'un frère gauche étiqueté par b. Or l'élagage substitue un état générique q à tout nœud étiqueté par a ou par b annoté négativement ; par conséquent, une fois élagués, plus rien ne permet de les distinguer.

Pourtant, $\mathcal{Q}$ est une requête monadique. À ce titre, elle peut théoriquement être identifiée, à partir d'exemples complètement annotés, par un algorithme d'apprentissage sans élagage (par exemple $t\mathcal{RPNI}$). Les arbres s_1 et s_2 font en outre partie de l'échantillon caractéristique. Mais ils ne sont plus d'aucune utilité dès lors qu'ils ont été élagués par $\mathcal{P}$. Ainsi se pose la question suivante : si un échantillon d'arbres annotés est caractéristique pour un algorithme d'inférence sans élagage, dans quelles conditions reste-t-il caractéristique pour un algorithme avec élagage ?

Nous répondons à cette question en établissant une caractérisation de la classe des requêtes que l'on peut identifier avec l'algorithme $t\mathcal{RPNI}_D^{\mathcal{P}}$ présenté à la section précédente. À cet effet, nous définissons une notion d'invariance, que nous appelons stabilité, pour une requête, relativement à un élagage. L'idée sous-jacente est qu'une modification structurelle dans la zone élaguée d'un arbre n'a aucune incidence sur l'ensemble des nœuds sélectionnés par la requête dans la partie non élaguée.

3. Dans ce cas précis, ils sont en fait complètement annotés.

4.5.1 Stabilité des requêtes monadiques

Avant de définir formellement la stabilité d'une requête relativement à un élagage, nous illustrons cette notion par deux d'exemples. Dans cette optique, nous allons utiliser un *élagage régulier* (1) relativement au langage universel (ou $\top$-élagage, noté $\mathcal{P}_\top$), et (2) relativement au schéma D (ou D-élagage, noté $\mathcal{P}_D$), que nous décrivons ici de manière informelle [4]. Le premier élagage ne conserve que les nœuds annotés positivement et les chemins menant de la racine à ces nœuds, les sous-arbres supprimés sont remplacés par un unique symbole d'élagage $\top$. Le second fonctionne sur le même principe, mais diffère dans le sens où les sous-arbres supprimés sont remplacés par des états de l'automate (factorisé) déterministe qui représente D. Cet élagage conserve une information sur le type des sous-arbres supprimés.

Nous considérons ci-dessous deux exemples de requêtes monadiques, et un schéma D représenté par l'automate déterministe de la figure 56 (cf. p. 138).

1. La première requête, notée $\mathcal{Q}_1$, consiste à sélectionner tous les nœuds étiquetés par le symbole a dont le père est étiqueté par le symbole f. On écrit $a+$ au lieu de $(a, 1)$ (resp. $a-$ au lieu de $(a, 0)$) si le nœud étiqueté par a est annoté positivement (resp. négativement), et q au lieu de $(q, 0)$ s'il s'agit d'un symbole d'élagage. Considérons par exemple l'arbre $t = f(a, b, g(b))$, alors $t * \mathcal{Q}_1(t) = f-(a+, b-, g-(b-))$, et $t' * \beta' = \mathcal{P}_\top(t * \mathcal{Q}_1(t)) = f-(a+, \top, \top)$. On peut prendre $t_1 = f(a, f(a), a)$ et $t_2 = f(a, b, g(b, b))$ parmi les éléments de $c_F(t' * \beta')$, c'est-à-dire l'ensemble des F-complétions de $t' * \beta'$ (cf. sect. 4.4.2) — notez que le symbole $\top$ peut ici être remplacé par n'importe quel sous-arbre sur $\Sigma = \{f, g, a, b\}$ puisque $\mathcal{P}_\top$ élague relativement au langage universel.

   On voit assez facilement dans cet exemple que le premier fils de la racine, qui est l'unique nœud sélectionné par $\mathcal{Q}_1$ dans l'arbre t, sera sélectionné quelles que soient les F-complétions de $t * \mathcal{Q}_1(t)$, dont bien sûr t_1 et t_2. Cela s'explique par le fait que l'information utile pour sélectionner ce nœud (son père est étiqueté par f) est conservée par l'élagage $\mathcal{P}_\top$. On dira que $\mathcal{Q}_1$ est stable par $\mathcal{P}_\top$.

   Notons que $\mathcal{Q}_1$ sélectionne des nœuds supplémentaires dans l'arbre t_1 (tous les nœuds étiquetés par a sont sélectionnés), ce qui n'aura pas d'incidence sur notre définition de stabilité, car on s'intéressera seulement à la partie non élaguée de t pour la caractériser (ici, elle se limite à la racine et à son premier enfant).

4. Ces élagages seront détaillés plus loin, section 4.5.3.

2. La seconde requête, $\mathcal{Q}_2$, consiste à sélectionner tous les nœuds étiquetés par le symbole a dont le prochain frère est étiqueté par le symbole a ou le symbole b. En considérant le même arbre t que dans l'exemple précédent, on a $t * \mathcal{Q}_2(t) = f-(a+, b-, g-(b-))$, et donc le même arbre élagué que précédemment $\mathcal{P}_\top(t * \mathcal{Q}_2(t)) = f-(a+, \top, \top)$ ainsi que les mêmes complétions.

   Cette fois, la requête $\mathcal{Q}_2$ appliquée à l'arbre F-complété t_1 ne sélectionne pas le premier enfant de la racine étiqueté par a car son prochain frère n'est étiqueté ni par a, ni par b. L'information pertinente pour le sélectionner (l'étiquette de son prochain frère) a été perdue par l'élagage $\mathcal{P}_\top$. On dira que la requête $\mathcal{Q}_2$ n'est pas stable par $\mathcal{P}_\top$.

   En revanche, si l'on choisit l'élagage $\mathcal{P}_D$, alors on a l'arbre élagué $t' * \beta' = \mathcal{P}_D(t * \mathcal{Q}_2(t)) = f-(a+, q_0, q_1)$. Dans ce cas, l'arbre t_1 ne fait pas partie de l'ensemble $c_F(t' * \beta')$, au contraire de t_2 — F est ici l'automate déterministe de la figure 56 p. 138. La présence du symbole d'élagage q_0 impose en effet que toute F-complétion comporte à cette position soit une étiquette a, soit une étiquette b, ce qui est précisément la condition pour laquelle le premier fils de la racine est sélectionné. On dira cette fois que la requête $\mathcal{Q}_2$ est stable par $\mathcal{P}_D$.

Nous donnons ci-dessous une définition formelle de la notion de stabilité esquissée dans les deux exemples précédents.

Définition 4.20 (Stabilité)
Une requête monadique $\mathcal{Q}$ est *stable* (ou *invariante*) par un F-élagage $\mathcal{P}$ si et seulement si pour tout arbre $t_0 \in \mathcal{L}^u(F)$ avec $s' = \mathcal{P}(t_0 * \mathcal{Q}(t_0))$, on a :

$$\forall t_1 \in c_F(s') \quad \mathcal{Q}(t_1) =_{e^c(s')} \mathcal{Q}(t_0)$$ ◄

Le sens de cette définition est que pour tout arbre F-complété d'un arbre élagué, la requête sélectionne exactement les mêmes nœuds sur la partie non élaguée que pour l'arbre d'origine. Cela s'accorde avec l'idée que l'arbre élagué porte en son sein les informations nécessaires et suffisantes pour que la requête détermine les nœuds à annoter positivement.

Le lemme suivant montre que le langage des arbres annotés élagués d'une requête stable est pseudo-fonctionnel.

Lemme 4.21
Soient une requête monadique $\mathcal{Q}$, un F-élagage $\mathcal{P}$, et un langage d'arbres élagués $L \subseteq T^{u}_{\Delta\times\mathbb{B}}$ tel que :

$$L = \{\mathcal{P}(t * \mathcal{Q}(t)) \mid t \in \mathcal{L}^{u}(F)\}$$

Si $\mathcal{Q}$ est stable par $\mathcal{P}$ alors L est F-pseudo-fonctionnel.

Preuve
Supposons que L n'est pas F-pseudo-fonctionnel. Alors il existe $t'_1 * \beta'_1$ et $t'_2 * \beta'_2$ sur $\Delta \times \mathbb{B}$ non F-compatibles, avec $\nu \in \mathrm{nod}(t'_1 * \beta'_1) \cap \mathrm{nod}(t'_2 * \beta'_2)$ tel que $\beta'_1(\nu) \neq \beta'_2(\nu)$ (cf. déf. 4.9). Considérons $\beta'_1(\nu) = 1$ et $\beta'_2(\nu) = 0$; le cas $\beta'_1(\nu) = 0$ et $\beta'_2(\nu) = 1$ est symétrique. Par définition de L, il existe $t_1, t_2 \in \mathcal{L}^{u}(F)$ tels que $t'_1 * \beta'_1 = \mathcal{P}(t_1 * \mathcal{Q}(t_1))$ et $t'_2 * \beta'_2 = \mathcal{P}(t_2 * \mathcal{Q}(t_2))$. Il s'ensuit que $\nu \in \mathcal{Q}(t_1)$ et $\nu \notin \mathcal{Q}(t_2)$. Par ailleurs, il existe par définition $t_3 \in \mathcal{L}^{u}(F)$ pour lequel $t'_1 * \beta'_1$ et $t'_2 * \beta'_2$ sont tous deux consistants. On note que $t_3 \in c_F(t'_1 * \beta'_1)$ et $t_3 \in c_F(t'_2 * \beta'_2)$ par définition. On distingue deux cas.

1. Cas $\nu \in \mathcal{Q}(t_3)$. Par définition de stabilité, on doit vérifier $\mathcal{Q}(t_3) =_{e^{c}(t'_2 * \beta'_2)} \mathcal{Q}(t_2)$. Or $\nu \in e^{c}(t'_2 * \beta'_2)$ et $\nu \notin \mathcal{Q}(t_2)$.
2. Cas $\nu \notin \mathcal{Q}(t_3)$. Par définition de stabilité, on doit vérifier $\mathcal{Q}(t_3) =_{e^{c}(t'_1 * \beta'_1)} \mathcal{Q}(t_1)$. Or $\nu \in e^{c}(t'_1 * \beta'_1)$ et $\nu \in \mathcal{Q}(t_1)$.

Dans les deux cas, il y a contradiction. D'où L est F-pseudo-fonctionnel. □

Notons que l'implication inverse est fausse, comme le montre le contre-exemple suivant. Soit (Σ, ar) l'alphabet d'arité bornée tel que $ar(f) = 2$ et $ar(a) = ar(b) = 0$. Considérons la requête monadique $\mathcal{Q}$ sur Σ qui sélectionne le premier nœud feuille étiqueté par a. Cette requête n'est clairement pas stable par l'élagage $\mathcal{P}_{\top}$. Néanmoins, le langage des arbres annotés par $\mathcal{Q}$ élagués par $\mathcal{P}_{\top}$ (contenant par exemple les arbres $f-(\top, a+)$ et $f-(a+, \top)$) est pseudo-fonctionnel.

4.5.2 Résultat d'apprenabilité des requêtes stables par élagage

Dans cette section, nous nous intéressons à l'apprenabilité des requêtes stables par élagage, qui s'inscrit dans le modèle d'identification à la limite de la 4.3. Nous démontrons le résultat suivant.

Théorème 4.22
Soit $\mathcal{P}$ un F-élagage. La classe des requêtes stables par $\mathcal{P}$ représentées par des transducteurs de nœuds F-élagueurs déterministes est identifiable en temps et données polynomiaux à partir d'exemples partiellement annotés.

La technique de démonstration de ce théorème est similaire à celle employée pour prouver l'apprenabilité des requêtes consistantes avec le schéma (th. 4.8). En effet, nous construisons un échantillon caractéristique à partir de l'échantillon caractéristique pour l'algorithme $t\mathcal{RPNI}$. Toutefois, l'élagage impose de considérer celui-ci pour l'apprentissage à partir d'arbres élagués.

Preuve

Considérons l'algorithme d'apprentissage avec élagage $t\mathcal{RPNI}_{D}^{\mathcal{P}}$ de la section 4.4.4. Il s'agit d'une variante de $t\mathcal{RPNI}$ qui opère d'abord l'élagage des arbres de l'échantillon donné en entrée puis infère un automate pour les arbres élagués, similairement à l'algorithme de base. La preuve est analogue à celle du théorème 4.8, à savoir que nous montrons : (1) $t\mathcal{RPNI}_{D}^{\mathcal{P}}$ retourne un TSNé déterministe en temps polynomial dans la taille de l'échantillon ; (2) pour tout TSNé déterministe A représentant une requête stable par $\mathcal{P}$, il existe un échantillon caractéristique de cardinalité polynomiale dans la taille de A pour lequel $t\mathcal{RPNI}_{D}^{\mathcal{P}}$ retourne un TSNé déterministe équivalent.

1. La première opération réalisée par l'algorithme $t\mathcal{RPNI}_{D}^{\mathcal{P}}$ est l'élagage des arbres de l'échantillon d'arbres partiellement annotés $\widetilde{S}$. Cette opération s'effectue, par hypothèse sur la fonction d'élagage $\mathcal{P}$, en temps polynomial dans la taille des arbres, mettons $\rho_{\mathcal{P}}(|\widetilde{S}|)$. En second lieu, $t\mathcal{RPNI}_{D}^{\mathcal{P}}$ produit un automate initial en temps linéaire dans la taille de l'échantillon. Ensuite, l'algorithme opère un nombre de fusions au plus quadratique dans la taille de l'automate initial. Soit A' l'automate issu d'une fusion déterministe. À chacune d'entre elles, sont vérifiées : (1) la F-pseudo-fonctionnalité du langage de A', en temps polynomial $\mathcal{O}(|A'| \times |F| + |A'|^2)$ (cf. th. 4.19); (2) sa non-nullité, en temps linéaire ; (3) la compatibilité avec l'échantillon, en temps polynomial $\mathcal{O}(|\widetilde{S}| \times (|A'| + |F|))$ (voir sect. 4.4.4) ; (4) la D-consistance en temps polynomial $\mathcal{O}(|A'| \times |F|)$. Le temps total d'exécution de l'algorithme $t\mathcal{RPNI}_{D}^{\mathcal{P}}$ est donc polynomial, en $\mathcal{O}(|\widetilde{S}|^4 + |\widetilde{S}|^3 \times (|\widetilde{S}| \times |F|) + \rho_{\mathcal{P}}(|\widetilde{S}|))$. Enfin, le test de pseudo-fonctionnalité garantit que la sortie de l'algorithme est un TSNé déterministe.

2. Soit $\mathcal{Q}$ une requête stable par $\mathcal{P}$ représentée par un TSN F-élagueur déterministe A. Par définition, A est aussi un TSN (non élagueur) sur l'alphabet Δ ; il définit ainsi une requête monadique (sur Δ). Par conséquent, il existe un polynôme ρ_2 et un échantillon caractéristique $CS' \uplus \widehat{CS'}$ de cardinalité inférieure à $\rho_2(|A|)$ pour lequel, étant donné un échantillon contenant $CS' \uplus \widehat{CS'}$ (consistant avec A), l'algorithme $t\mathcal{RPNI}$ retourne un TSN déterministe (minimal) A' équivalant à A (cf. th. 4.4). Notons que les arbres de $CS' \uplus \widehat{CS'}$ sont des arbres élagués (c.-à-d. sur $\Delta \times \mathbb{B}$).

Nous allons définir, à partir de $CS' \uplus \widehat{CS'}$, un échantillon d'arbres complets $CS \uplus \widehat{CS}$ et montrer qu'il est caractéristique pour l'algorithme $t\mathcal{RPNI}_D^{\mathcal{P}}$. Nous définissons d'abord deux fonctions c_F^{min} et $c_F^{min_{\mathcal{P}}^{\mathcal{Q}}}$, qui permettent de choisir de façon unique un élément dans l'ensemble des F-complétions d'un arbre élagué :

$$\begin{aligned} c_F^{min}(t' * \beta') &= min(c_F(t' * \beta')) \\ c_F^{min_{\mathcal{P}}^{\mathcal{Q}}}(t' * \beta') &= min(\{t \in c_F(t' * \beta') \mid t' * \beta' = \mathcal{P}(t * \mathcal{Q}_A(t))\}) \end{aligned}$$

La fonction c_F^{min} sélectionne un plus petit sous-arbre complet parmi l'ensemble des complétions d'un arbre élagué, tandis que $c_F^{min_{\mathcal{P}}^{\mathcal{Q}}}$ en choisit un pour lequel, une fois annoté par la requête, lui appliquer l'élagage permet de retrouver l'arbre élagué d'origine. Un tel arbre complet existe nécessairement puisque tout arbre élagué provient par définition d'un arbre complet — autrement dit, pour tout $t' * \beta' \in \mathcal{P}(\mathcal{L}(\mathcal{Q}_A))$ il existe $t * \beta \in \mathcal{L}(\mathcal{Q}_A)$ tel que $t' * \beta' = \mathcal{P}(t * \beta)$. Nous définissons d'abord deux échantillons CS_1 et CS_2 comme suit :

$$\begin{cases} CS_1 = \{t * \mathcal{Q}_A(t) \in T^u_{\Sigma \times \mathbb{B}} \mid t' * \beta' \in CS' \wedge t = c_F^{min_{\mathcal{P}}^{\mathcal{Q}}}(t' * \beta')\} \\ CS_2 = \{t * \mathcal{Q}_A(t) \in T^u_{\Sigma \times \mathbb{B}} \mid t' * \beta' \in \widehat{CS'} \wedge t = c_F^{min}(t' * \beta')\} \end{cases}$$

Ainsi, chaque arbre élagué de l'échantillon $CS' \uplus \widehat{CS'}$ est associé à un unique arbre complet dans l'ensemble $CS_1 \cup CS_2$. Il nous faut encore distinguer, dans l'ensemble CS_2, les arbres qui ne possèdent aucune annotation positive de ceux qui en possèdent au moins une. Nous définissons donc l'échantillon d'arbres complets et annotés $CS \uplus \widehat{CS}$ comme suit :

$$\begin{cases} \widehat{CS} = \{t * \beta \in T^u_{\Sigma \times \mathbb{B}} \mid t * \beta \in CS_2 \wedge \forall \nu \in nod(\beta)\ (\beta(\nu) = 0)\} \\ CS = (CS_1 \cup CS_2) \setminus \widehat{CS} \end{cases}$$

Par construction, on a $CS \subseteq \mathcal{L}(\mathcal{Q}_A)$, et la cardinalité de $CS \uplus \widehat{CS}$ est inférieure ou égale à celle de $CS' \uplus \widehat{CS'}$, donc inférieure ou égale à $\rho_2(|A|)$. Notez également que $\mathcal{P}(CS) \supseteq CS'$.

Nous montrons maintenant que $CS \uplus \widehat{CS}$ est un échantillon caractéristique pour l'algorithme $t\mathcal{RPNI}_D^{\mathcal{P}}$. Pour cela, nous considérons un échantillon d'arbres (complètement) annotés $\widetilde{S}$ consistants avec A contenant $CS \uplus \widehat{CS}$. La première opération réalisée par $t\mathcal{RPNI}_D^{\mathcal{P}}$ consiste à élaguer tous les arbres non-nuls de $\widetilde{S}$ (c.-à-d. qui contiennent au moins une annotation positive), ce qui produit un échantillon

$S' = \{\mathcal{P}(t * \beta) \mid t * \beta \in \widetilde{S} \wedge \exists v \in \mathrm{nod}(\beta) \text{ tq. } \beta(v) = 1\}$. Nous vérifions ensuite que l'algorithme $t\mathcal{RPNI}_D^{\mathcal{P}}$ se comporte de façon identique à l'algorithme $t\mathcal{RPNI}$ avec l'échantillon $S' \uplus \widehat{CS'}$. Notons que par construction S' contient $\mathcal{P}(CS)$, qui contient lui-même CS', d'où $S' \uplus \widehat{CS'}$ est caractéristique pour $t\mathcal{RPNI}$. L'automate initial produit par $t\mathcal{RPNI}$ est syntaxiquement équivalent à l'automate initial produit par $t\mathcal{RPNI}_D^{\mathcal{P}}$; les deux algorithmes prennent en effet en compte les exemples non-nuls, c'est-à-dire l'ensemble S', et seulement eux pour le construire. On montre dès lors que toute fusion déterministe rejetée par $t\mathcal{RPNI}$ est nécessairement rejetée par $t\mathcal{RPNI}_D^{\mathcal{P}}$, et réciproquement.

