Université des Sciences et Technologies de Lille – Lille 1

Département de formation doctorale en informatique **École doctorale SPI Lille**
UFR IEEA

Programmation Logique Inductive pour la classification et la transformation de documents semi-structurés

THÈSE

présentée et soutenue publiquement le 17 Juillet 2014

pour l'obtention du

Doctorat de l'Université des Sciences et Technologies de Lille

(spécialité informatique)

par

Jean Decoster

Composition du jury

Rapporteurs :	M. Antoine Cornuéjols	AgroParisTech
	Mme Céline Rouveirol	Université Paris-Nord
Directeur de thèse :	M. Rémi Gilleron	Université de Lille 3
Co-Encadreurs de thèse :	M. Fabien Torre	Université de Lille 3
	M. Sławek Staworko	Université de Lille 3

Laboratoire d'Informatique Fondamentale de Lille — UMR USTL/CNRS 8022
INRIA Lille - Nord Europe

Numéro d'ordre: 41457

Table des matières

Table des figures

Liste des tableaux

Chapitre 1

Introduction

À l'heure du tout-connecté et du tout-communicant, il est devenu habituel, voire indispensable, d'échanger et de partager avec son prochain, mais également d'être informé en permanence des événements mondiaux, nationaux ou bien encore familiaux. Cet échange d'informations constant a soulevé de nombreux problèmes de par la quantité d'informations transmises et acheminées aux différents protagonistes ainsi que par l'hétéroclisme des moyens utilisés pour partager et prendre connaissance de ces informations. Afin d'être capable de traiter ces flux de données et de les représenter, un certain nombre de standards ont été développés. Les données semi-structurées, au travers de formats comme le XML [Bray 2008], ont alors pris part à résoudre ce problème et sont devenues en peu de temps un standard d'échange utilisé et utilisable sur la plupart des outils de communication.

La structure d'un document XML reflète les relations logiques, sémantiques ou syntaxiques entre les différentes parties de ce document. Afin d'être valide du point de vue du XML, il respecte un ensemble de règles appelé *schéma*. Ces règles régissent sa forme et sont propres à l'utilisation du document ou au corpus auquel il appartient. Ainsi, les documents d'une même source d'information respectent généralement le même schéma. L'exemple 1 ci-dessous représente un annuaire sous la forme d'un document XML. Ce dernier contient deux informations, un nom (`Dixon`) et un prénom (`Daryl`), organisées à l'aide de balises, appelées éléments, que sont `Annuaire`, `Amis`, `Personne`, `Nom` et `Prénom`. Il spécifie que la personne nommée `Daryl Dixon` est présente dans cet annuaire comme `Amis` et `Collègues`. Un autre agenda contenant une ou plusieurs personnes peut être construit en suivant la même organisation.

Exemple 1 (Document XML représentant un annuaire)**.**

```
<Annuaire>
        <Amis>
             <Personne>
                     <Nom> Dixon  </Nom>
                     <Prénom> Daryl </Prénom>
             <Personne>
        </Amis>
        <Collègues>
             <Personne>
                      <Nom> Dixon </Nom>
                      <Prénom> Daryl </Prénom>
             <Personne>
        </Collègues>
</Annuaire>
```

De nombreuses applications utilisant le format XML ont vu le jour. En général, ces logiciels combinent des données semi-structurées à des langages permettant d'effectuer des traitements intelligents. Certains d'entre eux, du fait du nombre croissant de documents,

sont devenus des pré-requis pour utiliser le potentiel de ces données. Parmi ces traitements, on retrouve la classification et la transformation automatiques de documents XML.

La classification automatique de documents XML a pour objet la détection automatique de catégories, appelées *classes*, à partir de la structure des documents XML alors appelés *exemples*. Le groupement de plusieurs documents au sein d'une même classe suppose des points communs comme, par exemple, une même utilisation ou des informations communes. Cette tâche trouve plusieurs applications possibles. La plus naturelle consiste à réorganiser les documents d'un corpus en plusieurs catégories. Dans le cas d'un corpus de documents où chaque exemple représente une personne ou une entreprise, la classification peut servir à différencier les personnes des entreprises en formant deux classes. Ce concept de classification peut être étendu au cas de la recherche d'information dans un document. Ainsi, une classification serait utilisée afin de trouver des documents présentant certaines propriétés. Dans le cas d'une recherche d'information faite sur Internet, elle peut, par exemple, être utilisée pour sélectionner des documents contenant certaines informations ou provenant de certaines sources d'information.