"$\Rightarrow$" Une fusion est rejetée par l'algorithme $t\mathcal{RPNI}$ soit parce que l'automate A' qui en résulte n'est pas un TSN (son langage n'est pas fonctionnel ou n'est pas non-nul), soit parce qu'il n'est pas compatible avec un arbre de $\widehat{CS'}$.

a) *Cas A' n'est pas un TSN sur Δ.* Supposons que $t\mathcal{RPNI}_D^{\mathcal{P}}$ accepte la fusion. Alors A' est un TSN F-élagueur sur Σ. Donc, par définition, A' est aussi un TSN élagueur sur Δ, ce qui est impossible.

b) *Cas A' n'est pas compatible avec un arbre de $\widehat{CS'}$.* Alors il existe $t' * \beta_1' \in \widehat{CS'}$ et $t' * \beta_2' \in \mathcal{L}^u(A')$ tels que $\beta_1' \neq \beta_2'$ (cf. discussion autour de $t\mathcal{RPNI}$ p. 119). Dit autrement, il existe un nœud $v \in \mathrm{dom}(t')$ tel que $\beta_1'(v) \neq \beta_2'(v)$ avec $\beta_1'(v) = 0$ et $\beta_2'(v) = 1$. On pose $t = c_F^{min}(t' * \beta_1')$ et on distingue les deux cas suivants.

i. *Cas* $t * \mathcal{Q}_A(t) \in CS$. Dans ce cas, $\mathcal{P}(t * \mathcal{Q}_A(t))$ et $t' * \beta_1'$ ne sont pas F-compatibles. Or cette condition est nécessaire pour la F-pseudo-fonctionnalité que vérifie $t\mathcal{RPNI}_D^{\mathcal{P}}$.

ii. *Cas* $t * \mathcal{Q}_A(t) \in \widehat{CS}$. Cela implique que $\mathcal{Q}_{A'}(t) = \emptyset$. Or, d'un autre côté, $t' * \beta_2' \in \mathcal{L}^u(A')$ avec $\beta_2'(v) = 1$ entraîne que $v \in \mathcal{Q}_{A'}(t)$ (on rappelle que par déf. $\mathcal{Q}_{A'}(t) = \{v \in \mathrm{nod}(t) \mid \exists t' * \beta' \in \mathcal{L}^u(A') \text{ consistant avec } t \wedge \beta'(v) = 1\}$). Cette contradiction est capturée par $t\mathcal{RPNI}_D^{\mathcal{P}}$ *via* le test de compatibilité de A' avec $\widetilde{S}$.

Dans les deux cas, $t\mathcal{RPNI}_D^{\mathcal{P}}$ rejette la fusion.

"$\Leftarrow$" Dans l'autre sens, une fusion déterministe peut être refusée par $t\mathcal{RPNI}_D^{\mathcal{P}}$ si l'une des conditions suivantes n'est pas respectée : pseudo-fonctionnalité, non-nullité, compatibilité avec l'échantillon ou D-consistance. Nous basons la suite de la preuve sur le fait que la compatibilité, la stabilité et la consistance sont monotones dans le sens suivant : si une propriété P est fausse pour un

langage L, alors P est fausse pour tout surlangage $L' \supseteq L$. L'algorithme $t\mathcal{RPNI}$ procède par généralisations successives du langage de l'automate hypothèse. Par conséquent, une propriété fausse suite à une fusion acceptée restera fausse à la fin de l'exécution de l'algorithme. Il nous reste à énumérer les conditions.

a) *Cas incompatibilité.* La fusion est rejetée par $t\mathcal{RPNI}_D^{\mathcal{P}}$ car il existe $t * \beta \in \mathcal{L}^u(\mathcal{Q}_{A'})$ (où A' est l'automate issu de la fusion) et $\nu \in \mathrm{nod}(\beta)$ tel que $\beta(\nu) = 1$, mais $\nu \notin \mathcal{Q}_A(t)$. Autrement dit, la propriété $\mathcal{Q}_{A'}(t) \cap d_- = \emptyset$ pour tout $(t, d_+, d_-) \in \widetilde{S}$ est fausse. Supposons que $t\mathcal{RPNI}$ accepte la fusion. Alors, par monotonie, la propriété reste fausse à la fin de l'exécution de l'algorithme. Or, l'automate inféré, disons A'', doit être équivalent à A, donc par définition un TSN F-élagueur. Mais la condition de F-pseudo-fonctionnalité est contredite par $t * \beta \in \mathcal{L}(\mathcal{Q}_{A''})$, $\beta(\nu) = 1$ et $\nu \notin \mathcal{Q}_A(t)$. De ce fait, $t\mathcal{RPNI}$ refuse aussi la fusion.

b) *Cas pseudo-fonctionnalité.* Supposons que $t\mathcal{RPNI}$ accepte la fusion. Alors le langage de l'automate A' issu de la fusion n'est pas pseudo-fonctionnel, ce qui implique, d'après le lemme 4.21, que $\mathcal{Q}_{A'}$ n'est pas stable par $\mathcal{P}$. Par monotonie, la requête finalement inférée ne sera pas stable par l'élagage $\mathcal{P}$. Ceci contredit l'hypothèse ; il s'ensuit que $t\mathcal{RPNI}$ rejette la fusion.

c) *Cas non-nullité.* Supposons que $t\mathcal{RPNI}$ accepte la fusion. Alors le langage reconnu par l'automate issu de la fusion n'est pas non-nul. Une telle situation est impossible car $t\mathcal{RPNI}$ vérifie directement la non-nullité.

d) *Cas* D*-consistance*. Similairement aux deux premiers points, en supposant que $t\mathcal{RPNI}$ accepte la fusion, la requête finalement inférée ne serait pas D-consistante par monotonie. Or, la requête définie par A est nécessairement D-consistante. En effet, par définition, on ne peut pas compléter des arbres qui ne sont pas dans $\mathcal{L}(D)$. La fusion est donc aussi rejetée par $t\mathcal{RPNI}$.

En conséquence, les algorithmes $t\mathcal{RPNI}_D^{\mathcal{P}}$ et $t\mathcal{RPNI}$ retournent un même automate A'' ; celui-ci représente par définition une requête $\mathcal{Q}_{A''}$ stable par $\mathcal{P}$ équivalente à $\mathcal{Q}_A$. □

Notons que contrairement aux algorithmes sans élagage présentés dans les sections 4.2 et 4.3, la sortie de $t\mathcal{RPNI}_D^{\mathcal{P}}$ peut ne pas être compatible avec l'échantillon d'entrée, comme le montre l'exemple introductif p. 148. Cela justifie en partie le choix d'un modèle d'apprentissage moins contraignant pour l'induction de requêtes par rapport à l'inférence

de langages d'arbres. Du point de vue théorique, le point fondamental dans le modèle d'apprentissage est la complétude, à savoir que l'algorithme doit garantir l'identification dès lors qu'il est en présence d'un échantillon caractéristique. Si ce n'est pas le cas, il ne dispose pas de suffisamment d'information pour répondre de façon satisfaisante, et donc sa sortie a peu d'importance. L'interactivité est un moyen d'acquérir de l'information supplémentaire pour les algorithmes d'apprentissage. Carme *et al.* (2007), par exemple, proposent pour l'induction de requêtes une solution élégante basée sur le modèle d'apprentissage actif de Angluin (1987, 1988).

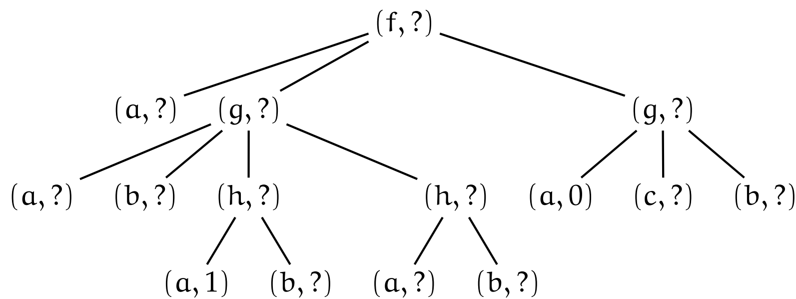


Figure 63. *Un arbre partiellement annoté.*

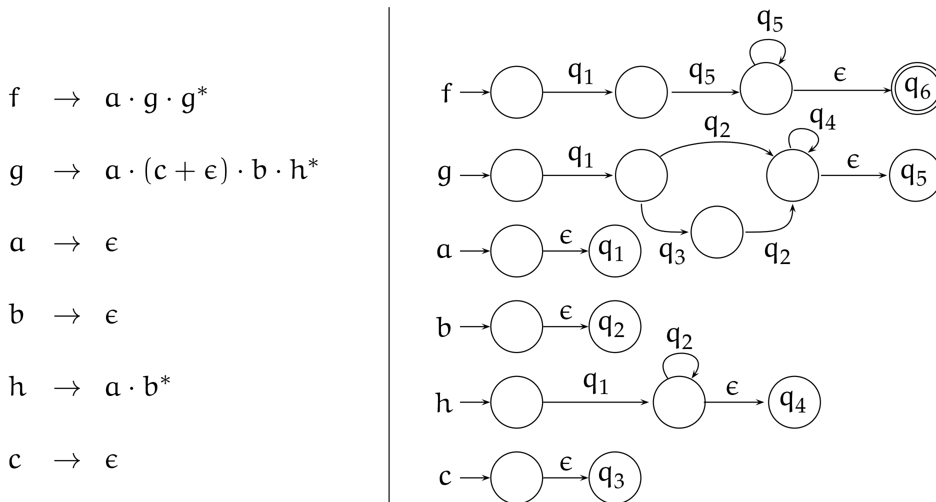


Figure 64. *Une DTD* D *et sa représentation par un automate factorisé déterministe* F.

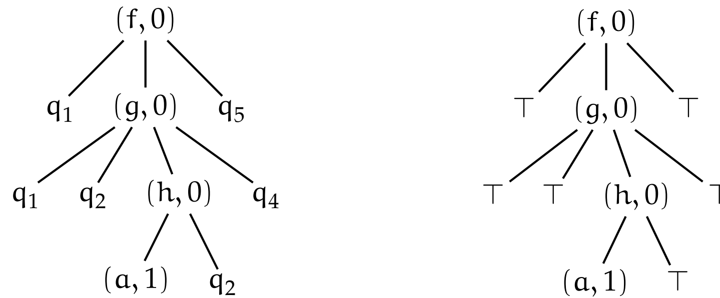


Figure 65. *Élagage régulier de l'arbre complet de la fig. 63. À gauche, en utilisant la DTD de la fig. 64 (D-élagage) ; à droite, en utilisant le langage universel ($\top$-élagage). L'élagage régulier doit son nom au fait qu'il utilise un automate (donc un langage régulier) pour opérer sur les arbres.*

4.5.3 Élagages utilisés en pratique

Dans cette section, nous présentons l'ensemble des élagages que nous avons examinés lors de nos expériences (cf. chapitre suivant), puis nous en établissons une classification relativement à la notion de stabilité.

Pour illustrer les élagages, nous considérons l'arbre partiellement annoté sur Σ montré à la figure 63, avec $\Sigma = \{f, g, h, a, b, c\}$, et la DTD D sur Σ de la figure 64, avec F l'automate factorisé déterministe représentant D. On pose $\Gamma = \mathtt{sta}_2(F) = \{q_1, q_2, q_3, q_4, q_5, q_6\}$.

Élagage identité

L'*élagage identité*, noté $\mathcal{P}_{\mathtt{id}}$, est le plus simple de tous les élagages puisqu'il consiste à ne rien élaguer (tous les nœuds dépourvus d'annotation sont considérés comme annotés négativement). Dire qu'un arbre n'est pas élagué est donc équivalent à dire qu'il est élagué avec l'élagage identité. Cet élagage présuppose que les exemples sont tous complètement annotés. Notons que toute requête monadique est stable par cet élagage.

Élagage régulier

L'*élagage régulier* a été évoqué à la section 4.5.1. Il consiste à remplacer les sous-arbres supprimés par des symboles d'élagage $q \in \Gamma$ annotés négativement, correspondant aux états dans lesquels ils sont évalués par l'automate factorisé déterministe représentant le schéma utilisé pour l'élagage. Ne subsistent que les nœuds annotés positivement et les

Paramètre: D, un schéma représenté par un automate factorisé déterministe F
Entrée: $(f(t_1, \dots, t_n), d_+, d_-)$, un arbre partiellement annoté
Sortie: $t' * \beta'$, un arbre élagué

1: *// appel récursif sur les descendants*
2: **for all** t_i **in** $t_1, \dots, t_n$ **do**
3: $\quad t'_i * \beta'_i \leftarrow \mathcal{P}_D(t_i, \mathtt{tail}(d_+, i), \mathtt{tail}(d_-, i))$
4: **if** $\epsilon \in d_+$ **then**
5: $\quad$ *// cas où la racine est annotée positivement*
6: $\quad$ **return** $(f, 1)(t'_1 * \beta_1, \dots, t'_n * \beta'_n)$
7: **else if** $\exists t'_i * \beta'_i \in \{t'_1 * \beta_1, \dots, t'_n * \beta'_n\}$ tq. $t'_i(\epsilon) \in \Sigma \times \mathbb{B}$ **then**
8: $\quad$ *// cas où un descendant de la racine est annotée positivement*
9: $\quad$ **return** $(f, 0)(t'_1 * \beta_1, \dots, t'_n * \beta'_n)$
10: **else**
11: $\quad$ *// cas où la racine n'est pas annotée positivement ni aucun de ses descendants*
12: $\quad$ **return** $(eval^u_{b(F)}(f(t_1, \dots, t_n)), 0)$

Figure 66. *Fonction* $\mathcal{P}_D$.

nœuds annotés négativement qui se situent sur un chemin menant de la racine à un nœud annoté positivement. On distingue le D-élagage et le $\top$-élagage, tous deux illustrés à la figure 65 et explicités ci-dessous.

D-élagage L'*élagage régulier relatif au schéma* D, ou D-*élagage*, est noté $\mathcal{P}_D$. La figure 66 présente cette fonction. Celle-ci opère récursivement en élaguant d'abord les sous-arbres afin de déterminer l'action à réaliser au niveau du nœud racine, à savoir conserver l'annotation si elle est positive ou si un descendant possède une annotation positive, ou bien remplacer l'arbre par son état d'évaluation par l'automate factorisé déterministe F qui représente D. L'appel récursif utilise une fonction $\mathtt{tail}$ qui permet de définir correctement les annotations positives et négatives des sous-arbres partiellement annotés :

$$\mathtt{tail}(d, i) = \{v \in \mathbb{N}^* \mid i \cdot v \in d\}$$

Le D-élagage d'un arbre partiellement annoté (t, d_+, d_-) sur Σ peut être réalisé en temps $\mathcal{O}(|t| \times |\Sigma| \times |D|)$, ce qui correspond à son évaluation par l'automate factorisé déterministe qui représente D.
L'élagage régulier relatif à D peut être utilisé, lors de l'inférence, en conjonction avec la D-consistance, à la seule condition que le schéma employé pour vérifier la consistance soit le même que celui utilisé pour l'élagage.

$\top$-élagage L'*élagage régulier relatif au langage universel*, ou $\top$-*élagage*, consiste à remplacer les sous-arbres supprimés par un unique symbole d'élagage $\top$ annoté négativement. On le note $\mathcal{P}_\top$.

Le $\top$-élagage est un cas particulier du D-élagage, où le schéma utilisé pour élaguer les arbres définit le langage universel. Il est facile de construire un automate factorisé déterministe reconnaissant le langage universel. Cet automate est constitué de deux états $\top$ (final) et $\top'$, de règles constantes $\alpha \to \top'$ pour tout α dans l'alphabet Σ, d'une seule ϵ-règle $\top' \overset{\epsilon}{\to} \top$, et d'une seule règle binaire $\top'@\top \to \top'$.

Vérifier la consistance avec le schéma en conjonction avec le $\top$-élagage n'a aucun sens car cela revient à tester l'inclusion dans le langage universel. En revanche, il est possible de combiner le $\top$-élagage avec l'heuristique des fusions horizontales.

Élagage étendu

L'*élagage étendu* étend l'élagage régulier en conservant les frères des nœuds qui se situent sur le chemin de la racine à un nœud annoté positivement. Les sous-arbres supprimés sont remplacés de façon similaire au cas régulier. Nous distinguons également les élagages étendus relativement au schéma (D-*élagage étendu*) et relativement au langage universel ($\top$-*élagage étendu*). Un exemple de ces deux élagages est donné figure 67. On les note respectivement $X\mathcal{P}_D$ et $X\mathcal{P}_\top$. Nous ne détaillons pas ici l'algorithme qui réalise l'élagage étendu.

Vérifier la consistance avec le schéma n'a de sens que pour le D-élagage étendu. L'heuristique des fusions horizontales, par contre, peut être utilisée en conjonction de l'un ou de l'autre.

Classification des élagages

Dans cette section, nous établissons une taxonomie des divers élagages présentés au-dessus, du plus "brutal" — celui qui conserve le moins d'informations — au moins brutal — celui qui en conserve le plus.

La compréhension de la classification repose sur la notion de F-complétion (cf. sect. 4.4.2). L'idée sous-jacente est qu'un élagage $\mathcal{P}_1$ conserve plus d'informations qu'un autre élagage $\mathcal{P}_2$ si, pour tout arbre annoté, l'ensemble des complétions pour cet arbre élagué

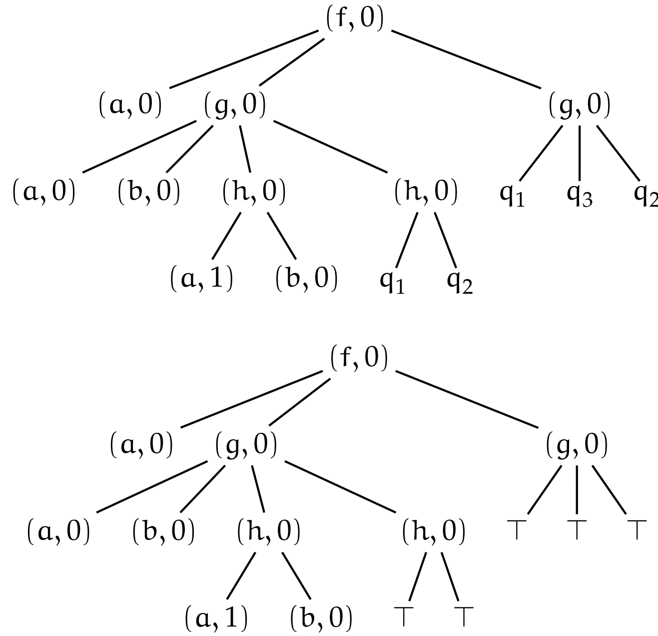


Figure 67. *Élagage étendu de l'arbre partiellement annoté de la fig. 63. En haut, en utilisant la DTD de la fig. 64 ; en bas, en utilisant le langage universel.*

par $\mathcal{P}_1$ est inclus dans l'ensemble des complétions du même arbre élagué par $\mathcal{P}_2$. La définition suivante traduit cette notion d'ordre partiel entre les élagages.

Définition 4.23

Soient $\mathcal{P}_1$ un F_1-élagage et $\mathcal{P}_2$ un F_2-élagage, $\mathcal{P}_1$ est *plus conservatif* que $\mathcal{P}_2$ si, pour tout arbre (complètement) annoté $t * \beta \in T^u_{\Sigma\times\mathbb{B}}$, on a :

$$c_{F_1}(\mathcal{P}_1(t * \beta)) \subseteq c_{F_2}(\mathcal{P}_2(t * \beta))$$ ◀

Intuitivement, plus un élagage est conservatif, plus la classe de requêtes qui peut être identifiée en l'utilisant dans l'algorithme d'inférence est expressive.

Proposition 4.24

Soient $\mathcal{Q}$ une requête monadique, et $\mathcal{P}_1$ et $\mathcal{P}_2$ deux élagages. Si $\mathcal{P}_1$ est plus conservatif que $\mathcal{P}_2$ et $\mathcal{Q}$ est stable par $\mathcal{P}_2$, alors $\mathcal{Q}$ est stable par $\mathcal{P}_1$.

Preuve

Découle des définitions 4.20 et 4.23. □

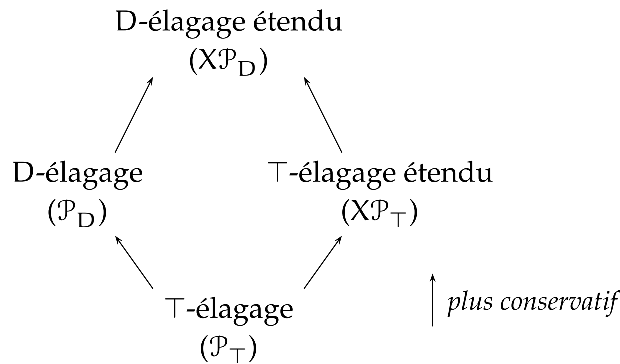


Figure 68. *Classification des élagages.*

La figure 68 présente une classification des élagages utilisés en pratique définis plus haut, relativement à leur potentiel conservatif. Parmi ces élagages, le moins conservatif est évidemment le $\top$-élagage. Il s'agit de celui qui conserve le minimum d'informations, à savoir les nœuds annotés positivement, les chemins qui mènent à eux depuis la racine, ainsi que leur position par rapport à leurs frères respectifs.

Ensuite, il est assez facile de voir que le D-élagage et le $\top$-élagage étendu restreignent le nombre de complétions par rapport au $\top$-élagage. Le premier parce qu'il utilise un alphabet d'élagage plus grand (les états de l'automate du schéma), le second parce qu'il conserve des nœuds étiquetés supplémentaires. En revanche, il n'est pas possible de comparer ces deux élagages dans le cadre défini ici, ce qu'illustre la figure 69. Dans cet exemple, nous exhibons pour un même arbre partiellement annoté un élément de l'ensemble des complétions pour le D-élagage qui n'appartient pas à l'ensemble des complétions pour le $\top$-élagage étendu, et inversement.

Le D-élagage étendu est plus conservatif que tous les élagages précédents. Il est aisé de vérifier que, pour tout arbre annoté, l'ensemble des complétions de cet arbre élagué par D-élagage étendu est forcément inclus dans l'ensemble des complétions du même arbre élagué avec l'un des trois autres élagages.

Au regard de la proposition 4.24, et étant donnée une requête monadique $\mathcal{Q}$, on observe les relations suivantes :

1. si $\mathcal{Q}$ est stable par $\mathcal{P}_\top$, alors $\mathcal{Q}$ est stable par $\mathcal{P}_D$, $X\mathcal{P}_\top$ et $X\mathcal{P}_D$;
2. si $\mathcal{Q}$ est stable par $\mathcal{P}_D$ ou $X\mathcal{P}_\top$, alors $\mathcal{Q}$ est stable par $X\mathcal{P}_D$.

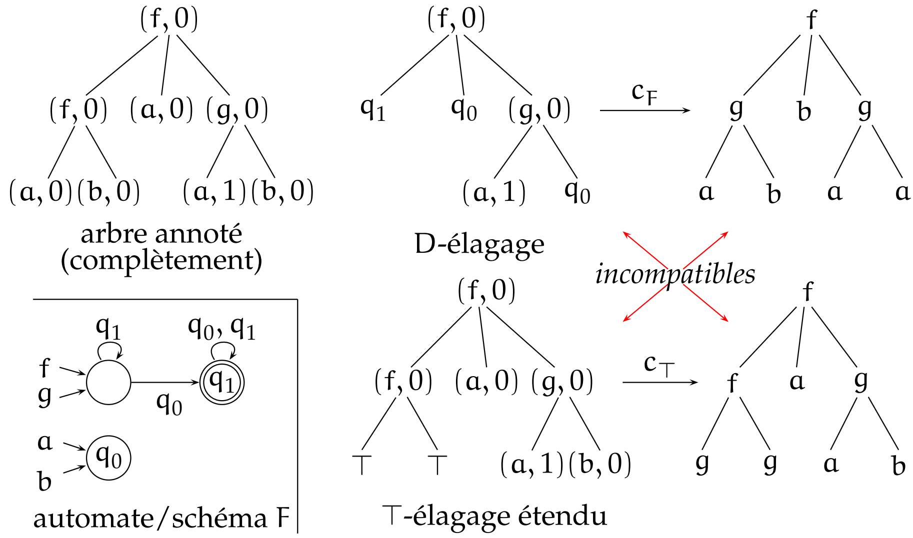


Figure 69. *Exemple de situation dans laquelle les* D*-élagage et* ⊤*-élagage étendu sont incomparables.*

Notons enfin, pour conclure cette classification, que l'élagage identité (qui n'apparaît pas dans la figure 68) est par définition le plus conservatif de tous les élagages.

5 Évaluation expérimentale

Ce chapitre est dédié à l'évaluation expérimentale des algorithmes d'induction de requêtes guidée par schéma présentés au chapitre précédent. Les expériences montrent que les connaissances apportées par le schéma permettent d'améliorer nettement la qualité de l'apprentissage. Elles confirment le rôle primordial de l'élagage et révèlent l'intérêt de la consistance. L'évaluation porte à la fois sur des jeux de données HTML et XML. Le protocole d'expérimentation consiste à simuler le comportement d'un utilisateur lors de la définition d'une nouvelle requête. Pour évaluer la qualité de l'apprentissage, nous employons une méthode de validation croisée qui s'apparente à un sous-échantillonage aléatoire répété, et nous mesurons des valeurs classiques utilisées dans les systèmes d'extraction d'information, principalement la F-mesure.

Sommaire

5.1 Introduction **167**
5.2 Paramètres d'expérimentation **168**
5.3 Jeux de données **169**
5.3.1 XML 169
5.3.2 HTML 177
5.3.3 Récapitulatif 178
5.4 Protocole d'expérimentation **179**
5.5 Résultats et analyse **183**
5.5.1 Effets de la vérification de la consistance avec le schéma 184
5.5.2 Apports de l'élagage sans vérifier la consistance avec le schéma . 187
5.5.3 Combinaison de l'élagage et de la consistance avec le schéma . . 191
5.5.4 Remarques conclusives 194

5.1 Introduction

Plusieurs systèmes d'induction de requêtes (ou *wrappers*) pour des tâches d'extraction d'information ont été mis au point et testés par le passé (p. ex. Muslea *et al.*, 2001; Chidlovskii, 2001; Cohen *et al.*, 2002). Des expériences, menées par Kosala *et al.* (2003), Carme *et al.* (2007) et Raeymaekers *et al.* (2008) notamment, ont montré tout l'intérêt de l'inférence d'automates d'arbres par rapport aux techniques basées sur les chaînes. Cependant, les systèmes d'apprentissage présentés dans tous ces travaux ont été développés et optimisés pour des tâches d'extraction d'information sur des collections de pages Web, donc uniquement des documents HTML. L'induction de requêtes dans des documents XML n'a pas été traitée au cours de ces diverses expériences. De plus, aucun des précédents systèmes n'utilise de l'information fournie par la DTD de HTML.

Ainsi nous intéressons-nous, dans le présent chapitre, à de nouvelles expériences portant sur l'induction de requêtes dans un ensemble de documents XML, au-delà du simple langage HTML — sans l'exclure, étant donné sa prédominance dans les applications d'extraction d'information. En particulier, nous évaluons les algorithmes d'inférence de langages d'arbres guidée par un schéma présentés au chapitre précédent.

Nous envisageons les deux utilisations du schéma, vérification de la consistance et élagage enrichi, comme des heuristiques génériques pour l'induction de requêtes, et qui peuvent être appliquées à toute collection de documents semi-structurés dès lors que ceux-ci sont valides par rapport au schéma. Nous nous limitons, dans nos expériences, à des schémas représentés par des DTD au sens W3C (Bray *et al.*, 2008). Notons que si aucune DTD n'est disponible, il est possible d'en inférer une approximation à partir d'un ensemble fini de documents XML (Papakonstantinou et Vianu, 2000; Bex *et al.*, 2008) ; nous n'explorons pas cette possibilité dans la suite.

Notre objectif est de mesurer l'apport de la consistance et de l'élagage relativement à un schéma dans le cadre de l'induction de requêtes ; ces deux heuristiques peuvent être employées de façon indépendante ou en association l'une avec l'autre. En particulier, nous évaluons, d'une part, dans quelle mesure la vérification de la consistance permet d'empêcher de mauvaises généralisations, et d'autre part, l'adéquation de l'élagage en fonction de la requête cible au regard de la propriété de stabilité. C'est pourquoi les requêtes que nous considérons dans les expériences sont toutes consistantes par rapport à un schéma donné, et la plupart stables par un ou deux élagages.

Tableau 2. *Heuristiques implémentées et algorithmes d'apprentissage afférents.*

	Pas de consistance	D-consistance inclusion	D-consistance typage	Fusions horizontales
Pas d'élagage	$\mathtt{t}\mathcal{RPNI}$	$\mathtt{t}\mathcal{RPNI}_{\mathrm{D}}$	$\mathtt{static}\text{-}\mathtt{t}\mathcal{RPNI}_{\mathrm{D}}$	–
D-élagage $\mathcal{P}$	–	$\mathtt{t}\mathcal{RPNI}_{\mathrm{D}}^{\mathcal{P}}$	–	–

L'évaluation expérimentale porte sur des jeux de données XML et HTML. La comparaison avec d'autres systèmes d'apprentissage est délicate pour au moins deux raisons : d'une part, aucun n'a été testé avec des jeux de données XML, et d'autre part, les documents utilisés pour évaluer ces systèmes ne sont généralement pas valides relativement à une DTD de HTML. Toutefois, nos expériences montrent que notre système d'apprentissage est compétitif sur des tâches HTML standard ; la qualité d'apprentissage est en effet comparable à celle de l'outil développé et testé par Raeymaekers (2008), qui s'est lui-même comparé exhaustivement avec les systèmes de la littérature. Notons que ce dernier est optimisé avec une technique proche de l'élagage qui n'a pas été formalisée.

La suite du chapitre s'organise comme suit. La seconde section donne les paramètres d'expérimentation. La troisième section détaille les différents jeux de données XML et HTML utilisés dans les expériences. La quatrième section décrit le protocole expérimental retenu et précise les mesures permettant d'estimer la qualité de l'apprentissage. Enfin, la cinquième et dernière section expose et analyse les résultats expérimentaux.

5.2 Paramètres d'expérimentation

Nous avons développé un système d'apprentissage entièrement nouveau. Les principaux algorithmes ont été codés en Objective Caml [1], un langage fonctionnel fortement typé dont les programmes compilés en code natif s'exécutent très rapidement. La gestion des expériences (prétraitements, présentation des données, collection des résultats, etc.) a été réalisée essentiellement avec des scripts Bash et Python.

L'algorithme d'apprentissage implémenté est paramétré de telle sorte qu'il permette différentes combinaisons d'heuristiques. La plupart sont directement liées aux algorithmes théoriques discutés au chapitre précédent. Le tableau 2 donne un récapitulatif.

1. `http://caml.inria.fr/ocaml/index.fr.html`.

Certaines combinaisons d'heuristiques testées n'ont pas de support théorique. En particulier, un D-élagage $\mathcal{P}$ peut être utilisé sans vérifier la consistance avec le schéma D, ce qui conduit à un algorithme incomplet ; en outre, les requêtes ainsi inférées peuvent être définies en dehors du domaine D. Toutefois, il ne s'agit pas d'un réel problème. En pratique, en effet, on ne peut pas compter sur la présence d'un échantillon caractéristique pour apprendre. De plus, les requêtes inférées ne sont jamais appliquées à des arbres hors-domaine.

5.3 Jeux de données

Un *jeu de données* est un ensemble fini de documents XML ou HTML, accompagné d'une DTD à laquelle tous les documents se conforment, ainsi que d'un *document compagnon* qui décrit, pour une requête considérée, les annotations positives — autrement dit les nœuds extraits — de l'ensemble d'arbres XML ou HTML. On utilise aussi un document compagnon, lors des expériences, pour définir l'échantillon d'arbres complètement et/ou partiellement annotés.

Dans la suite, nous présentons les jeux de données XML et HTML que nous avons utilisés pour nos expérimentations. Différentes caractéristiques quantitatives de ces jeux de données — proportion de documents avec au moins une annotation positive (c.-à-d. d'arbres non-nuls), nombre moyen d'annotations par document annoté, taille moyenne des documents (en Ko) et des arbres XML (nombre de nœuds) — sont rassemblées dans le tableau 3 p. 179. Notez que nos jeux de données sont disponibles en ligne à l'adresse `http://www.grappa.univ-lille3.fr/~champavere/Recherche/datasets/`.

5.3.1 XML

Une base de données XML n'est généralement pas organisée comme une collection de documents XML à proprement parler, mais est plutôt constituée d'un unique "gros" document XML qui contient toutes les données. Néanmoins, du fait de sa structure arborescente régulière, il est généralement possible de fractionner la base de données en un ensemble de sous-arbres indépendants. Ainsi, nous considérons toujours de telles collections d'arbres XML pour élaborer des jeux de données.

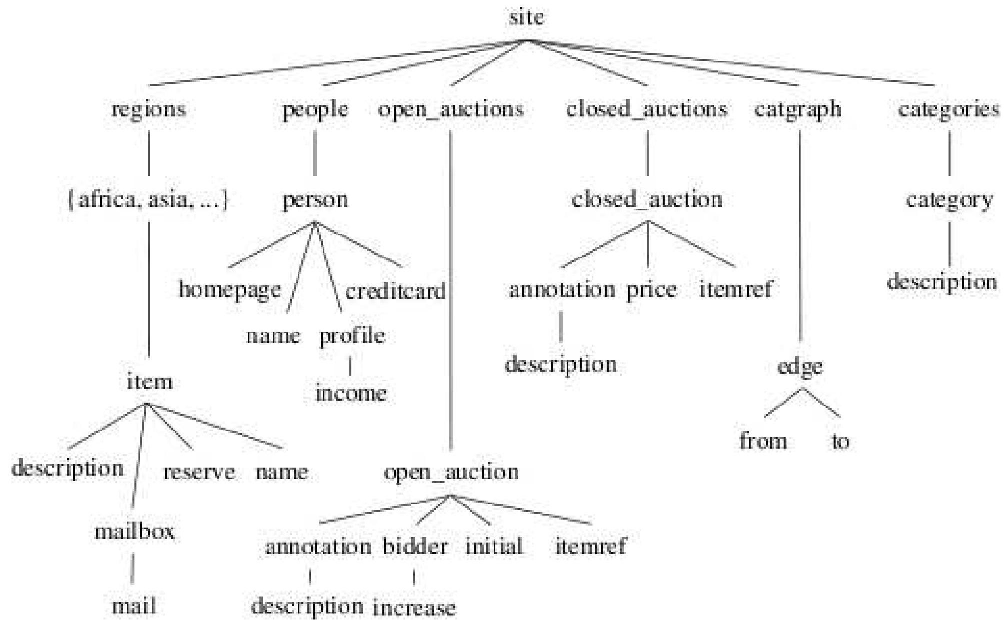


Figure 70. *Hiérarchie (partielle) des balises formant un document XMark (illustration tirée de Schmidt et al. (2002, 2001)).*

Pour nos expérimentations, nous avons utilisé principalement deux jeux de données XML : XMark et Mondial.

XMark XMark (Schmidt *et al.*, 2002, 2001, voir aussi `http://www.ins.cwi.nl/projects/xmark/`) est un projet de *benchmark* de référence dans la communauté des bases de données XML.

Un document XMark représente une base de données où est stockée un ensemble d'enchères. Il contient diverses informations comme les articles (`item`) soumis aux enchères, les personnes participant aux enchères (`person`), les enchères en cours (`open_auction`) ou terminées (`closed_auction`), etc. Un aperçu de l'organisation générale d'un document XML XMark est proposé figure 70 (une description plus détaillée est donnée par Schmidt *et al.* (2001)) ; la figure 71 présente des extraits d'un document XMark.

L'utilisation de XMark présente plusieurs intérêts pour nous. D'abord, XMark propose une DTD "complexe" [2] qui définit un ensemble d'arbres de structures variées, avec des récursions aussi bien horizontales que verticales. Cette DTD nous paraît beaucoup

2. Celle-ci ne peut être reproduite ici en raison de sa taille ; le lecteur la retrouvera à l'annexe A.2, ou bien en ligne à l'adresse `http://www.ins.cwi.nl/projects/xmark/Assets/auction.dtd`.

```
<site>
  <regions>...</regions>
  ...
  <people>
    <person id="person105">
      <name>Evangeline Emery</name>
      <emailaddress>mailto:Emery@fernuni-hagen.de</emailaddress>
      ...
      <homepage>http://www.fernuni-hagen.de/~Emery</homepage>
      ...
      <profile income="50735.15">
        ...
        <gender>male</gender>
        <business>No</business>
        <age>40</age>
      </profile>
      ...
    </person>
    ...
  </people>
  <open_auctions>
    <open_auction id="open_auction10">
      <initial>120.99</initial>
      <reserve>306.45</reserve>
      <bidder>
        ...
        <increase>15.00</increase>
      </bidder>
      ...
      <itemref item="item16"/>
      ...
    </open_auction>
    ...
  </open_auctions>
  <closed_auctions>
    ...
    <closed_auction>
      ...
      <annotation><description>...</description></annotation>
      ...
    <closed_auction>
  </closed_auctions>
</site>
```

Figure 71. *Extraits d'un document XMark.*

plus intéressante que celle de HTML dans notre cadre, dans le sens où la structure est prépondérante dans les documents XMark, par rapport aux pages Web où ce sont les données textuelles qui priment. Ensuite, les auteurs de XMark ont proposé un jeu de transformations de documents XML destiné à tester les processeurs XQuery. Celles-ci sont basées sur des requêtes XPath dont nous pouvons nous inspirer pour établir nos jeux de données. Enfin, dans le même ordre d'idées, Franceschet (2005) a élaboré XPathMark, un ensemble de requêtes XPath pour interroger une base de données XMark. Bien que définies dans le but de tester les performances des évaluateurs de requêtes XPath ou des moteurs XQuery, les requêtes considérées dans ces différents travaux correspondent à des requêtes "réalistes" que des utilisateurs (non experts, donc ne connaissant pas XPath) pourraient souhaiter inférer. Rappelons toutefois que nous nous limitons aux requêtes qui portent sur la structure des documents (éléments) ; nous excluons les attributs et les données textuelles.

Nous avons utilisé le générateur de documents XML fourni par les auteurs de XMark afin de produire automatiquement des jeux de données pour nos expériences. Plus précisément, un jeu de données XMark est obtenu en générant dans un premier temps un "gros" fichier XML contenant une base de données d'enchères. Puis, ce gros document est découpé, grâce à un ensemble de scripts que nous avons mis au point, en un jeu de documents XMark (c.-à.-d. conformes à la DTD de XMark) plus petits sur la base d'enchères en cours et/ou terminées choisies aléatoirement. Cela donne au final un jeu de données constitué de fichiers XML dont le nombre d'éléments open_auction et closed_auction est le même, et dont le nombre d'éléments item, person, categories, etc. — reconstitués grâce aux références contenues dans les sous-arbres de open_auctions et closed_auctions — est variable ; un jeu de données ainsi généré présente donc à la fois une dimension homogène et une dimension hétérogène. L'utilisation d'un tel script est nécessaire en raison d'une limitation du générateur de XMark (entre autres, la structure des documents n'est pas générée aléatoirement).

En outre, notre script de génération de jeu de données XMark permet l'utilisation d'un facteur pour la taille attendue des documents générés. Par exemple, le facteur 1 permet de générer des arbres XML dont la taille (en nombre de nœuds) est comparable en moyenne à celles des arbres XML du jeu de données HTML *Yahoo* ; les documents XMark générés avec un facteur 5 ont une taille comparable à ceux du jeu de données *Ebay* (cf. jeux de données HTML p. 177). Un autre paramètre du script permet de spécifier le nombre de documents XMark désirés dans le jeu de données.

Enfin, et surtout, la génération de jeux de données XMark est paramétrée par une requête définie en XPath. Celle-ci nous permet de produire facilement un document compagnon pour l'ensemble des documents XML générés auparavant.

Ci-dessous, nous décrivons quelques jeux de données XMark que nous avons générés comme indiqué au-dessus et auxquels nous nous référerons plus bas. Un jeu de données est nommé *XMark X-Y-Z*, où *X* est un identifiant de requête, *Y* un facteur, et *Z* le nombre de documents XML dans le jeu de données. Pour chacune des requêtes, nous donnons une expression XPath qui la représente ainsi que le ou les élagages sous lesquels elle est stable (se référer à la sect. 4.5.3 pour leur signification). Notons que toutes ces requêtes sont consistantes avec la DTD de XMark.

XMark 02-2-100 La requête consiste à sélectionner le montant de la première offre pour toutes les enchères en cours. À savoir, pour chaque premier enfant bidder (s'il existe) des éléments open_auction, extraire l'unique nœud étiqueté par increase. L'algorithme d'apprentissage doit inférer que seul le premier fils bidder est concerné.

$$/site/open_auctions/open_auction/bidder[1]/increase$$

Cette requête est stable par D-élagage et ⊤-élagage étendu.

XMark 17-1-100 La requête consiste à sélectionner les personnes qui n'ont pas de page Web. Concrètement, il s'agit d'examiner si un nœud étiqueté par person contient un enfant homepage et, dans le cas contraire, à sélectionner son fils name. La DTD indique que l'élément name est obligatoire, tandis que homepage est facultatif. L'algorithme d'apprentissage doit inférer que la sélection de name dépend de l'absence de homepage parmi ses successeurs.

$$/site/people/person[\mathtt{not}(homepage)]/name$$

Cette requête est stable par D-élagage et ⊤-élagage étendu.

XMark 21-2-50 La requête consiste à sélectionner les enchères en cours qui ont eu au moins trois participations. Cela revient à trouver, dans un arbre XML, les nœuds étiquetés par open_auction qui ont trois fils ou plus d'étiquette bidder, puis à extraire son enfant itemref (obligatoirement présent). L'algorithme d'apprentissage doit inférer que la sélection de itemref dépend de la présence de bidder parmi ses prédécesseurs. Une difficulté supplémentaire est qu'il doit en compter au minimum

trois.

$$/\mathit{site}/\mathit{open_auctions}/\mathit{open_auction}[\texttt{count}(\mathit{bidder}) \geqslant 3]/\mathit{itemref}$$

Cette requête est stable par D-élagage et $\top$-élagage étendu.

XMark A1-6-50 La requête consiste à sélectionner certains mots-clés dans les descriptions des enchères terminées, à savoir les nœuds étiquetés par $\mathit{keyword}$ ayant pour ascendance $\mathit{closed_auction}/\mathit{annotation}/\mathit{description}/\mathit{text}$. L'algorithme d'apprentissage doit inférer cette simple propriété d'hérédité.

$$/\mathit{site}/\mathit{closed_auctions}/\mathit{closed_auction}/\mathit{annotation}/\mathit{description}/\mathit{text}/\mathit{keyword}$$

Cette requête est stable par $\top$-élagage.

XMark A6-1-250 La requête consiste à sélectionner les personnes qui ont renseigné à la fois leur sexe et leur âge dans leur profil, autrement dit extraire les nœuds étiquetés par name ayant un successeur $\mathit{profile}$, qui lui-même a deux enfants gender et age. L'algorithme d'apprentissage doit inférer que la sélection de name dépend de la présence conjointe de deux "neveux".

$$/\mathit{site}/\mathit{people}/\mathit{person}[\mathit{profile}/\mathit{gender}\ \texttt{and}\ \mathit{profile}/\mathit{age}]/\mathit{name}$$

Cette requête est stable par D-élagage étendu.