Une transformation de documents XML en documents XML est une fonction d'un ensemble de documents XML vers un autre ensemble de documents XML. Elle est habituellement définie manuellement en utilisant des langages complexes comme XSLT [Bray 2006]. Bien que des outils soient développés et existent pour aider les utilisateurs dans l'écriture de programmes de conversion, ce processus n'en demeure pas moins complexe et nécessite des compétences d'expert. Il est donc inadapté à des situations où les sources d'informations sont nombreuses et changent fréquemment. Cette émergence d'applications et de documents d'origines diverses met ainsi en avant l'importance d'un processus qui permettrait l'apprentissage automatique de transformations d'un format vers un autre. C'est ce que permet l'apprentissage de transformations qui a alors pour objectif d'apprendre automatiquement à transformer des documents XML en d'autres documents XML. L'une des utilisations possibles de cette opération est l'adaptation de données à une application. Elle permet ainsi à des applications différentes d'être capables de communiquer, malgré des schémas différents, en recevant des données qui leur sont adaptées. Une autre utilisation possible concerne la gestion de l'affichage de données sur des périphériques. En effet, avec la multitude de supports destinés à une application, il est devenu intéressant d'adapter à la volée des données à un support.

Afin de réaliser ces traitements, nous avons choisi la Programmation Logique Inductive [Muggleton 1991, Muggleton 1994b]. La PLI est une technique d'apprentissage relationnel. Le choix de l'utiliser a été dicté par la présence, dans les documents XML, de relations implicites aux données. Elles sont, par exemple, présentes à la figure 1.1 contenant la représentation arborescente du document XML de l'exemple 1. Dans cet arbre, sont utilisées plusieurs fois les données Daryl et Dyxon. Cependant, du fait de la représentation arborescente, aucun lien direct n'est établi entre ces nœuds malgré un lien évident. L'apprentissage relationnel, comme son nom l'indique, permet d'exhiber ces relations entre les différents éléments d'un document. Elle s'appuie principalement sur l'usage de la logique du premier ordre [Heijenoort 1967, Alfred North Whitehead 1912, Tarski 1956]. Cette dernière, utilisée dans ce paradigme, permet la formulation de règles, appelées *clauses*, au travers de la définition d'un langage. Elle dispose également de briques algorithmiques, comme des relations de subsomption, nécessaires à la conception d'un algorithme d'apprentissage. Une description de chaque problème d'apprentissage est ainsi réalisable grâce à un langage du premier ordre. Notre intérêt pour ce cadre d'apprentissage est enfin renforcé par le lien existant entre un problème en PLI et un problème de classification binaire. Ces deux problèmes présentent, en effet, de nombreuses similitudes. Le passage de l'un à l'autre en est ainsi facilité.

L'utilisation de la PLI comporte cependant des inconvénients. L'expressivité du langage

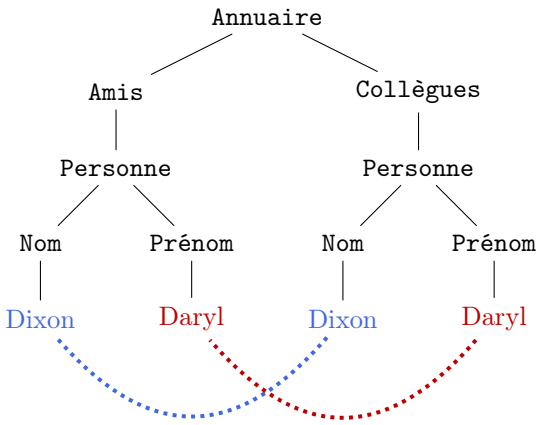


FIGURE 1.1 – Arbre XML construit à partir du document XML de l'exemple 20.

utilisé pour décrire un problème en PLI en est l'un des principaux. En effet, le choix du langage influe directement sur le nombre de solutions possibles d'un problème. Ce nombre est généralement important (exponentiel, voire infini) pour un concept à apprendre. Une approche par énumération n'est pas envisageable. Le second point critique d'un système de PLI est la relation de subsomption utilisée pour guider la recherche dans cet ensemble de solutions. Elle indique si une solution est plus générale, équivalente ou incomparable à une autre. La relation de subsomption la plus naturelle est l'implication logique utilisée dans la définition d'un problème en PLI [Muggleton 1991, Muggleton 1994b]. Elle est indécidable pour les clauses [Schmidt-Schauß 1988] et clauses de Horn [Marcinkowski 1992]. Le défi d'un système de PLI est alors de trouver un moyen efficace de parcourir l'espace des solutions afin de trouver la meilleure possible. Cette recherche repose principalement sur la définition du langage utilisé pour décrire les exemples et les solutions ainsi que sur une relation de subsomption adaptée à ce langage.