XMark A8-1-100 La requête consiste à sélectionner les personnes qui ont fourni certains renseignements (adresse, téléphone ou page personnelle, carte de paiement ou profil). Il s'agit d'extraire les nœuds étiquetés par name qui sont des enfants de person, et qui comptent parmi leurs successeurs les éléments $\mathit{address}$, et phone et/ou $\mathit{homepage}$, ainsi que $\mathit{creditcard}$ et/ou $\mathit{profile}$. L'algorithme d'apprentissage doit inférer cette conjonction de disjonctions.

$$/\mathit{site}/\mathit{people}/\mathit{person}[\mathit{address}\ \texttt{and}\ (\mathit{phone}\ \texttt{or}\ \mathit{homepage})\ \texttt{and}\ (\mathit{creditcard}\ \texttt{or}\ \mathit{profile})]/\mathit{name}$$

Cette requête est stable par D-élagage et $\top$-élagage étendu.

```
<!ELEMENT country (name, population?, population_growth?, infant_mortality?,
                  gdp_total?, gdp_agri?, gdp_ind?, gdp_serv?, inflation?,
                  (indep_date | dependent)?, government?, encompassed*,
                  ethnicgroups*, religions*, languages*, border*,
                  (province+ | city+))>
<!ELEMENT name (#PCDATA)>
<!ELEMENT area (#PCDATA)>
<!ELEMENT population (#PCDATA)>
<!ELEMENT population_growth (#PCDATA)>
<!ELEMENT infant_mortality (#PCDATA)>
<!ELEMENT gdp_total (#PCDATA)>
<!ELEMENT gdp_ind (#PCDATA)>
<!ELEMENT gdp_agri (#PCDATA)>
<!ELEMENT gdp_serv (#PCDATA)>
<!ELEMENT inflation (#PCDATA)>
<!ELEMENT indep_date (#PCDATA)>
<!ELEMENT dependent EMPTY>
<!ELEMENT government (#PCDATA)>
<!ELEMENT encompassed EMPTY>
<!ELEMENT ethnicgroups (#PCDATA)>
<!ELEMENT religions (#PCDATA)>
<!ELEMENT languages (#PCDATA)>
<!ELEMENT border EMPTY>
<!ELEMENT province (name, area?, population, city*)>
<!ELEMENT city (name, longitude?, latitude?, population*,
               located_at*, located_on*)>
<!ELEMENT longitude (#PCDATA)>
<!ELEMENT latitude (#PCDATA)>
<!ELEMENT located_at EMPTY>
<!ELEMENT located_on EMPTY>
```

Figure 72. *DTD (simplifiée) du jeu de données Mondial.*

Mondial La base de données XML Mondial (May, 1999, voir aussi `http://www.dbis.informatik.uni-goettingen.de/Mondial/`) décrit, comme l'indique son nom, un ensemble de données géographiques sur le monde (pays, continents, océans, etc.).

Nous avons créé des jeux de données en extrayant de la base de données Mondial l'ensemble des informations sur les pays, à savoir les sous-arbres XML dont la racine est un nœud étiqueté par un symbole country, que nous avons répartis en autant de documents XML, soit 238 fichiers (cf. fig. 73 pour un exemple).

L'ensemble des documents du jeu de données XML Mondial se différencie de XMark sur les points suivants. La DTD (cf. fig. 72), dont le symbole initial est country, compte peu d'éléments (25). Elle définit des récursions horizontales mais pas de récursions verticales ; la profondeur (longueur du chemin menant de la racine à une feuille) maximale des arbres XML est petite (bornée à 3). Par conséquent, les documents sont de petite taille (4 Ko en moyenne seulement) et les arbres XML comportent très peu de nœuds (90 en

```
<country car_code="B" ...>
  <name>Belgium</name>
  <population>10170241</population>
  ...
  <province id="prov−cid−cia−Belgium−5" ...>
    <name>Hainaut</name>
    <area>3787</area>
    <population>1283252</population>
    ...
    <city id="cty−cid−cia−Belgium−Mons" ...>
      <name>Mons</name>
      <longitude>4.5</longitude>
      <latitude>50.4</latitude>
      <population year="87">90720</population>
    </city>
  </province>
  ...
</country>
```

Figure 73. *Extraits d'un document Mondial.*

moyenne) par rapport aux autres jeux de données. Par ailleurs, le jeu de données Mondial présente la particularité d'être très hétérogène (on observe un facteur de 1 à 100 entre le nombre de nœuds de l'arbre XML du plus petit pays et celui du plus grand). Enfin, contrairement au jeu de données XML XMark, le corpus de documents est fixé à 238 et l'on n'opère aucun traitement particulier à ces fichiers.

Similairement au cas de XMark, un jeu de données Mondial est obtenu à partir d'un script paramétré par une requête XPath. Cependant, par rapport à XMark, l'auteur de Mondial ne propose pas de requêtes avec la base de données qu'il a élaborée. Mais il est possible d'écrire soi-même des requêtes XPath pour ces documents XML. Au cours de l'évaluation expérimentale, nous utiliserons l'unique jeu de données Mondial suivant.

Mondial J1 La requête consiste à sélectionner les noms des villes pour lesquelles la latitude et la longitude ne sont pas connues, dans des régions dont la superficie est spécifiée, le tout dans les pays indépendants. Elle s'exprime comme suit en XPath :

```
/country[not(dependent)]
  /province[area]
    /city[not(longitude or latitude)]
      /name
```

L'algorithme d'apprentissage doit inférer l'ensemble de ces conditions. Notons que cette requête est consistante avec la DTD de Mondial et stable par D-élagage et T-élagage étendu.

5.3.2 HTML

Nous voyons le langage Web HTML comme un cas particulier de langage XML. Il existe plusieurs versions de HTML, mais les différences et évolutions entre ces versions sont relativement minimes du point de vue des schémas qui les accompagnent, du moins en ce qui concerne les balises les plus couramment utilisées. Nous utilisons comme schéma la DTD `XHTML 1.0 Transitional`, employée par de nombreux sites Internet. Notons que la sémantique des balises HTML porte sur la présentation des documents et non sur leur contenu, contrairement aux jeux de données XML considérés plus haut. Cela laisse présumer, *a priori*, un apport limité du schéma pour aider à l'apprentissage de requêtes dans le cadre HTML ; les résultats expérimentaux montrent cependant qu'il n'est pas inutile.

Nous considérons deux jeux de données HTML : *Yahoo* et *Ebay*.

Yahoo Le jeu de données *Yahoo* est constitué de 79 documents HTML extraits du site *Yahoo!®* SEARCH *Directory*, qui regroupe des liens par catégories, soit vers d'autres catégories, soit vers des pages sur le Web, ou encore des liens promotionnels. La requête consiste à extraire les liens du deuxième type. Une difficulté pour l'algorithme d'apprentissage provient du fait que la structure générale des arbres HTML varie d'un document à l'autre de l'échantillon.

Ebay Le jeu de données *Ebay* est composé de 34 documents HTML extraits du site d'enchères en ligne *ebay*™. Les pages Web contiennent les résultats de recherche pour un produit regroupés dans un tableau. La requête consiste à sélectionner les liens vers les différents articles résultants. L'apprentissage est rendu difficile par la présence de "pollution" (liens parasites, images, texte formaté, etc.) autour des nœuds à extraire, et par certaines irrégularités dans le tableau où se trouvent tous ces nœuds.

Ces jeux de données ont initialement été mis au point par Carme (2005). Toutefois, les documents HTML ont subi un prétraitement avec le logiciel `tidy` [3] afin, dans un premier

[3]. `http://tidy.sourceforge.net/`.

temps, de les rendre conformes à la DTD `XHTML 1.0 Transitional`. En outre, nous avons réimplémenté une heuristique de filtrage des arbres HTML utilisée par Carme (2005), à savoir qu'une simplification des arbres est opérée en ne conservant que les nœuds dont les étiquettes sont jugées pertinentes structurellement (par exemple, les balises de fonte `i`, `b`, les méta-informations `head`, `script`, sont supprimées, tandis que `table`, `tr`, `td`, `ul`, `li`, etc., sont conservées). Cela implique l'utilisation, lors de l'apprentissage, d'une DTD adaptée et simplifiée pour ces arbres filtrés. Les données présentées dans le tableau 3 ne tiennent pas compte du filtrage, qui est effectué à la volée. Notez par ailleurs que d'autres jeux de données disponibles, comme Google, Okra ou Bigbook, ont été testés ; néanmoins, ils présentent peu d'intérêt pour l'induction guidée par schéma.

Contrairement au cas des jeux de données XML précédents, les requêtes pour les jeux de données *Yahoo* et *Ebay* ne sont pas formellement définies par une requête XPath, mais simplement par l'énumération des annotations positives pour chacun des documents qui les composent (les annotations ont été produites manuellement et réunies dans le document compagnon). Dans notre perspective, elles sont toutes deux consistantes par rapport à la DTD de HTML (ou sa version simplifiée), mais il ne nous est pas possible de déterminer précisément, parmi l'ensemble des élagages que nous considérons, par lesquels elles sont stables ; notons simplement que la sélection de nœuds peut dépendre de quelques éléments contextuels — ce qui exclut *a priori* la stabilité par $\top$-élagage, mais nous verrons que l'emploi de cet élagage reste un bon choix car les tâches d'extraction sont relativement simples.

5.3.3 Récapitulatif

Nous avons présenté des jeux de données pour XML, d'une part, et pour HTML, d'autre part. Leur constitution a nécessité quelques prétraitements. Les jeux de données XML XMark sont obtenus par un script qui permet de fractionner de façon aléatoire une base de données composée d'un unique fichier XML ; le jeu de données Mondial est constitué de 238 sous-arbres XML représentant des pays. Dans les deux cas, les requêtes étudiées sont exprimables avec une expression XPath. Concernant HTML, nous réutilisons des jeux de données déjà existants, mais il est nécessaire de rendre les documents conformes au schéma.

Le tableau 3 récapitule les caractéristiques quantitatives des jeux de données XML et HTML considérés pour l'évaluation expérimentale de l'induction de requêtes guidée par

Tableau 3. *Caractéristiques des jeux de données XML et HTML.*

Jeux XML	nb. documents	% doc. annotés (non-nuls)	nb. annotations / doc. annoté	taille moyenne (Ko)	nb. moy. de nœuds / arbre	stabilité / élagage
XMark 02-2-100	100	98 %	1,8	80	732	$\mathcal{P}_D, X\mathcal{P}_T$
XMark 17-1-100	100	99 %	5,6	37	365	$\mathcal{P}_D, X\mathcal{P}_T$
XMark 21-2-50	50	88 %	1,3	78	674	$\mathcal{P}_D, X\mathcal{P}_T$
XMark A1-6-50	50	88 %	2,2	208	1 842	$\mathcal{P}_T$
XMark A6-1-250	250	79 %	2,2	36	357	$X\mathcal{P}_D$
XMark A8-1-100	100	92 %	2,7	36	346	$\mathcal{P}_D, X\mathcal{P}_T$
Mondial J1	238	23 %	36,3	4	90	$\mathcal{P}_D, X\mathcal{P}_T$

Jeux HTML	nb. documents	% doc. annotés (non-nuls)	nb. annotations / doc. annoté	taille moyenne (Ko)	nb. moy. de nœuds / arbre	stabilité / élagage
Yahoo	79	100 %	23,7	34	281	?
Ebay	34	100 %	47,0	206	1 680	?

schéma. La dernière colonne du tableau indique par quel(s) élagage(s) la requête associée au jeu de données est stable.

5.4 Protocole d'expérimentation

Cette section présente le protocole d'expérimentation utilisé pour évaluer nos algorithmes d'induction de requêtes guidée par schéma. Pour cela, nous simulons le comportement d'un utilisateur lors de la définition d'une nouvelle requête.

Définir interactivement une requête s'apparente à un dialogue entre l'utilisateur et le système d'apprentissage, que l'on peut résumer comme suit. L'utilisateur fournit des informations au système par le biais d'exemples annotés ; à chaque étape, le système infère une nouvelle requête et la soumet à l'utilisateur, qui l'évalue ; l'utilisateur apporte de nouvelles informations au système tant que la qualité de l'apprentissage ne le satisfait pas.

On peut envisager essentiellement deux modes d'apprentissage dans cette optique. Le premier, *fortement interactif*, considère des exemples partiellement annotés, affinant le résultat de l'inférence grâce à des informations obtenues au niveau de chaque nœud, l'utilisateur étant invité par le système à corriger les annotations erronées dues à un apprentissage imparfait. Dans le second, *faiblement interactif*, l'utilisateur fournit plus directement des exemples complètement annotés, les uns après les autres ; le système ajuste son apprentissage avec chaque nouveau document.

Nous avons examiné les deux modes d'apprentissage précédemment décrits. Dans le premier, on compte le nombre de corrections nécessaires à l'identification — ou à une approximation satisfaisante — de la requête cible ; dans le deuxième, on en comptabilise le nombre de documents. Les résultats de nos expérimentations témoignent d'une tendance très sensiblement identique, c'est pourquoi nous présenterons uniquement ceux concernant le mode d'apprentissage faiblement interactif. Le protocole expérimental que nous avons retenu est détaillé dans la suite.

Dans le mode faiblement interactif, on suppose en entrée du programme d'apprentissage un échantillon composé d'exemples complètement annotés uniquement. L'objectif des expériences est de déterminer combien d'exemples sont nécessaires pour identifier la requête cible ou, à défaut, s'en rapprocher au maximum. Pour cela, nous faisons croître progressivement le nombre d'exemples complètement annotés dans l'échantillon et évaluons, à chaque étape, la qualité de l'apprentissage.

À cette fin, nous utilisons une méthode de validation croisée qui s'apparente à un sous-échantillonage aléatoire répété (en anglais, *repeated random subsampling validation*). Cette méthode partage aléatoirement et de façon répétée un jeu de données en deux ensembles, l'un pour l'apprentissage et l'autre pour la validation. Une erreur de prédiction est estimée pour chaque partage réalisé et le taux d'erreur renvoyé est la moyenne sur l'ensemble des partages effectués. Notons que cette méthode autorise la sélection d'un même exemple plusieurs fois dans l'ensemble de validation, ou, inversement, sa non-sélection.

Plus précisément, considérons un échantillon S composé de n arbres complètement annotés. On tire k exemples parmi les n pour constituer l'ensemble d'apprentissage, noté S_l, et m exemples parmi les $n - k$ restant pour l'ensemble de validation, dénoté S_v (d'où $m = \#S_v$). Une expérience se déroule comme suit. D'abord, on établit sans perte de généralité un ordre sur l'ensemble S_l, c'est-à-dire $S_l = \{s_1, \ldots, s_k\}$ avec $s_1 < \ldots < s_k$. Ensuite, l'apprentissage est effectué avec i éléments de l'ensemble S_l pour i variant

de 1 à k. Chaque apprentissage aboutit à une requête $\mathcal{Q}_i$ et donne lieu à une validation sur S_v, et l'on calcul un ensemble de *vrais positifs* (TP), de *faux positifs* (FP) et de *faux négatifs* (FN) comme suit, où $\mathcal{Q}$ est la requête cible.

– Les vrais positifs correspondent aux nœuds sélectionnés par $\mathcal{Q}_i$ qui sont effectivement sélectionnés par $\mathcal{Q}$:

$$\mathsf{TP}(\mathcal{Q}_i) = \sum_{t * \mathcal{Q}(t) \in S_v} \#\big(\mathcal{Q}_i(t) \cap \mathcal{Q}(t)\big)$$

– Les faux positifs sont les nœuds sélectionnés par $\mathcal{Q}_i$ mais qui ne le sont pas par $\mathcal{Q}$:

$$\mathsf{FP}(\mathcal{Q}_i) = \sum_{t * \mathcal{Q}(t) \in S_v} \#\big(\mathcal{Q}_i(t) \setminus \mathcal{Q}(t)\big)$$

– Les faux négatifs sont les nœuds qui ne sont pas sélectionnés par $\mathcal{Q}_i$ mais qui le sont par $\mathcal{Q}$:

$$\mathsf{FN}(\mathcal{Q}_i) = \sum_{t * \mathcal{Q}(t) \in S_v} \#\big(\mathcal{Q}(t) \setminus \mathcal{Q}_i(t)\big)$$

Ces trois valeurs permettent de déterminer le *rappel* (R), la *précision* (P) et la *F-mesure* (F), qui sont des mesures classiques pour évaluer la qualité des systèmes d'extraction d'information :

$$\mathsf{R}(\mathcal{Q}_i) = \frac{\mathsf{TP}(\mathcal{Q}_i)}{\mathsf{TP}(\mathcal{Q}_i) + \mathsf{FN}(\mathcal{Q}_i)} \quad \mathsf{P}(\mathcal{Q}_i) = \frac{\mathsf{TP}(\mathcal{Q}_i)}{\mathsf{TP}(\mathcal{Q}_i) + \mathsf{FP}(\mathcal{Q}_i)} \quad \mathsf{F}(\mathcal{Q}_i) = \frac{2 \times \mathsf{R}(\mathcal{Q}_i) \times \mathsf{P}(\mathcal{Q}_i)}{\mathsf{R}(\mathcal{Q}_i) + \mathsf{P}(\mathcal{Q}_i)}$$

La valeur de ces trois mesures est un réel compris entre 0 et 1. Un rappel de 1 indique que tous les nœuds à extraire ont été effectivement sélectionnés par l'application de la requête ; une précision de 1 signifie que tous les nœuds sélectionnés sont effectivement des nœuds à extraire. Dit autrement, la précision mesure la pertinence de la requête apprise, tandis que le rappel mesure sa complétude. La F-mesure constitue un compromis harmonieux entre rappel et précision. Une F-mesure proche de 1 témoigne donc d'une très bonne qualité d'apprentissage.

Le triplet rappel/précision/F-mesure évalue la qualité de la requête $\mathcal{Q}_i$ en terme de nœuds correctement annotés ou non. Une mesure alternative est la *couverture* (C), qui correspond à une mesure de la qualité de $\mathcal{Q}_i$ en terme d'arbres correctement annotés ou non :

$$\mathsf{C}(\mathcal{Q}_i) = \frac{\#\{t * \beta \in S_v \mid t * \beta = t * \mathcal{Q}_i(t)\}}{\#S_v}$$

La valeur de $\mathsf{C}(\mathcal{Q}_i)$ est également un réel compris entre 0 et 1, qui s'apparente à un taux de réussite complémentaire à l'erreur de prédiction évoquée plus haut. Comme pour la

F-mesure, une couverture proche de 1 signifie une très bonne qualité d'apprentissage. Bien que plus grossière (il suffit d'un seul nœud mal annoté par la requête pour qu'un arbre soit décompté), cette mesure de qualité peut être pertinente dans un contexte où l'on souhaite minimiser les erreurs d'extraction sur les documents pris dans leur globalité. Les résultats de nos expériences montrent que la couverture est généralement corrélée au rappel et/ou à la F-mesure, c'est pourquoi nous exploiterons principalement la F-mesure pour nos analyses.

Conformément à la méthode de sous-échantillonage aléatoire discutée en amont, cette procédure d'évaluation est répétée p fois, et l'on calcule des valeurs moyennes pour chacune des mesures de qualité définie au-dessus. À partir de celles-ci, nous traçons une courbe ayant pour abscisse le nombre d'exemples complètement annotés utilisés pour apprendre (de 1 à k) et pour ordonnée la valeur moyenne calculée pour les p expériences effectuées avec ce nombre d'exemples. L'idée est de comparer sur un même graphe, en fonction du nombre d'exemples complètement annotés considérés, l'évolution des mesures de qualité pour les différentes heuristiques employées (consistance, élagage, combinaison des deux).

Dans les expériences que nous étudions dans la suite, la valeur de k (nombre maximal d'exemples) dépend essentiellement de la difficulté de la requête cible considérée. Certaines requêtes nécessitent un nombre important d'exemples pour que la qualité d'apprentissage soit raisonnable, d'autres requièrent peu d'exemples pour obtenir un résultat satisfaisant. Nous fixons par ailleurs la valeur de m (nombre d'arbres dans l'ensemble de validation), ainsi que celle de p (nombre de répétitions de l'expérience), à 30 pour chaque expérience, ce qui correspond au nombre couramment employé dans la littérature (Dietterich, 1998).

Outre la qualité de l'apprentissage pour chacune des combinaisons d'heuristiques testée, nous avons mesuré, lors de chaque expérience :

- la durée d'apprentissage (en secondes) ;
- les tailles de l'automate initial et de l'automate inféré ;
- le nombre de fusions tentées par l'algorithme d'apprentissage.

De manière générale, ces valeurs croissent avec le nombre d'exemples. Mais elles peuvent permettre de tirer différents enseignements. Un nombre important de fusions tentées (souvent corrélé à une longue durée d'apprentissage) indique que l'algorithme a probablement commencé par de mauvaises fusions ; dans ce cas, la qualité d'apprentissage est généralement mauvaise. Inversement, une convergence rapide avec un faible nombre de

fusions tentées est souvent signe d'un apprentissage de bonne qualité ; toutefois, un petit nombre de fusions tentées peut être associé à faible rappel, indiquant que les généralisations effectuées sont peu satisfaisantes. Cette observation peut être renforcée par la taille de l'automate inféré : si celle-ci est n'est pas beaucoup plus petite que celle de l'automate initial, cela signifie que peu de fusions ont été effectuées et témoigne d'un apprentissage proche du "par cœur" ; en revanche, une taille réduite de l'automate appris est le signe d'un niveau de généralisation important, fréquemment associé à un bon rappel. Nous n'irons pas au-delà des quelques remarques d'ordre général qui précèdent en ce qui concerne ces mesures.

5.5 Résultats et analyse

Cette section présente les résultats d'expériences menées autour de l'induction de requêtes guidée par schéma et en propose une analyse. Nous nous intéressons, successivement, à la vérification de la consistance avec le schéma indépendamment de l'élagage, puis à l'élagage indépendamment de la consistance, et enfin à la conjonction des deux.

Les questions pratiques que nous nous posons sont en lien direct avec les contributions théoriques exposées au chapitre précédent. Concernant la consistance avec le schéma, dans quelle mesure la vérifier empêche de mauvaises généralisations ? La vérification dynamique est-elle plus intéressante que la vérification statique et/ou l'heuristique des fusions horizontales ? Concernant l'élagage, nos interrogations portent principalement sur la notion de requêtes stables. Si une requête est stable par plusieurs élagages, en est-il un plus approprié pour l'apprentissage ?

Nous verrons qu'il est généralement intéressant de tester la consistance de la requête apprise avec le schéma (dynamiquement ou statiquement) ou bien d'employer l'heuristique des fusions horizontales, mais que l'utilisation d'un élagage s'avère indispensable dans tous les cas. Nous tenterons plus particulièrement de déterminer, parmi l'ensemble des élagages à notre disposition — régulier, relatif au schéma ou au langage universel, éventuellement étendu —, lequel est le plus adéquat, relativement à la stabilité ; nous remarquerons que le meilleur élagage est en général celui dans lequel le schéma intervient. Nous constaterons, enfin, que la combinaison de l'élagage et de la consistance avec le test d'inclusion donne, dans certains cas, des résultats significativement meilleurs.

5.5.1 Effets de la vérification de la consistance avec le schéma

Dans un premier temps, nous étudions les effets de la vérification de la consistance avec un schéma (incrémentale ou statique) et de l'heuristique des fusions horizontales sur la qualité générale de l'apprentissage, indépendamment de l'élagage. Pour cela, nous considérons des expériences menées sur quatre jeux de données : *Yahoo*, *Ebay*, *XMark 02-2-100* et *Mondial J1*. Nous observons notamment l'évolution de la F-mesure, qui mesure la qualité générale de l'apprentissage, en fonction du nombre d'exemples dans l'échantillon d'entrée. Les résultats sont présentés dans les figures 74 et 75.

Ces quatre expériences montrent que vérifier la consistance avec le schéma ou bien utiliser l'heuristique des fusions horizontales permet d'améliorer la qualité de l'apprentissage, souvent nettement, par rapport à l'algorithme de base. Ne pas utiliser d'heuristique donne généralement des résultats médiocres. Ces expériences montrent également que tel biais d'apprentissage n'est pas *a priori* systématiquement meilleur que tel autre ; cela dépend en effet de la requête cible considérée. Les deux paragraphes suivants détaillent ces remarques.

Dans les deux jeux de données HTML (fig. 74), vérifier statiquement la consistance avec le schéma donne les meilleurs résultats, de façon significative dans *Ebay*. Dans ce cas, l'écart est plus sensible en terme de précision (courbe non présentée) : la valeur pour la consistance avec inclusion oscille entre $0,6$ et $0,8$, tandis qu'elle est toujours de 1 pour la consistance avec le typage des états. De même, l'algorithme d'apprentissage opère un nombre de fusions en moyenne deux fois plus important dans le premier cas par rapport au second. On peut expliquer ce phénomène par le fait que le typage statique permet probablement, dans le cas de ces deux requêtes, de distinguer utilement les états et donc d'empêcher de mauvaises généralisations.

Dans le jeu de données *XMark 02-2-100* (fig. 75 en haut), l'heuristique des fusions horizontales permet de converger légèrement plus vite que vérifier la consistance. Au contraire, dans l'expérience *Mondial J1* (fig. 75 en bas), la vérification de la consistance avec le test d'inclusion donne le meilleur résultat — ceci dans un temps excessivement rapide (moins de 3 secondes avec 30 exemples, là où les fusions horizontales en nécessitent plus de 20, et 2 à 3 fois moins de fusions tentées). Cette observation peut s'expliquer par le caractère de la requête cible du jeu de données *Mondial J1* (cf. p. 176). La sélection d'un nœud repose en effet sur une combinaison de propriétés que le langage doit respecter ; tester l'inclusion dans le langage défini par le schéma empêche la fusion de certaines paires

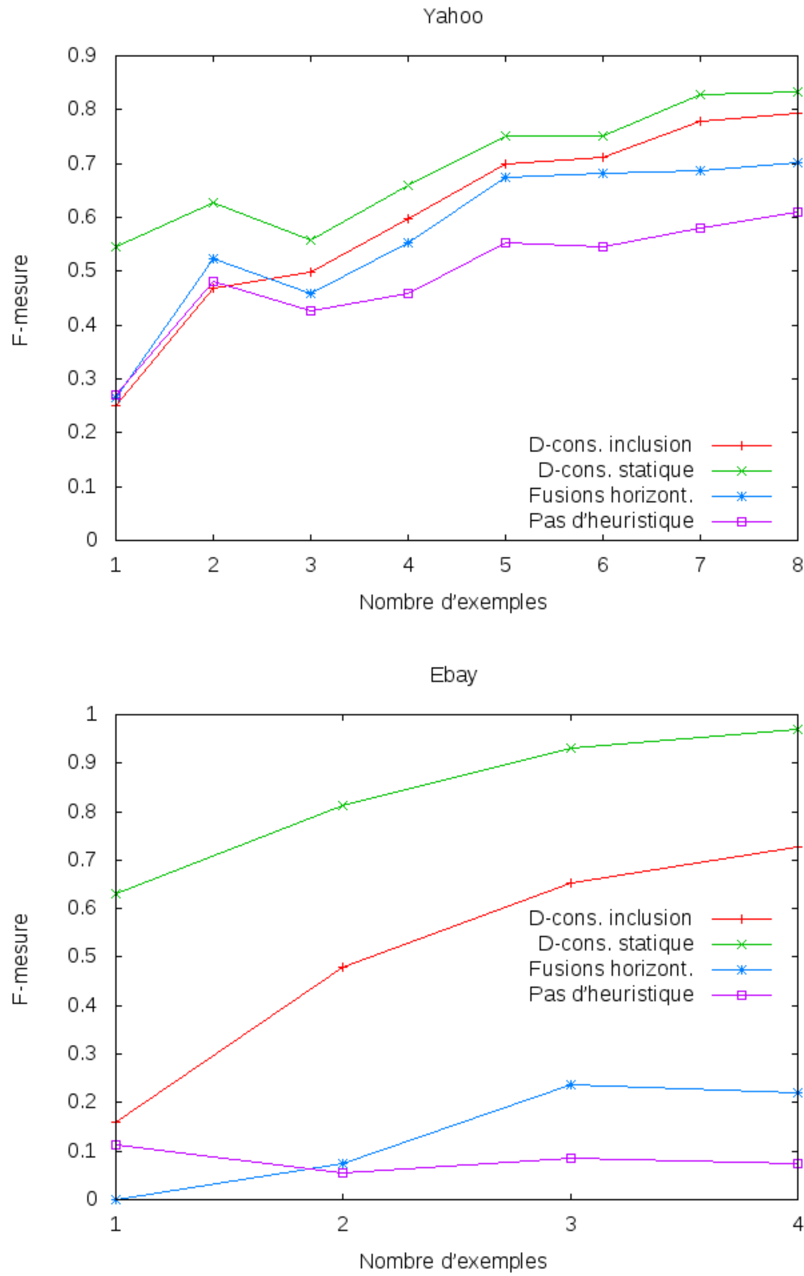


Figure 74. *Courbes de F-mesure de différents biais d'apprentissage pour les expériences Yahoo (en haut) et Ebay (en bas).*

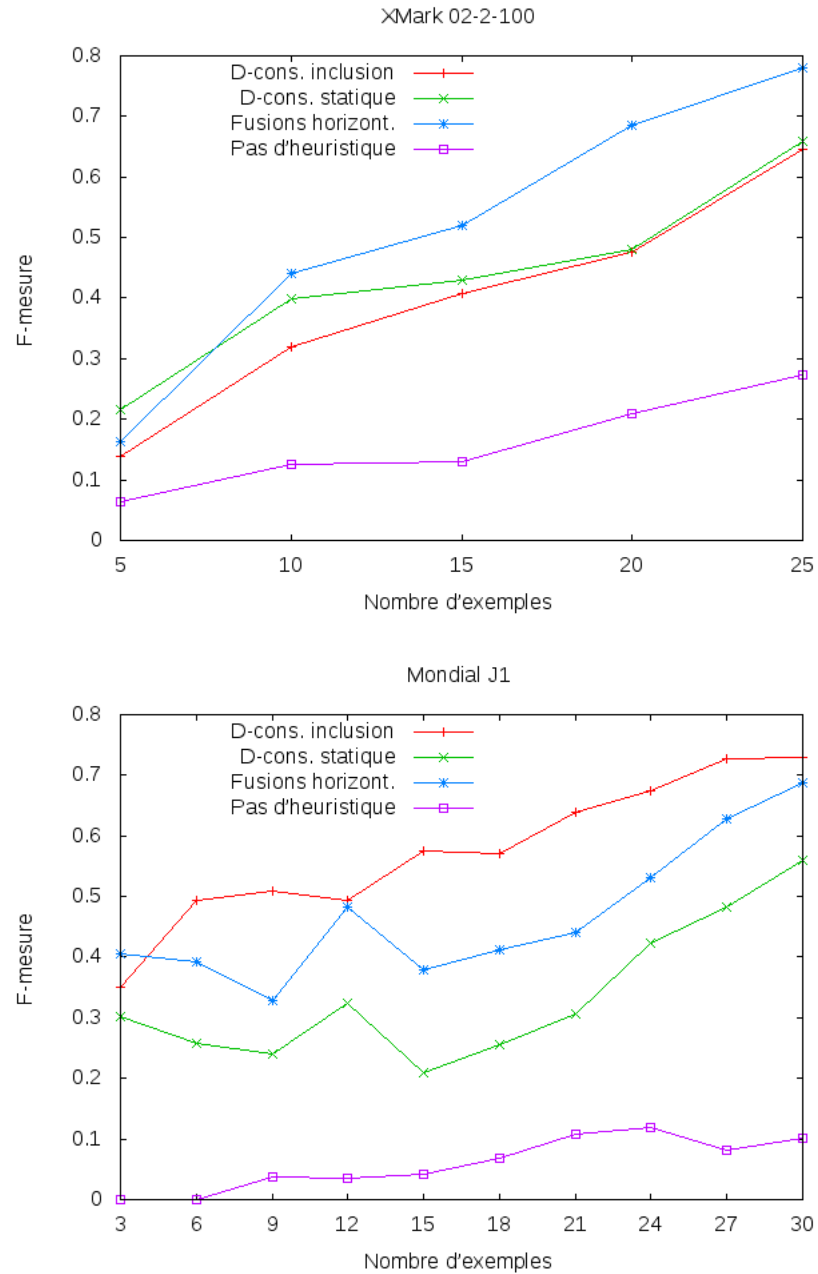


Figure 75. *Courbes de F-mesure de différents biais d'apprentissage pour les expériences XMark 02-2-100 (en haut) et Mondial J1 (en bas).*

d'états, ce qui permet de vérifier ces propriétés sans que des exemples explicites pour tous les cas soient nécessaires. De mauvaises généralisations sont ainsi proscrites naturellement dans l'algorithme qui teste la consistance de façon incrémentale, alors que l'heuristique des fusions horizontales les autorise. L'autre jeu de données ne présente pas une telle particularité.

Bilan Ces premières expériences confirment que vérifier la consistance avec le schéma permet d'améliorer la qualité de l'apprentissage. Elles montrent que les différents biais d'apprentissage testés sont complémentaires. On peut en effet constater que leur apport vaut pour des requêtes cibles ayant des caractéristiques différentes ; le test d'inclusion apparaît comme un biais d'autant plus intéressant que la requête cible comporte plusieurs conditions sur le langage. Toutefois, on remarque, notamment dans le cas des deux jeux de données XML, que malgré un nombre important d'exemples dans l'échantillon d'apprentissage (respectivement 25 et 30), la F-mesure moyenne demeure inférieure à $0,8$.

5.5.2 Apports de l'élagage sans vérifier la consistance avec le schéma

L'élagage permet de réduire la taille des arbres donnés à l'algorithme d'apprentissage, et donc d'accélérer son exécution. En outre, si la requête cible est stable par l'élagage utilisé, la théorie garantit la convergence de l'algorithme (cf. sect. 4.5). En contrepartie, certaines requêtes ne peuvent plus être capturées — celles qui ne sont pas stables par l'élagage employé. Néanmoins, comme mentionné à la section 5.2, l'échantillon caractéristique est rarement disponible en pratique. C'est pourquoi il peut être pertinent d'évaluer des algorithmes avec élagage guidé par schéma, mais qui n'assurent pas la consistance — dans ce cas, des requêtes définies hors du domaine peuvent être inférées. Dans cette section, nous menons donc des expériences avec des requêtes stables par plusieurs élagages, indépendamment de la vérification de la consistance. Les différents jeux de données étudiés ici sont *Ebay*, *XMark A1-6-50*, *XMark 17-1-100* et *XMark 21-2-50*. Les résultats expérimentaux sont montrés figures 76 et 77.

Nous commençons avec le jeu de données *Ebay*. Rappelons que, contrairement aux requêtes pour les jeux XML, nous ignorons par quel(s) élagage(s) la requête *Ebay* est stable — nos expériences n'ont abouti à aucune inconsistance. La courbe de F-mesure

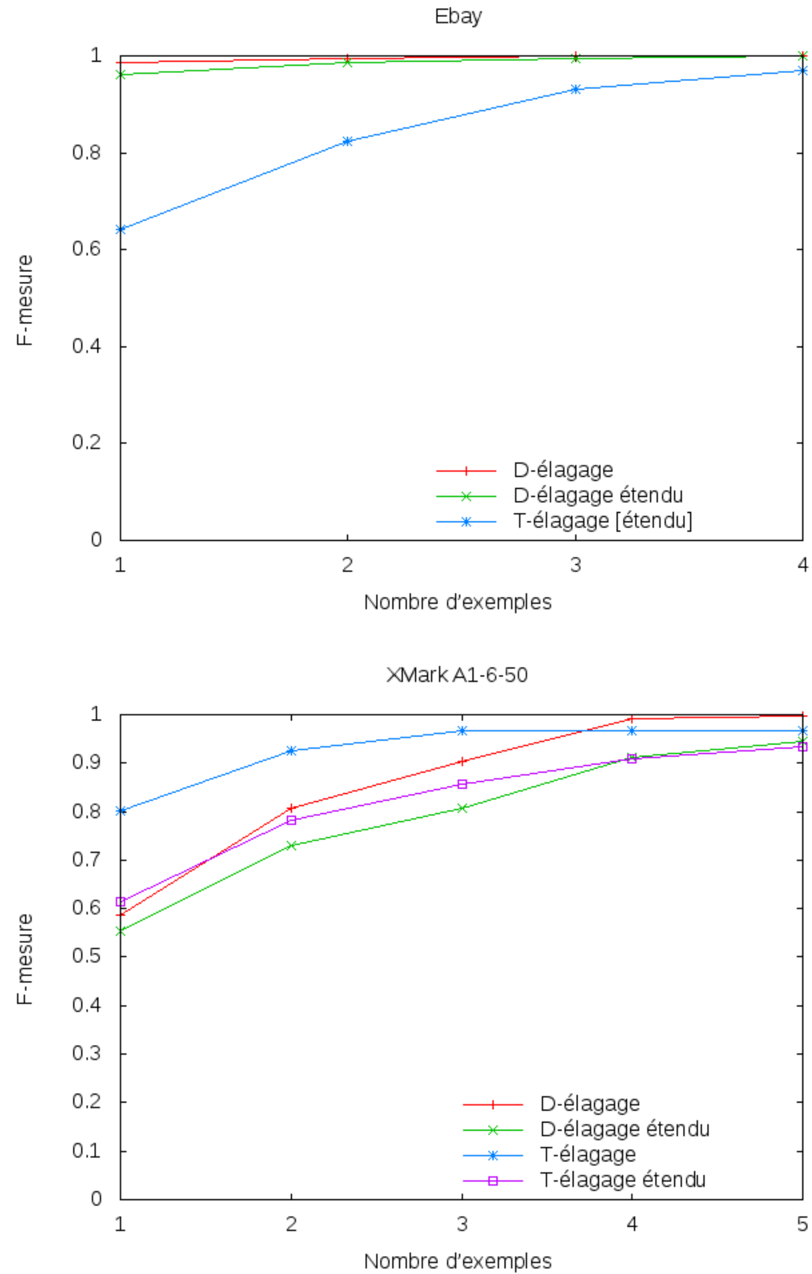


Figure 76. *Courbes de F-mesure des différents élagages pour les expériences Ebay (en haut) et XMark A1-6-50 (en bas).*

présentée à la figure 76 montre la supériorité des D-élagage et D-élagage étendu, c'est-à-dire les élagages tirant parti de la DTD, par rapport aux $\top$-élagage et $\top$-élagage étendu (leurs courbes sont confondues), qui ne s'en servent pas. Dans l'autre jeu de données HTML (*Yahoo*, courbe non présentée ici), le résultat est similaire, bien que l'écart soit moins prononcé. La DTD de HTML peut donc fournir de l'information utile pour l'apprentissage avec élagage.

Nous poursuivons avec les jeux de données XML, d'abord *XMark A1-6-50* (fig. 76). Il s'agit du seul jeu dont la requête cible est stable par $\top$-élagage, donc aussi pour tout autre élagage plus conservatif (tous les élagages que nous utilisons le sont, cf. p.161). Avec trois exemples ou moins, la qualité d'apprentissage est meilleure en utilisant le $\top$-élagage — cet élagage ne conserve que les chemins menant de la racine aux feuilles annotées positivement, seule information *a priori* utile pour apprendre la requête *XMark A1-6-50*. Mais à partir de quatre exemples, le D-élagage prend légèrement le dessus et offre un apprentissage presque parfait (au regard de l'ensemble de validation). Cela suggère que, même pour des requêtes qui semblent plus simples à apprendre, l'information contextuelle apportée par la connaissance du schéma n'est pas inutile.

Nous terminons cette analyse des élagages avec les expériences issues des jeux de données *XMark 17-1-100* et *XMark 21-2-50* (cf. fig. 77). Les deux requêtes sont stables par D-élagage et $\top$-élagage étendu (donc par D-élagage étendu) mais pas par $\top$-élagage — nous avons vérifié expérimentalement, dans ce dernier cas, la survenue de l'automate initial incompatible (lire la discussion p. 148). D'après les résultats, le D-élagage constitue le meilleur choix par rapport aux élagages étendus. Pour le jeu de données *XMark 17-1-100*, avec un seul exemple, la F-mesure est déjà en moyenne de $0,80$ pour le D-élagage contre $0,48$ pour le $\top$-élagage étendu ; le D-élagage étendu rejoint le D-élagage à partir de trois exemples. Pour le jeu de données *XMark 21-2-50*, il est nécessaire de disposer d'un plus grand nombre d'exemples afin d'observer une qualité d'apprentissage satisfaisante. Au-delà de cinq exemples, le D-élagage se détache des élagages étendus ; au bout de vingt exemples, la F-mesure est de $0,97$ pour le premier contre autour de $0,80$ pour les seconds. Là encore, l'information fournie par la DTD se révèle très pertinente.

Bilan L'ensemble de ces résultats expérimentaux permet de confirmer un constat déjà tiré par les expériences conduites par Carme *et al.* (2007), à savoir qu'élaguer est indispensable pour l'induction de requêtes dans les arbres. Cependant, par rapport à ces travaux, nous avons intégré le schéma à la fonction d'élagage. Ainsi, les élagages réguliers

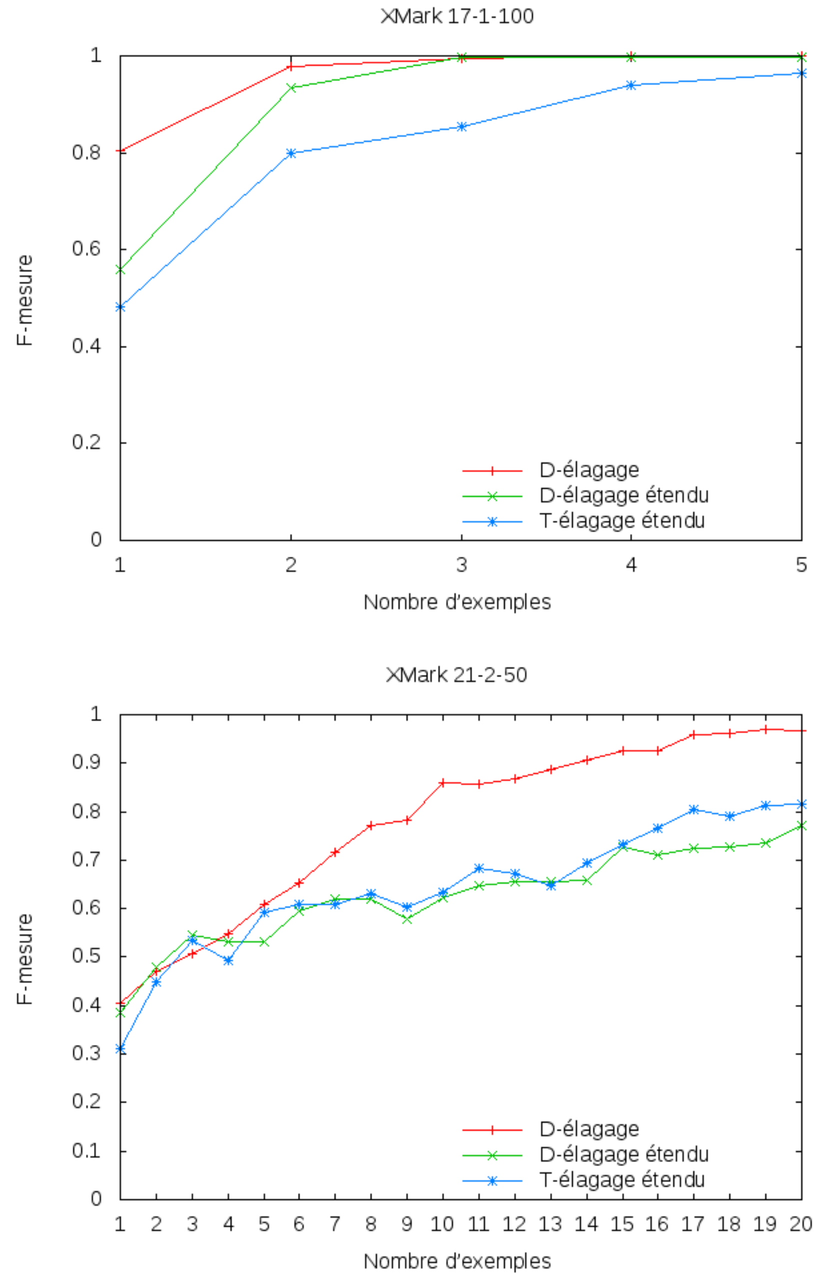


Figure 77. *Courbes de F-mesure des différents élagages pour les expériences XMark 17-1-100 et XMark 21-2-50.*

utilisant le schéma permettent de conserver des informations perdues par les élagages qui utilisent le langage universel ; ces informations d'ordre contextuel semblent utiles, voire essentielles, puisque élaguer à l'aide du schéma apparaît empiriquement être une meilleure option. De plus, un élagage qui se fonde sur le schéma permet de capturer une classe de requêtes plus importante qu'un élagage qui utilise le langage universel, moins conservatif par nature.

5.5.3 Combinaison de l'élagage et de la consistance avec le schéma

Il nous reste à examiner l'impact sur la qualité de l'apprentissage de la combinaison d'un élagage avec la vérification de la consistance relativement au schéma ou avec l'heuristique des fusions horizontales. Nous considérons pour cela les jeux de données *Yahoo*, *XMark 02-2-100*, *XMark A6-1-250* et *XMark A8-1-100*. Les figures 78 et 79 exposent les résultats expérimentaux.

Nous commençons avec les jeux de données *Yahoo* et *XMark 02-2-100* (fig. 78), déjà présentés sans élagage. Pour le premier, vérifier statiquement la consistance avec le schéma constituait la meilleure alternative ; combiner cette heuristique avec le D-élagage améliore l'apprentissage à partir de deux exemples, mais utiliser l'élagage seul se trouve en réalité être le meilleur choix dans ce cas. On observe ainsi une F-mesure supérieure à $0,8$ avec un unique exemple. Notons que le jeu de données HTML *Ebay* (non présenté ici) possède cette même caractéristique, à savoir qu'employer le D-élagage sans vérifier la consistance conduit à un résultat de qualité supérieure.

Pour le jeu de données XML *XMark 02-2-100*, l'heuristique des fusions horizontales apparaissait comme l'heuristique la plus intéressante. Néanmoins, même avec de nombreux exemples la qualité de l'apprentissage se révélait médiocre. Mais contrairement aux jeux de données HTML, où le D-élagage seul est plus intéressant, il apparaît clairement dans cette expérience qu'utiliser une heuristique supplémentaire peut représenter un net avantage. En l'occurrence, combiner D-élagage et fusions horizontales, dans ce cas, permet d'atteindre une F-mesure supérieure à $0,9$ avec deux exemples seulement, alors qu'il en faut 11 pour observer une F-mesure similaire sans l'heuristique des fusions horizontales. Ces deux premières expériences suggèrent ainsi que la combinaison des heuristiques est plus intéressante dans le cas d'un jeu de données XML que dans le cas HTML.

Nous terminons donc notre étude avec deux autres expériences sur des jeux de données XML, dont les résultats sont présentés à la figure 79. Pour l'expérience *XMark A8-1-100*,

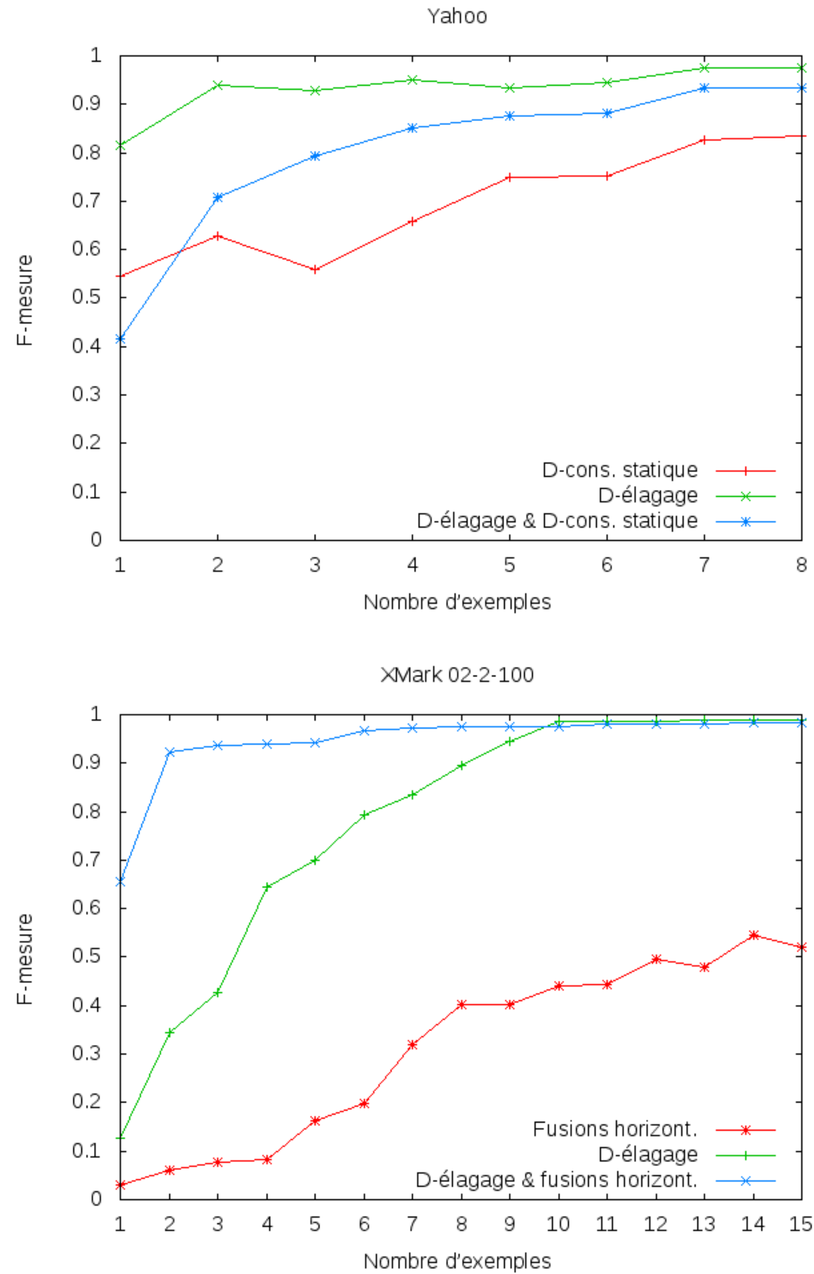


Figure 78. *Courbes de F-mesure de différentes combinaisons d'heuristiques pour les expériences Yahoo et XMark 02-2-100.*

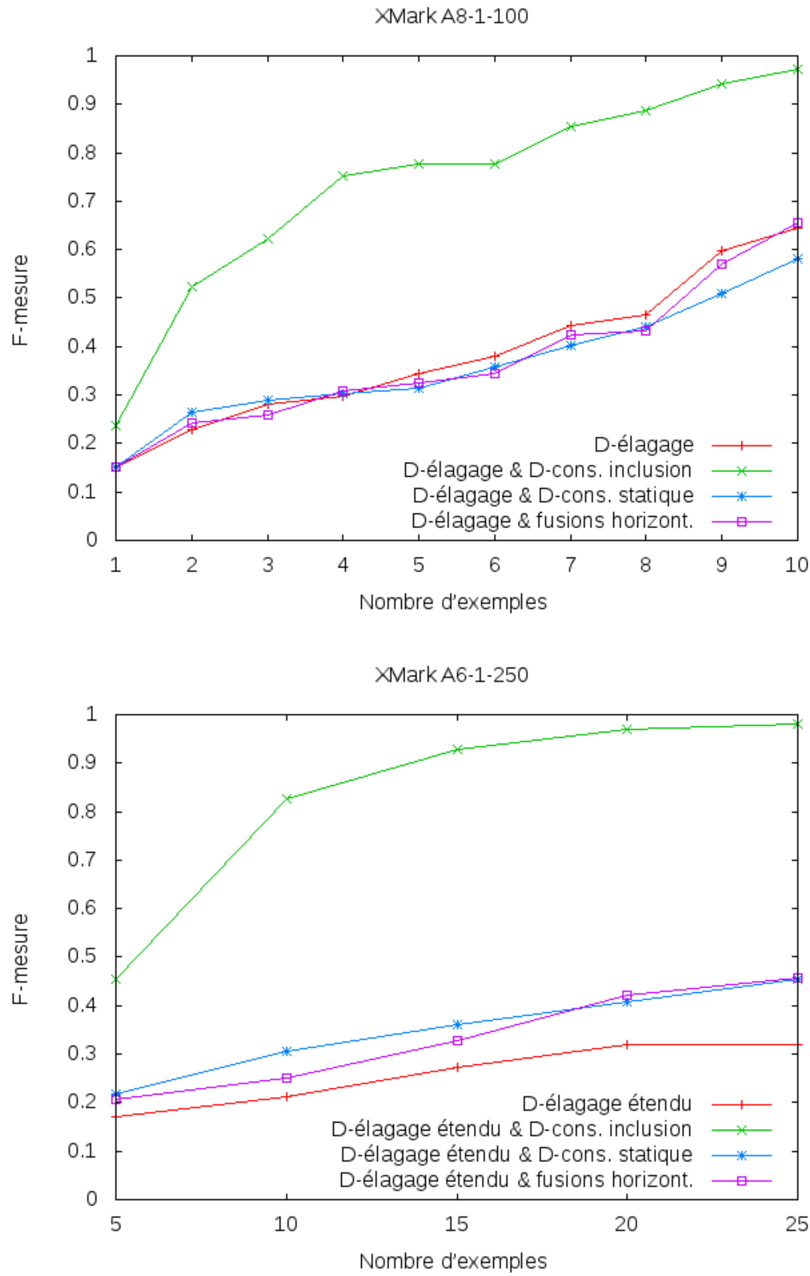


Figure 79. *Courbes de F-mesure de différentes combinaisons d'heuristiques pour les expériences XMark A8-1-100 et XMark A6-1-250.*

nous examinons l'élagage régulier avec le schéma combiné ou non avec la consistance (incrémentale ou statique) ou les fusions horizontales. Il se dégage nettement que vérifier la consistance avec le test d'inclusion en addition de cet élagage est la meilleure option. On observe ainsi approximativement, à partir de trois exemples, un écart de $0,3$ en terme de F-mesure avec les autres combinaisons testées, qui présentent une évolution semblable. De même, la courbe pour l'expérience *XMark A6-1-250* montre que le D-élagage étendu associé à la vérification de la consistance avec l'inclusion représente de loin le meilleur choix heuristique. L'écart en terme de F-mesure est beaucoup plus important que dans le cas précédent puisqu'il atteint la valeur de $0,5$.

Ces deux résultats notables peuvent s'expliquer relativement aux requêtes liées aux jeux de données. Celles-ci apparaissent en effet assez complexes. La sélection d'un nœud dépend essentiellement de propriétés structurelles qu'à la fois l'inclusion dans le schéma et un élagage régulier utilisant ce même schéma permettent de capturer. On peut parler ici de deux cas idéaux pour l'induction de requêtes guidée par schéma car l'élagage relativement au schéma se révèle parfaitement complémentaire de la vérification incrémentale de la consistance avec ce même schéma.

Bilan Les expériences présentées dans cette section permettent d'illustrer tout l'intérêt de l'induction de requêtes guidée par un schéma. Nous avons constaté à nouveau que l'élagage est indispensable dans tous les cas, et qu'il est recommandé d'utiliser un élagage se servant du schéma plutôt que du langage universel. Vérifier la consistance avec le schéma ou utiliser l'heuristique des fusions horizontales en complément de l'élagage permet en général d'améliorer la qualité de l'apprentissage des requêtes XML — ce qui n'est pas le cas pour les données HTML, d'après nos expériences. De plus, le test d'inclusion, associé à un élagage régulier tirant parti du schéma, se révèle extrêmement profitable lorsque la sélection d'un nœud dépend de multiples conditions liées au langage. Cela confirme ainsi que la sémantique contenue dans les schémas XML peut constituer de l'information précieuse pour l'apprentissage.

5.5.4 Remarques conclusives

La théorie nous indique que les requêtes monadiques consistantes avec un schéma et/ou stables par élagage peuvent être identifiées par nos algorithmes d'inférence. Nos expériences montrent que, ayant connaissance de la classe de la requête cible, on peut

apprendre celle-ci efficacement (c.-à-d. avec relativement peu d'exemples) à la seule condition d'employer un élagage approprié — un pour lequel la requête est stable —, éventuellement accompagné de la vérification de la consistance ou des fusions horizontales pour améliorer encore la qualité l'apprentissage le cas échéant. En pratique, cependant, on ne peut pas supposer que l'utilisateur ait une connaissance sur la classe de la requête qu'il souhaite définir grâce au système d'apprentissage. Nous discutons dans les deux derniers paragraphes de perspectives concernant le choix de l'élagage à utiliser et l'opportunité d'utiliser la vérification de la consistance.

Pour le choix de l'élagage, on pourrait se servir de l'interactivité du système d'apprentissage afin de réviser la stratégie d'élagage ; on commencerait, par exemple, avec l'élagage le moins conservatif (T-élagage) et, en cas d'incompatibilité de l'automate initial ou d'un seuil d'interactions jugé trop important, on évoluerait progressivement vers des élagages plus conservatifs (D-élagage, T-élagage étendu, D-élagage étendu). Une autre possibilité serait de solliciter l'utilisateur en lui demandant des annotations supplémentaires sur des nœuds non sélectionnés mais qui justifient la sélection de tel ou tel nœud ; ces nœuds annotés seraient alors conservés par l'élagage et constitueraient donc des informations utiles pour l'algorithme d'apprentissage. L'idée commune à ces deux propositions est de permettre au système d'apprentissage d'aboutir à un élagage pour lequel la requête est stable, ce qui garantit théoriquement son identification. La première présente l'avantage de ne pas demander de travail supplémentaire à l'utilisateur, tandis que l'intérêt de la seconde est d'obtenir de sa part des informations plus précises.

Il serait enfin intéressant de réfléchir à des méta-heuristiques qui permettent, à partir d'une observation des exemples proposés à l'algorithme d'apprentissage par l'utilisateur, de déterminer s'il est approprié de vérifier la consistance avec le schéma ou bien l'heuristique des fusions horizontales et, si oui, lequel est le plus adapté. Nous avons vu expérimentalement que des requêtes relativement complexes sont mieux apprises lorsque l'inclusion dans le schéma est activée. Un mécanisme examinant la structure et la (dis)similarité des exemples une fois ceux-ci élagués pourrait détecter des irrégularités structurelles indiquant que l'emploi de la consistance semble plutôt judicieux dans tel ou tel cas.

Conclusion

L'UTILISATION des connaissances contenues dans le schéma permet-elle d'améliorer la qualité de l'apprentissage de requêtes dans les documents semi-structurés ? L'objet principal de cette thèse était d'étudier cette question et, si possible, de lui apporter une réponse positive.

Pour cela, nous avons d'abord examiné par quels moyens intégrer ces connaissances dans un algorithme d'inférence d'automates d'arbres à base de fusions d'états. Une première façon est d'utiliser le schéma comme une connaissance sur le langage cible : la requête inférée doit être consistante avec le schéma. Dans cette optique, nous avons conçu de nouveaux algorithmes pour vérifier efficacement l'inclusion dans les automates d'arbres déterministes et des représentations de schémas XML par automates d'arbres. En particulier, nous avons montré comment vérifier l'inclusion d'un automate *stepwise* A dans une DTD D sur Σ en temps $\mathcal{O}(|A| \times |\Sigma| \times |D|)$ grâce à la notion d'automate factorisé déterministe, un modèle de représentation compacte des DTD que nous avons introduit. Nous avons également présenté la détection "au plus tôt" de l'échec de l'inclusion, une optimisation qui permet de rendre l'algorithme incrémental par rapport à l'ajout d'ϵ-règles à l'automate A. Cette dernière opération simule la fusion de deux états au cours du processus d'apprentissage ; ainsi, vérifier que le langage de l'automate hypothèse — plus exactement sa projection sur Σ — est contenu dans celui d'une DTD ne nécessite pas de tester l'inclusion après chaque fusion. Les expériences présentées dans la section 2.7 montrent l'efficacité de cette méthode en pratique. L'utilisation du test d'inclusion incrémental dans les algorithmes d'apprentissage n'augmente pas le temps d'exécution. Notons, en outre, qu'il peut s'agir d'une technique intéressante pour vérifier efficacement l'inclusion d'automates dans un contexte différent de l'induction de requêtes — par exemple l'inclusion de schémas XML ou l'inclusion de requêtes.

Une autre façon d'intégrer les connaissances du schéma dans l'algorithme d'induction de requêtes est de l'employer dans les heuristiques d'élagage. Élaguer un arbre consiste à en remplacer des sous-arbres par différents symboles d'élagage. Nous utilisons pour ces derniers les états de l'automate (factorisé) déterministe qui représente le schéma, ce qui permet de conserver une information précieuse pour l'apprentissage. L'algorithme d'induction infère en effet, à partir des arbres annotés élagués, un automate *stepwise* qui représente une requête monadique. Toutefois, l'élagage restreint la classe de requêtes qui peuvent être identifiées puisqu'il implique une perte d'information. Nos travaux nous ont permis de déterminer précisément quelles requêtes sont apprenables par un algorithme qui emploie l'élagage ; nous les avons baptisées requêtes stables et avons prouvé un résultat d'apprenabilité pour cette classe de requêtes et leur représentation par automates d'arbres. Cette contribution permet en outre de résoudre une question laissée en suspens par Carme (2005). Par ailleurs, nous avons proposé quelques élagages pouvant être utilisés en pratique pour l'induction de requêtes. Une classification de ces élagages a été présentée, relativement à la notion de stabilité. Les élagages prenant en compte les états de l'automate du schéma sont par nature plus conservatifs que ceux simplement basés sur le langage universel. L'idée sous-jacente est que cet enrichissement sémantique de l'élagage peut avoir un effet bénéfique sur la qualité de l'apprentissage.

Parallèlement aux contributions théoriques récapitulées au-dessus, nous avons adopté une démarche expérimentale afin de valider leur intérêt pratique. Pour cela, nous avons développé un nouveau système d'apprentissage intégrant l'ensemble de nos algorithmes. Pour tester ceux-ci, nous simulons le comportement d'un utilisateur lors de la définition d'une nouvelle requête et évaluons la qualité de l'apprentissage en fonction du nombre d'interactions nécessaires à l'identification ou à une approximation raisonnable de la requête. Nos expériences ont porté à la fois sur des jeux de données HTML et XML. Les résultats confirment que l'élagage est indispensable pour atteindre une qualité d'apprentissage satisfaisante ; ils montrent également que vérifier la consistance avec le schéma peut s'avérer judicieux, notamment pour des requêtes cibles comportant des conditions sur le langage (conjonction de disjonctions, négation, etc.). Plus intéressant dans la perspective de l'induction de requêtes guidée par schéma, et à la lumière de la notion de stabilité introduite dans cette thèse, nous avons constaté qu'un élagage tenant compte de l'information fournie par le schéma est plus performant qu'un élagage basé sur le langage universel, même si la requête cible est par définition stable par ces différents élagages. En conclusion, nous pouvons affirmer que la sémantique contenue dans le schéma est réellement utile pour l'induction de requêtes XML.

Perspectives

Nous avons déjà esquissé quelques perspectives au cours des précédents chapitres. Au chapitre 4, nous avons laissé ouverte la question de l'apprenabilité des requêtes D-consistantes avec l'algorithme $\mathtt{static}\text{-}\mathtt{t}\mathcal{RPNI}_{\mathrm{D}}$, dont la preuve de la véracité reste encore à élaborer. Par ailleurs, nous avons argumenté, à la fin du chapitre 5, l'intérêt de mettre au point des méta-heuristiques pour choisir un élagage approprié, par exemple en tirant parti de l'interactivité du processus d'apprentissage. Nous avons aussi mentionné, à plusieurs reprises, la possibilité d'utiliser un schéma inféré par un processus indépendant pour guider l'apprentissage. Notons toutefois que quelques expériences très préliminaires menées avec des données HTML et un algorithme d'inférence de DTD inspiré par Bex *et al.* (2006) n'ont pas été concluantes. Cela est dû au fait que les langages horizontaux des schémas inférés de cette manière ne capturent que la classe des langages 2-testables. D'autres algorithmes doivent être testés pour obtenir des schémas utiles pour l'induction de requêtes.

Par ailleurs, les principaux résultats théoriques et expérimentaux sur l'induction de requêtes présentés dans cette thèse concernent les requêtes *monadiques*. Nous discutons ainsi de l'extension de l'induction guidée par schéma à des requêtes plus élaborées, à savoir les requêtes n-monadiques et les requêtes n-aires. Puis nous abordons le cas des requêtes qui portent sur le contenu textuel dans les documents semi-structurés. Enfin, nous évoquons l'apprentissage de transformations d'arbres guidé par schéma.

Les *requêtes n-monadiques* combinent n requêtes monadiques indépendantes les unes des autres. Le modèle de représentation des requêtes monadiques peut facilement être étendu pour représenter des requêtes n-monadiques ; il suffit de considérer un vecteur de booléens de dimension n au lieu de l'alphabet $\mathbb{B}$ (qui est un vecteur de booléens de dimension 1) pour représenter les annotations sur les nœuds. Les aspects théoriques de nos travaux sur l'induction de requêtes monadiques devraient ainsi aisément s'adapter pour l'induction de requêtes n-monadiques. Notons qu'une requête n-monadique est stable par élagage si les n requêtes monadiques qu'elles combinent le sont. Quelques expériences préliminaires que nous avons pu mener sur le jeu de données XMark montrent que l'induction de requêtes guidée par schéma présente un intérêt pour les requêtes n-monadiques. D'autres expériences doivent être conduites, notamment dans le cas où les n requêtes sont stables par différents élagages.

Contrairement aux requêtes (n-)monadiques, les *requêtes* n*-aires* permettent de sélectionner des tuples de n nœuds. Lemay *et al.* (2006) ont étendu le formalisme des transducteurs de sélection de nœuds afin de représenter celles-ci ; ils ont également identifié une sous-classe apprenable, les requêtes n-aires *polynomialement bornées* (il s'agit d'une restriction sur le nombre de réponses calculées par une requête étant donné un arbre). Vérifier la consistance avec le schéma dans le cadre de l'induction de requêtes n-aires est trivial puisque cela ne concerne que la composante non annotée des arbres. En revanche, différentes stratégies pourraient être retenues pour l'élagage : faut-il élaguer directement l'arbre support (il contient l'ensemble des tuples) ou bien les arbres de sa décomposition indépendamment (un arbre par tuple) ? La notion de stabilité d'une requête n-aire relativement à un élagage devrait être révisée en conséquence. Des expériences sont nécessaires pour évaluer l'intérêt de guider l'apprentissage avec le schéma.

Les requêtes XML que nous avons considérées dans nos travaux portent simplement sur la structure arborescente des documents. Or les documents semi-structurées contiennent des parties textuelles plus ou moins conséquentes sur lesquelles peuvent se baser des requêtes qui intéressent les utilisateurs. L'inférence d'automates d'arbres doit donc être combinée avec des approches sur les chaînes pour capturer ces requêtes. Certains langages de schémas, comme XML Schema ou Relax NG, apportent des informations sur le type des données contenues dans les documents XML, aussi bien au niveau des éléments que des attributs. Du point de vue de l'induction de requêtes guidée par schéma, il pourrait être intéressant d'exploiter ces informations afin de définir des biais d'apprentissage.

Ajoutons, pour conclure, que l'induction de requêtes XML trouve son prolongement dans l'apprentissage de transformations XML (p. ex. Lemay *et al.*, 2010). Dans ce cadre, les exemples correspondent à des couples d'entrée-sortie de la transformation cible. En outre, l'arbre d'entrée (resp. de sortie) d'une transformation est supposé consistant avec un schéma d'entrée (resp. de sortie), d'où la possible extension de l'induction guidée par schéma aux transformations d'arbres.

A

Sommaire

A.1 Calculer la réponse d'une requête représentée par un TSNé 202
A.2 DTD de XMark 204

A.1 Calculer la réponse d'une requête représentée par un transducteur de sélection de nœuds élagueur

Le calcul de la réponse d'une requête avec un transducteur de sélection de nœuds élagueur fonctionne de façon analogue au cas non élagueur. Une phase ascendante génère des états candidats et une phase descendante conserve ceux qui peuvent être utilisés pour sélectionner des nœuds. Cependant, le calcul s'effectue avec l'automate de jonction défini à la section 4.4.3 afin de simuler les élagages sur l'arbre d'entrée.

Formellement, si A est un TSNé$_F$ sur Σ, t un arbre sur Σ tel que $t \in \mathcal{L}^u(F)$, et t_c son codage curryfié, la phase ascendante calcule l'arbre t'_c sur $\Sigma_@ \times 2^{sta(j(A,F))}$ comme suit :

$$t'_c(\nu) = \big(t_c(\nu), eval_{\Pi_\Sigma(j(A,F))}((t_c)_{|\nu})\big)$$

Par rapport au cas non élagueur, des règles $(a,1) \to p$ et $(a,0) \to q$ de $j(A,F)$ peuvent apparaître en concurrence pour la sélection d'une feuille dans l'arbre curryfié. Par F-pseudo-fonctionnalité, on est assuré que $p \in sta(A)$ et $q \in sta(F)$, donc dans ce cas la feuille est sélectionnée. La fonction de sélection $select^{\downarrow}_{j(A,F)} : T_{\Sigma_@ \times 2^{sta(j(A,F))}} \times 2^{sta(j(A,F))} \to T^{bin}_{(\Sigma\times\mathbb{B})_@}$ est définie en conséquence comme suit :

$$select^{\downarrow}_{j(A,F)}\big((a,Q),S\big) = \begin{cases} (a,1) & \text{si } (a,1) \to p \in rul(j(A,F)) \text{ et } p \in Q \cap S \\ (a,0) & \text{sinon} \end{cases}$$

$$\begin{aligned} select^{\downarrow}_{j(A,F)}\big((@(t'_1,t'_2),Q),S\big) = \ & @(t1,t2) \text{ avec} \\ & t_1 = select^{\downarrow}_{j(A,F)}(t'_1,S_1), \\ & t_2 = select^{\downarrow}_{j(A,F)}(t'_2,S_2), \\ & S_1 \times S_2 = \{(p_1,p_2) \in sta(j(A,F))^2 \mid \\ & \qquad p_1@p_2 \to p \in rul(j(A,F)) \wedge p \in Q \cap S\} \end{aligned}$$

L'appel initial à cette fonction est $t''_c = select^{\downarrow}_A\big(t'_c, fin(j(A,F))\big)$. Le résultat du calcul est l'ensemble $\mathcal{Q}_A(t) = \{\nu \in nod(t'') \mid curry^u(t'') = t''_c \wedge t''(\nu) = 1\}$. Notons que comme t est élément de $\mathcal{L}^u(F)$, aucun des ensembles S ne peut être vide (ils contiennent au moins un état d'évaluation de F).

Le calcul de la réponse d'une requête par un TSNé$_F$ A sur un arbre t, tous deux sur Σ, peut s'effectuer en temps $\mathcal{O}(|t| \times (|A| + |F|))$. La phase ascendante s'apparente à tester l'appartenance de t dans $\mathcal{L}^u(\Pi_\Sigma(j(A,F)))$, où $\Pi_\Sigma(j(A,F))$ est possiblement non déterministe,

et produit en sortie un arbre de taille $\mathcal{O}(|t| \times (|A| + |F|))$. La phase descendante traverse cet arbre en une passe et en temps linéaire.

A.2 DTD de XMark

```
<!ELEMENT site          (regions, categories, catgraph, people, open_auctions,
                        closed_auctions)>

<!ELEMENT categories    (category+)>
<!ELEMENT category      (name, description)>
<!ATTLIST category      id ID #REQUIRED>
<!ELEMENT name          (#PCDATA)>
<!ELEMENT description   (text | parlist)>
<!ELEMENT text          (#PCDATA | bold | keyword | emph)*>
<!ELEMENT bold          (#PCDATA | bold | keyword | emph)*>
<!ELEMENT keyword       (#PCDATA | bold | keyword | emph)*>
<!ELEMENT emph          (#PCDATA | bold | keyword | emph)*>
<!ELEMENT parlist       (listitem)*>
<!ELEMENT listitem      (text | parlist)*>

<!ELEMENT catgraph      (edge*)>
<!ELEMENT edge          EMPTY>
<!ATTLIST edge          from IDREF #REQUIRED to IDREF #REQUIRED>

<!ELEMENT regions       (africa, asia, australia, europe, namerica, samerica)>
<!ELEMENT africa        (item*)>
<!ELEMENT asia          (item*)>
<!ELEMENT australia     (item*)>
<!ELEMENT namerica      (item*)>
<!ELEMENT samerica      (item*)>
<!ELEMENT europe        (item*)>
<!ELEMENT item          (location, quantity, name, payment, description, shipping,
                        incategory+, mailbox)>
<!ATTLIST item          id ID #REQUIRED
                        featured CDATA #IMPLIED>
<!ELEMENT location      (#PCDATA)>
<!ELEMENT quantity      (#PCDATA)>
<!ELEMENT payment       (#PCDATA)>
<!ELEMENT shipping      (#PCDATA)>
<!ELEMENT reserve       (#PCDATA)>
<!ELEMENT incategory    EMPTY>
<!ATTLIST incategory    category IDREF #REQUIRED>
<!ELEMENT mailbox       (mail*)>
<!ELEMENT mail          (from, to, date, text)>
<!ELEMENT from          (#PCDATA)>
<!ELEMENT to            (#PCDATA)>
<!ELEMENT date          (#PCDATA)>
<!ELEMENT itemref       EMPTY>
<!ATTLIST itemref       item IDREF #REQUIRED>
<!ELEMENT personref     EMPTY>
<!ATTLIST personref     person IDREF #REQUIRED>
```

```
<!ELEMENT people           (person*)>
<!ELEMENT person           (name, emailaddress, phone?, address?, homepage?,
                           creditcard?, profile?, watches?)>
<!ATTLIST person           id ID #REQUIRED>
<!ELEMENT emailaddress     (#PCDATA)>
<!ELEMENT phone            (#PCDATA)>
<!ELEMENT address          (street, city, country, province?, zipcode)>
<!ELEMENT street           (#PCDATA)>
<!ELEMENT city             (#PCDATA)>
<!ELEMENT province         (#PCDATA)>
<!ELEMENT zipcode          (#PCDATA)>
<!ELEMENT country          (#PCDATA)>
<!ELEMENT homepage         (#PCDATA)>
<!ELEMENT creditcard       (#PCDATA)>
<!ELEMENT profile          (interest*, education?, gender?, business, age?)>
<!ATTLIST profile          income CDATA #IMPLIED>
<!ELEMENT interest         EMPTY>
<!ATTLIST interest         category IDREF #REQUIRED>
<!ELEMENT education        (#PCDATA)>
<!ELEMENT income           (#PCDATA)>
<!ELEMENT gender           (#PCDATA)>
<!ELEMENT business         (#PCDATA)>
<!ELEMENT age              (#PCDATA)>
<!ELEMENT watches          (watch*)>
<!ELEMENT watch            EMPTY>
<!ATTLIST watch            open_auction IDREF #REQUIRED>

<!ELEMENT open_auctions    (open_auction*)>
<!ELEMENT open_auction     (initial, reserve?, bidder*, current, privacy?, itemref,
                           seller, annotation, quantity, type, interval)>
<!ATTLIST open_auction     id ID #REQUIRED>
<!ELEMENT privacy          (#PCDATA)>
<!ELEMENT initial          (#PCDATA)>
<!ELEMENT bidder           (date, time, personref, increase)>
<!ELEMENT seller           EMPTY>
<!ATTLIST seller           person IDREF #REQUIRED>
<!ELEMENT current          (#PCDATA)>
<!ELEMENT increase         (#PCDATA)>
<!ELEMENT type             (#PCDATA)>
<!ELEMENT interval         (start, end)>
<!ELEMENT start            (#PCDATA)>
<!ELEMENT end              (#PCDATA)>
<!ELEMENT time             (#PCDATA)>
<!ELEMENT status           (#PCDATA)>
<!ELEMENT amount           (#PCDATA)>

<!ELEMENT closed_auctions  (closed_auction*)>
<!ELEMENT closed_auction   (seller, buyer, itemref, price, date, quantity, type,
                           annotation?)>
<!ELEMENT buyer            EMPTY>
```

```
<!ATTLIST buyer        person IDREF #REQUIRED>
<!ELEMENT price        (#PCDATA)>
<!ELEMENT annotation   (author, description?, happiness)>

<!ELEMENT author       EMPTY>
<!ATTLIST author       person IDREF #REQUIRED>
<!ELEMENT happiness    (#PCDATA)>
```

Bibliographie

Serge ABITEBOUL, Peter BUNEMAN et Dan SUCIU : *Data on the Web: From Relational to Semistructured Data and XML*. Morgan Kaufmann Publishers Inc., 2000. *Cité page 1.*

Serge ABITEBOUL, Richard HULL et Victor VIANU : *Foundations of Databases*. Addison-Wesley Longman Publishing Co., Inc., 1995. *Cité page 62.*

Helena AHONEN : Automatic generation of SGML content models. *Electronic Publishing–Origination, Dissemination and Design*, 8(2/3):195–206, septembre 1995. *Cité page 121.*

Helena AHONEN : *Generating Grammars for Structured Documents Using Grammatical Inference Methods*. Thèse de doctorat, University of Helsinki, novembre 1996. *Cité page 121.*

Dana ANGLUIN : Learning regular sets from queries and counterexamples. *Information and Computation*, 75(2):87–106, novembre 1987. *Cité page 157.*

Dana ANGLUIN : Queries and concept learning. *Machine Learning*, 2(4):319–342, avril 1988. *Cité page 157.*

Valentin ANTIMIROV : Partial derivatives of regular expressions and finite automaton constructions. *Theoretical Computer Science*, 155(2):291–319, mars 1996. *Cité page 24.*

Robert BAUMGARTNER, Sergio FLESCA et Georg GOTTLOB : Visual web information extraction with Lixto. *In 27th International Conference on Very Large Data Bases*, 2001. *Cité page 5.*

Marie-Pierre BÉAL et Maxime CROCHEMORE : Minimizing incomplete automata. *In 7th International Workshop on Finite-State Methods and Natural Language Processing*, 2008. *Cité pages 23 et 25.*

Anders BERGLUND, Scott BOAG, Don CHAMBERLIN, Mary F. FERNÁNDEZ, Michael KAY et Jonathan ROBIE : XML path language (XPath) 2.0. W3C Recommendation, 23 janvier 2007. URL : `http://www.w3.org/TR/2007/REC-xpath20-20070123/`. *Cité pages 3 et 99.*

Gérard BERRY et Ravi SETHI : From regular expressions to deterministic automata. *Theoretical Computer Science*, 48(1):117–126, décembre 1986. *Cité page 24.*

Jean BERSTEL et Luc BOASSON : Formal properties of XML grammars and languages. *Acta Informatica*, 38(9):649–671, août 2002. *Cité page 47.*

Geert Jan BEX, Wouter GELADE, Frank NEVEN et Stijn VANSUMMEREN : Learning deterministic regular expressions for the inference of schemas from XML data. *In 17th International Conference on World Wide Web*, 2008. *Cité pages 7 et 167.*

Geert Jan BEX, Frank NEVEN, Thomas SCHWENTICK et Karl TUYLS : Inference of concise DTDs from XML data. *In 32nd International Conference on Very Large Data Bases*, 2006. *Cité page 199.*

Geert Jan BEX, Frank NEVEN et Jan VAN DEN BUSSCHE : DTDs *versus* XML Schema: A practical study. *In 7th International Workshop on the Web and Databases*, 2004. *Cité pages 47 et 60.*

Norbert BLUM : An $O(n \log n)$ implementation of the standard method for minimizing n-state finite automata. *Information Processing Letters*, 57(2):65–69, janvier 1996. *Cité page 23.*

Scott BOAG, Don CHAMBERLIN, Mary F. FERNÁNDEZ, Daniela FLORESCU, Jonathan ROBIE et Jérôme SIMÉON : XQuery 1.0: An XML query language. W3C Recommendation, 23 janvier 2007. URL : `http://www.w3.org/TR/2007/REC-xquery-20070123/`. *Cité pages 3 et 99.*

Ahmed BOUAJJANI, Peter HABERMEHL, Lukáš HOLÍK, Tayssir TOUIL et Tomáš VOJNAR : Antichain-based universality and inclusion testing over nondeterministic finite tree automata. *In 13th International Conference on Implementation and Applications of Automata*, 2008. *Cité page 60.*

Tim BRAY, Jean PAOLI, C. M. SPERBERG-MCQUEEN, Eve MALER et François YERGEA : Extensible markup language (XML) 1.0 (fifth edition). W3C Recommendation, 26 no-

vembre 2008. URL : `http://www.w3.org/TR/2008/REC-xml-20081126/`. *Cité pages 1, 3, 46, 48 et 167.*

Anne BRÜGGEMANN-KLEIN : Regular expressions into finite automata. *Theoretical Computer Science*, 120(2):197–213, novembre 1993. *Cité pages 24 et 25.*

Anne BRÜGGEMANN-KLEIN, Makoto MURATA et Derick WOOD : Regular tree and regular hedge languages over unranked alphabets: Version 1. Rapport technique HKUST-TCSC-2001-05, 3 avril 2001. *Cité pages 5, 12 et 41.*

Anne BRÜGGEMANN-KLEIN et Derick WOOD : One-unambiguous regular languages. *Information and Computation*, 140(2):229–253, février 1998. *Cité pages 20, 21 et 25.*

Julien CARME : *Inférence de requêtes régulières dans les arbres et applications à l'extraction d'information sur le Web*. Thèse de doctorat, Université Lille 3, 2005. *Cité pages 96, 177, 178 et 198.*

Julien CARME, Rémi GILLERON, Aurélien LEMAY et Joachim NIEHREN : Interactive learning of node selecting tree transducers. *Machine Learning*, 66(1):33–67, janvier 2007. *Cité pages 4, 6, 7, 8, 13, 97, 101, 103, 105, 111, 112, 118, 134, 157, 167 et 189.*

Julien CARME, Joachim NIEHREN et Marc TOMMASI : Querying unranked trees with stepwise tree automata. *In 15th International Conference on Rewriting Techniques and Applications*, 2004. *Cité pages 11 et 35.*

Pascal CARON et Djelloul ZIADI : Characterization of Glushkov automata. *Theoretical Computer Science*, 233(1-2):75–90, février 2000. *Cité page 24.*

Stefano CERI, Georg GOTTLOB et Letizia TANCA : What you always wanted to know about Datalog (and never dared to ask). *IEEE Transactions on Knowledge and Data Engineering*, 1(1):146–166, mars 1989. *Cité page 62.*

Jean-Marc CHAMPARNAUD et Djelloul ZIADI : Canonical derivatives, partial derivatives and finite automaton constructions. *Theoretical Computer Science*, 289(1):137–163, octobre 2002. *Cité page 24.*

Chia-Hsiang CHANG et Robert PAIGE : From regular expressions to DFA's using compressed NFA's. *Theoretical Computer Science*, 178(1-2):1–36, mai 1997. *Cité pages 24 et 25.*

Boris CHIDLOVSKII : Wrapping web information providers by transducer induction. *In 12th European Conference on Machine Learning*, 2001. *Cité pages 6 et 167.*

James CLARK : XSL transformations (XSLT) version 1.0. W3C Recommendation, 16 novembre 1999. URL : `http://www.w3.org/TR/1999/REC-xslt-19991116`. *Cité pages 3 et 99.*

James CLARK et Steve DEROSE : XML path language (XPath) version 1.0. W3C Recommendation, 16 novembre 1999. URL : `http://www.w3.org/TR/1999/REC-xpath-19991116`. *Cité pages 3 et 99.*

Edgar Frank CODD : Extending the database relational model to capture more meaning. *ACM Transactions on Database Systems*, 4(4):397–434, décembre 1979. *Cité page 46.*

William W. COHEN, Matthew HURST et Lee S. JENSEN : A flexible learning system for wrapping tables and lists in HTML documents. *In 11th International Conference on World Wide Web*, 2002. *Cité pages 6, 111 et 167.*

Dario COLAZZO, Giorgio GHELLI et Carlo SARTIANI : Efficient asymmetric inclusion between regular expression types. *In 12th International Conference on Database Theory*, 2009. *Cité page 59.*

Hubert COMON, Max DAUCHET, Remi GILLERON, Florent JACQUEMARD, Denis LUGIEZ, Christof LÖDING, Sophie TISON et Marc TOMMASI : Tree automata techniques and applications. URL : `http://tata.gforge.inria.fr/`. octobre 2007. *Cité pages 3, 4, 7, 12, 30, 32, 33, 34, 42, 47, 56 et 59.*

François COSTE, Daniel FREDOUILLE, Christopher KERMORVANT et Colin DE LA HIGUERA : Introducing domain and typing bias in automata inference. *In 7th International Colloquium on Grammatical Inference*, 2004. *Cité pages 8, 12, 111 et 121.*

Haskell Brooks CURRY et Robert FEYS : *Combinatory Logic*, volume 1. North-Holland, Amsterdam, 1958. *Cité page 29.*

Evgeny DANTSIN, Thomas EITER, Georg GOTTLOB et Andrei VORONKOV : Complexity and expressive power of logic programming. *ACM Computing Surveys*, 33(3):374–425, septembre 2001. *Cité page 62.*

Colin DE LA HIGUERA : Characteristic sets for polynomial grammatical inference. *Machine Learning*, 27(2):125–138, mai 1997. *Cité pages 113 et 114.*

Colin DE LA HIGUERA : A bibliographical study of grammatical inference. *Pattern Recognition*, 38(9):1332–1348, septembre 2005. *Cité pages 7 et 113.*

Colin DE LA HIGUERA : *Grammatical inference*. Cambridge University Press, 2010. *Cité pages 6 et 113.*

Thomas Glen DIETTERICH : Approximate statistical tests for comparing supervised classification learning algorithms. *Neural Computation*, 10(7):1895–1924, octobre 1998. *Cité page 182.*

John DONER : Tree acceptors and some of their applications. *Journal of Computer and System Sciences*, 5(4):406–451, octobre 1970. *Cité pages 30 et 32.*

Andrzej EHRENFEUCHT et Paul ZEIGER : Complexity measures for regular expressions. *Journal of Computer and System Sciences*, 12(2):134–146, avril 1976. *Cité page 24.*

Fabrice ESTIEVENART, Jean-Roch MEURISSE, Jean-Luc HAINAUT et Philippe THIRAN : Semi-automated extraction of targeted data from web pages. *In 22nd International Conference on Data Engineering Workshop*, 2006. *Cité page 5.*

David C. FALLSIDE et Priscilla WALMSLEY : XML Schema part 0: Primer second edition. W3C Recommendation, 28 octobre 2004. URL : `http://www.w3.org/TR/2004/REC-xmlschema-0-20041028/`. *Cité pages 3 et 46.*

Henning FERNAU : Learning XML grammars. *In 2nd International Workshop on Machine Learning and Data Mining in Pattern Recognition*, 2001. *Cité pages 7 et 121.*

Massimo FRANCESCHET : XPathMark: An XPath benchmark for the XMark generated data. *In 3rd International XML Database Symposium*, 2005. *Cité page 172.*

Dayne FREITAG et Nicholas KUSHMERICK : Boosted wrapper induction. *In 17th National Conference on Artificial Intelligence*, 2000. *Cité page 6.*

Dayne FREITAG et Andrew Kachites MCCALLUM : Information extraction with HMMs and shrinkage. *In AAAI Workshop on Machine Learning for Information Extraction*, 1999. *Cité page 6.*

Markus FRICK, Martin GROHE et Christoph KOCH : Query evaluation on compressed trees (extended abstract). *In 18th Annual IEEE Symposium on Logic in Computer Science*, 2003. *Cité pages 4 et 29.*

Pedro GARCÍA : Learning k-testable tree sets from positive data. Rapport technique DSIC-II/46/93, Universidad Politécnica de Valencia, 1993. *Cité pages 7 et 121.*

Pedro GARCÍA et Jose ONCINA : Inference of recognizable tree sets. Rapport technique DSIC-II/47/93, Universidad de Alicante, 1993. *Cité pages 7, 9, 12, 113 et 115.*

Pedro GARCÍA et Enrique VIDAL : Inference of k-testable languages in the strict sense and application to syntactic pattern recognition. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 12(9):920–925, septembre 1990. *Cité page 7.*

Rémi GILLERON, Patrick MARTY, Marc TOMMASI et Fabien TORRE : Interactive tuples extraction from semi-structured data. *In 2006 IEEE/WIC/ACM International Conference on Web Intelligence*, 2006. *Cité pages 6 et 111.*

Victor Mikhaylovich GLUSHKOV : The abstract theory of automata. *Russian Mathematical Surveys*, 16(5):1–53, 1961. *Cité page 24.*

E. Mark GOLD : Language identification in the limit. *Information Control*, 10(5):447–474, mai 1967. *Cité pages 7, 109, 112 et 113.*

E. Mark GOLD : Complexity of automaton identification from given data. *Information Control*, 37(3):302–320, juin 1978. *Cité pages 7 et 113.*

Georg GOTTLOB, Erich GRÄDEL et Helmut VEITH : Datalog LITE: a deductive query language with linear time model checking. *ACM Transactions on Computational Logic*, 3 (1):42–79, janvier 2002. *Cité page 62.*

Georg GOTTLOB et Christoph KOCH : Monadic queries over tree-structured data. *In 17th Annual IEEE Symposium on Logic in Computer Science*, 2002. *Cité page 100.*

Georg GOTTLOB et Christoph KOCH : Monadic Datalog and the expressive power of languages for Web information extraction. *Journal of the ACM*, 51(1):74–113, janvier 2004. *Cité pages 3 et 100.*

Aarti GUPTA : Formal hardware verification methods: A survey. *Formal Methods in System Design*, 1(2-3):151–238, octobre 1992. *Cité page 59.*

Ferenc GÉCSEG et Magnus STEINBY : *Tree Automata*. Akadémiai Kiadó, Budapest, 1984. *Cité page 59.*

John Edward HOPCROFT : An $n \log n$ algorithm for minimizing states in a finite automaton. Rapport technique CS-TR-71-190, Stanford University, 1971. *Cité page 23.*

John Edward HOPCROFT et Jeffrey David ULLMAN : *Introduction to Automata Theory, Languages and Computation*. Addison-Wesley Longman Publishing Co., 1re édition, 1979. *Cité page 19.*

Haruo HOSOYA et Benjamin C. PIERCE : Regular expression pattern matching for XML. *Journal of Functional Programming*, 13(6):961–1004, novembre 2003. *Cité page 52.*

Haruo HOSOYA, Jérôme VOUILLON et Benjamin C. PIERCE : Regular expression types for XML. *ACM Transactions on Programming Languages and Systems*, 27(1):46–90, janvier 2005. *Cité page 60.*

Chun-Nan HSU et Ming-Tzung DUNG : Generating finite-state transducers for semi-structured data extraction from the web. *Information Systems*, 23(8):521–538, décembre 1998. *Cité page 6.*

Lucian ILIE et Sheng YU : Follow automata. *Information and Computation*, 186(1):140–162, octobre 2003. *Cité page 24.*

A. ITAI et J. A. MAKOWSKY : Unification as a complexity measure for logic programming. *Journal of Logic Programming*, 4(2):105–117, juin 1987. *Cité page 63.*

Florent JOUSSE, Rémi GILLERON, Isabelle TELLIER et Marc TOMMASI : Champs conditionnels aléatoires pour l'annotation d'arbres. *In 8e conférence francophone sur l'apprentissage automatique*, 2006. *Cité pages 6 et 111.*

Stephen Cole KLEENE : Automata studies, chapitre Representation of Events in Nerve Nets and Finite Automata, pages 3–41. Princeton University Press, 1956. *Cité page 23.*

Donald E. KNUTH : *The Art of Computer Programming: Fundamental Algorithms*, volume 1. Addison-Wesley Longman Publishing Co., Inc., 3e édition, 1997. *Cité page 25.*

Timo KNUUTILA : Re-describing an algorithm by Hopcroft. *Theoretical Computer Science*, 1-2:333–363, janvier 2001. *Cité page 23.*

Christoph KOCH : Efficient processing of expressive node-selecting queries on XML data in secondary storage: A tree automata-based approach. *In 29th International Conference on Very Large Data Bases*, 2003. *Cité page 29.*

Raymond KOSALA, Maurice BRUYNOOGHE, Jan VAN DEN BUSSCHE et Hendrik BLOCKEEL : Information extraction from web documents based on local unranked tree automaton inference. *In 18th International Joint Conference on Artificial Intelligence*, 2003. *Cité pages 6, 7, 111 et 167.*

Raymondus KOSALA : *Information Extraction by Tree Automata Inference*. Thèse de doctorat, K.U. Leuven, juillet 2003. *Cité page 112.*

Nicholas KUSHMERICK : *Wrapper Induction for Information Extraction*. Thèse de doctorat, University of Washington, 1997. *Cité page 6.*

Nicholas KUSHMERICK : Wrapper induction: Efficiency and expressiveness. *Artificial Intelligence*, 118(1-2):15–68, avril 2000. *Cité page 6.*

Nicholas KUSHMERICK : Finite-state approaches to web information extraction. *In 3rd Summer Convention on Information Extraction*, 2002. *Cité pages 6 et 111.*

Bernard LAMBEAU, Christophe DAMAS et Pierre DUPONT : State-merging DFA induction algorithms with mandatory merge constraints. *In 9th International Colloquium on Grammatical Inference*, 2008. *Cité page 121.*

Dongwon LEE et Wesley W. CHU : Comparative analysis of six XML schema languages. *ACM SIGMOD Record*, 29(3):76–87, septembre 2000. *Cité page 46.*

Aurélien LEMAY, Sebastian MANETH et Joachim NIEHREN : A learning algorithm for top-down XML transformations. *In 29th ACM SIGMOD-SIGACT-SIGART Symposium on Principles of Database Systems*, 2010. *Cité page 200.*

Aurélien LEMAY, Joachim NIEHREN et Rémi GILLERON : Learning n-ary node selecting tree transducers from completely annotated examples. *In 8th International Colloquium on Grammatical Inference*, 2006. *Cité page 200.*

Ling LIU, Calton PU et Wei HAN : XWRAP: An XML-enabled wrapper construction system for web information sources. *In 16th International Conference on Data Engineering*, 2000. *Cité page 5.*

Wim MARTENS : *Static Analysis of XML Transformation and Schema Languages*. Thèse de doctorat, Universiteit Hasselt, 2006. *Cité page 3.*

Wim MARTENS, Frank NEVEN et Thomas SCHWENTICK : Complexity of decision problems for simple regular expressions. *In 29th International Symposium on Mathematical Foundations of Computer Science*, 2004. *Cité page 59.*

Wim MARTENS, Frank NEVEN et Thomas SCHWENTICK : Which XML schemas admit 1-pass preorder typing? *In 10th International Conference on Database Theory*, 2005. *Cité page 54.*

Wim MARTENS, Frank NEVEN, Thomas SCHWENTICK et Geert Jan BEX : Expressiveness and complexity of XML Schema. *ACM Transactions on Database Systems*, 31(3):770–813, septembre 2006. *Cité pages 3, 46, 53, 61 et 92.*

Wim MARTENS et Joachim NIEHREN : On the minimization of XML schemas and tree automata for unranked trees. *Journal of Computer and System Sciences*, 73(4):550–583, janvier 2007. *Cité pages 42 et 56.*

Maarten MARX : Conditional XPath. *ACM Transactions on Database Systems*, 30(4):929–959, décembre 2005. *Cité pages 3 et 99.*

Wolfgang MAY : Information extraction and integration with FLORID: The MONDIAL case study. Rapport technique 131, Universität Freiburg, Institut für Informatik, décembre 1999. *Cité page 175.*

R. MCNAUGHTON et H. YAMADA : Regular expressions and state graphs for automata. *IEEE Transactions on Electronic Computers*, EC-9(1):39–47, mars 1960. *Cité page 24.*

Tova MILO, Dan SUCIU et Victor VIANU : Typechecking for XML transformers. *Journal of Computer and System Sciences*, 66(1):66–97, février 2003. *Cité pages 29 et 59.*

Michel MINOUX : LTUR: A simplified linear-time unit resolution algorithm for horn formulae and computer implementation. *Information Processing Letters*, 29(1):1–12, septembre 1988. *Cité page 63.*

Tom MITCHELL : The need for biases in learning generalizations. *In Readings in Machine Learning*. Morgan Kaufmann Publishers Inc., 1990. *Cité pages 7 et 111.*

Tom MITCHELL : *Machine Learning*. McGraw Hill, 1997. *Cité page 112.*

Stephen MUGGLETON et Luc DE RAEDT : Inductive logic programming: Theory and methods. *Journal of Logic Programming*, 19/20:629–679, 1994. *Cité page 111.*

Makoto MURATA, Dongwon LEE, Murali MANI et Kohsuke KAWAGUCHI : Taxonomy of XML schema languages using formal language theory. *ACM Transactions on Internet Technology*, 5(4):660–704, novembre 2005. *Cité pages 3, 7, 46, 47, 51 et 53.*

Ion MUSLEA, Steven MINTON et Craig A. KNOBLOCK : Hierarchical wrapper induction for semistructured information sources. *Autonomous Agents and Multi-Agent Systems*, 4 (1-2):93–114, 2001. *Cité pages 6 et 167.*

John R. MYHILL : Finite automata and the representation of events. Rapport technique WADC TR-57-624, Wright-Paterson Air Force Base, 1957. *Cité page 23.*

Anil NERODE : Linear automaton transformations. *Proceedings of the American Mathematical Society*, 9(4):541–544, août 1958. *Cité page 23.*

Frank NEVEN : Automata, logic, and XML. *In 16th International Workshop on Computer Science Logic*, 2002a. *Cité pages 30 et 38.*

Frank NEVEN : Automata theory for XML researchers. *ACM SIGMOD Record*, 31(3):39–46, septembre 2002b. *Cité pages 30 et 38.*

Frank NEVEN et Thomas SCHWENTICK : Query automata over finite trees. *Theoretical Computer Science*, 275(1-2):633–674, mars 2002. *Cité page 4.*

Joachim NIEHREN, Laurent PLANQUE, Jean-Marc TALBOT et Sophie TISON : N-ary queries by tree automata. *In 10th International Workshop on Database Programming Languages*, 2005. *Cité pages 3, 100 et 105.*

Jose ONCINA et Pedro GARCÍA : Identifying regular languages in polynomial time. *In Advances in Structural and Syntactic Pattern Recognition*. World Scientific Publishing Co., Inc., 1992. *Cité pages 7, 113 et 115.*

Jose ONCINA et Miguel Angel VARÓ : Using domain information during the learning of a subsequential transducer. *In 3rd International Colloquium on Grammatical Interference*, 1996. *Cité page 121.*

Faïssal OUARDI : *Expressions rationnelles et automates réduits*. Thèse de doctorat, Université de Rouen, mars 2007. *Cité page 24.*

Yannis PAPAKONSTANTINOU et Victor VIANU : DTD inference for views of XML data. *In 19th ACM SIGMOD-SIGACT-SIGART Symposium on Principles of Database Systems*, 2000. *Cité pages 7, 51 et 167.*

Leonard PITT : Inductive inference, DFAs, and computational complexity. *In 2nd International Workshop on Analogical and Inductive Inference*, 1989. *Cité pages 7 et 113.*

Michael O. RABIN : Decidability of second-order theories and automata on infinite trees. *Transactions of the American Mathematical Society*, 141:1–35, juillet 1969. *Cité page 29.*

Stefan RAEYMAEKERS : *Information Extraction from Web Pages based on Tree Automata Induction*. Thèse de doctorat, K.U. Leuven, janvier 2008. *Cité pages 112 et 168.*

Stefan RAEYMAEKERS, Maurice BRUYNOOGHE et Jan VAN DEN BUSSCHE : Learning (k, l)-contextual tree languages for information extraction from web pages. *Machine Learning*, 71(2-3):155–183, juin 2008. *Cité pages 6, 7, 111, 121 et 167.*

Arnaud SAHUGUET et Fabien AZAVANT : Building light-weight wrappers for legacy web data-sources using W4F. *In 25th International Conference on Very Large Databases*, 1999. *Cité page 5.*

Yasubumi SAKAKIBARA : Recent advances of grammatical inference. *Theoretical Computer Science*, 185(1):15–45, octobre 1997. *Cité pages 7 et 113.*

Albrecht SCHMIDT, Florian WAAS, Martin KERSTEN, Michael J. CAREY, Ioana MANOLESCU et Ralph BUSSE : XMark: A benchmark for XML data management. *In 28th International Conference on Very Large Data Bases*, 2002. *Cité page 170.*

Albrecht SCHMIDT, Florian WAAS, Martin KERSTEN, Daniel FLORESCU, Ioana MANOLESCU, Michael J. CAREY et Ralph BUSSE : The XML benchmark project. Rapport technique INS-R0103, Centrum voor Wiskunde en Informatica, 30 avril 2001. *Cité page 170.*

Thomas SCHWENTICK : Automata for XML—a survey. *Journal of Computer and System Sciences*, 73(3):289–315, mai 2007. *Cité pages 3, 30, 38 et 47.*

Helmut SEIDL : Deciding equivalence of finite tree automata. *SIAM Journal on Computing*, 19(3):424–437, juin 1990. *Cité pages 11 et 59.*

Tadahiro SUDA et Haruo HOSOYA : Non-backtracking top-down algorithm for checking tree automata containment. *In 10th International Conference on Implementation and Applications of Automata*, 2005. *Cité page 60.*

James W. THATCHER et Jesse B. WRIGHT : Generalized finite automata theory with an application to a decision problem of second-order logic. *Mathematical Systems Theory*, 2 (1):57–81, mars 1968. *Cité pages 3, 30, 32 et 100.*

Ken THOMPSON : Regular expression search algorithm. *Communications of the ACM*, 11 (6):419–422, juin 1968. *Cité page 24.*

Akihiko TOZAWA et Masami HAGIYA : XML schema containment checking based on semi-implicit techniques. *In 8th International Conference on Implementation and Application of Automata*, 2003. *Cité pages 59 et 60.*

Eric VAN DER VLIST : *Relax NG*. O'Reilly & Associates, 2003. URL : http://books.xmlschemata.org/relaxng/. *Cité pages 47 et 52.*

Victor VIANU : A web odyssey: from Codd to XML. *In 20th ACM SIGMOD-SIGACT-SIGART Symposium on Principles of Database Systems*, 2001. *Cité page 1.*

J. VIRÁGH : Deterministic ascending tree automata I. *Acta Cybernetica*, 5(1):33–42, 1980. *Cité page 34.*

Norman WALSH, Alex MILOWSKI et Henry S. THOMPSON : XProc: An XML pipeline language. W3C Candidate Recommendation, 28 mai 2009. URL : http://www.w3.org/TR/2009/CR-xproc-20090528/. *Cité page 99.*

Bruce W. WATSON : A taxonomy of finite automata construction algorithms. Rapport technique 93/43, Eindhoven University of Technology, 1993a. *Cité page 24.*

Bruce W. WATSON : A taxonomy of finite automata minimization algorithms. Rapport technique 93/44, Eindhoven University of Technology, 1993b. *Cité page 23.*

Djelloul ZIADI, Jean-Luc PONTY et Jean-Marc CHAMPARNAUD : Passage d'une expression rationnelle à un automate fini non-déterministe. *The Bulletin of the Belgian Mathematical Society*, 4(1):177–203, 1997. *Cité pages 24 et 25.*

Index

alphabet, 19
arité bornée, 26
arité non bornée, 28
taille, 19
annotation, 102
apprenabilité
automates d'arbres, 114
requêtes, 118
arbre
annoté, 102
par une requête, 103
arité bornée, 26
arité non bornée, 28
binaire, 27
complet, 136
élagué, 136
consistant, 137
F-complétion, 137
hauteur, 27
partiellement annoté, 144
consistant avec un requête, 144
projection, 102
arbres
égalité, 27
élagués
F-compatibles, 137
arité, 26
automate
arbres (d'), 31
complet, 32
descendant
déterministe, 34
déterministe
ascendant, 32
évaluation (fonction d'), 31
factorisé, 43
frère-fils, 38, 40
hedge, 41
initial, 115
jonction, 139
mots, 21
productif, 64
produit, 66
projection, 102
stepwise, 35, 40
taille, 31
automates
équivalents, 31

chaîne, voir mot
clause, 62
clôture
par chemin, 34
préfixe, 26

codage
- curryfié, 29
- frère-fils, 29

composante, 102
constante
- Datalog clos, 62

contexte, 27
couverture, 181
curryfication, voir codage curryfié

Datalog
- programme clos, 62

domaine
- arbre, 27
- requête, 103

DTD, 47
- déterministe, 48
- étendue, 51

échantillon, 114
- taille, 114

EDTD, voir DTD étendue
élagage, 135
- conservatif, 161
- étendu, 160
- F-élagage, 136
- identité, 158
- régulier, 150, 158
  - D-élagage, 159
  - $\top$-élagage, 160

état
- accessible, 64
- coaccessible, 64
- sorte, 43

états
- interdits, 69

étiquette, 27
exemple, 114
- complètement annoté, 117
- négatif, 114
- positif, 114

expression régulière, 19
- déterministe, 21
- taille, 20

extension (opérateur d'), 29

F-mesure, 181
feuille, 27
fils, 27
fonctionnalité, voir langage fonctionnel
frère, 27
fusions horizontales, 131

haie, 28

jeu de données, 169

langage
- arbres d'arité bornée, 27
- arbres d'arité non bornée, 28
- automate, 31
- DTD, 48
- fonctionnel, 104
- mots, 19
- non-nul, 104
- pseudo-fonctionnel, 138
- réguliers (arbres), 32
- requête, 103
- *stepwise*, 36, 40

littéral, 62

mot, 19
- vide, 19

nœud, 27
ancêtre, 27
père, 27

plus petit point fixe, 62
point fixe, voir plus petit point fixe
position, voir nœud
prédicat, 62
précision, 181
pseudo-fonctionnalité, voir langage pseudo-fonctionnel

racine, 27
rappel, 181
règles
binaires, 35
constantes, 35
ϵ-règles, 31
requête
consistante avec un schéma, 122
invariante, voir requête stable
monadique, 103
stable, 151

sous-arbre, 27
stepwise, voir automate
symbole
d'élagage, 135
initial (DTD), 47

transducteur de sélection de nœuds, 104
trou, voir contexte