De nombreuses restrictions pour les langages ont été proposées afin de réduire le nombre de solutions lors d'une tâche de classification. Nous retrouvons, par exemple, la notion de clauses déterminées [Quinlan 1991, Muggleton 1990, Kietz 1994a]. Les clauses déterminées ont pour principal intérêt de permettre une relation de subsomption polynomiale dans la taille de l'entrée. Ceci se fait au détriment d'une expressivité qui est limitée rendant son usage pour certains problèmes impossible comme la représentation de propriétés chimiques de molécules [Muggleton 2010]. Cette restriction a donné lieu à de nombreuses variantes comme les clauses i, j-déterminées [Muggleton 1990], les k-locales [Kietz 1994a]. Ces relations sont parfois plus expressives mais ne permettent pas la définition d'une subsomption dont la complexité est attrayante. Sans de telles restrictions, un système de PLI peut produire des solutions aberrantes dans le cas général, ne produire aucune solution dans le pire des cas ou nécessite des temps de calcul trop élevés dans le meilleur. En complément, des relations de subsomption propres à ces restrictions ont été définies. Parmi ces relations, on retrouve la θ-subsomption [Plotkin 1970], l'OI-subsomption [Semeraro 1994, Geibel 1997]. Elles bénéficient généralement d'une complexité élevée, la θ-subsomption et l'OI-subsomption sont NP-complet [Plotkin 1970, Malerba 2001] mais demeurent utilisables contrairement à l'implication logique. Ces restrictions et relations de subsomption ont été utilisées afin de définir des systèmes d'apprentissage. On retrouve par exemple : Progol [Muggleton 1995], ProGolem [Muggleton 2010], Alpeh [Srinivasan 2004], April [Santos 2003], HOC-learner [Santos 2009], FILP [Bergadano 1993b] et FFOIL [Quinlan 1996]. Certains de

ces systèmes, comme FILP, FFOIL et HOC-learner, sont capables d'apprendre des programmes fonctionnels. Un programme fonctionnel est assimilable à une composition de fonctions. Il est ainsi réductible à la représentation d'une transformation. Nous opterons pour cette approche afin d'étudier le problème d'apprentissage de transformations.

Le but de cette thèse est double. Le premier objectif est de proposer un framework d'apprentissage pour la classification de documents XML. Le second est la définition d'algorithmes pour l'apprentissage de transformations de documents XML.

Nous définissons en premier lieu le framework pour la classification de documents XML. La classification automatique de documents XML consiste à apprendre une solution capable d'associer à chaque exemple la classe correspondante. Une solution doit être ainsi assez générale pour expliquer tous les exemples de sa classe mais pas assez pour décrire des exemples d'autres classes. Cette notion de généralisation trouve une existence en logique du premier ordre sous le nom de *moindre généralisation*. L'existence du moindre généralisé pour un ensemble de clauses n'est cependant pas garantie pour tout langage de clauses. Si l'existence de ce dernier est avéré, il peut demeurer exponentiellement grand dans le nombre d'exemples généralisés. Il représente ainsi un premier problème que nous devrons solutionner. Comme mentionné précédemment, le choix de la relation de subsomption est également un facteur déterminant lors d'un apprentissage en PLI. Notre décision s'est portée sur l'usage de la θ-subsomption. Ce choix se justifie par l'équivalence entre la θ-subsomption et de l'implication logique pour la classe des clauses non-récursives et l'existence de restrictions sur les clauses permettant la définition de tests de θ-subsomption plus efficaces. Ainsi, le framework que nous proposons est basé sur la θ-subsomption comme relation de subsomption et le calcul du moindre généralisé comme opération de généralisation. L'élaboration de ce framework pour les documents XML passe par la conception et l'étude de plusieurs langages de clauses capables de représenter des structures arborescentes. Nous avons ainsi défini plusieurs langages : $\mathcal{L}^{T_\Sigma}_{child}$, $\mathcal{L}^{T_\Sigma}_{fcns}$ et $\mathcal{L}^{T_\Sigma}_{pc}$ capables de représenter des arbres. Pour chacun de ces langages de clauses est proposée une réécriture des opérations de base de l'apprentissage relationnel que sont la θ-subsomption et le calcul de moindre généralisé. Ces réécritures sont adaptées à notre tâche et à nos langages et sont en conséquence plus efficaces que les algorithmes d'autres familles de clauses. Ainsi, là où les opérations standards ont des complexités en temps exponentielles dans la taille de leur entrée, nous en proposons de complexité polynomiale en temps. Nous nous sommes ensuite placés dans une démarche de généralisation de nos classes. Ainsi, pour chaque algorithme et langage définis, nous avons proposé de nouvelles familles de clauses incluant ces langages. Nous avons ainsi établi les familles $\mathcal{F}_{sop}, \mathcal{F}_{\mathsf{QD}}, \mathcal{F}_{\mathsf{MQD}}$ et $\mathcal{F}_{\mathsf{FDP}}$. Nous avons ensuite montré que nos langages de clauses, capables de représenter des arbres, sont apprenables dans le paradigme de l'identification à la limite [Gold 1967]. Enfin, des expériences ont été réalisées sur des corpus de données réelles ou non afin de tester leur efficacité à solutionner notre problème de classification de documents XML. Ainsi, nous avons défini des algorithmes et langages constituant un framework polynomial d'apprentissage basé sur l'utilisation du moindre généralisé et de la θ-subsomption.

Notre seconde contribution porte sur l'élaboration d'une méthode pour l'apprentissage de transformations de documents XML. Elle commence par la modélisation d'un langage de clauses capable de représenter des transformations d'éléments d'un ensemble donné vers d'autres de ce même ensemble. Nous avons ainsi défini un type de clauses appelé *clause de transformation*. Ces clauses sont une adaptation de la notion de script d'édition, i.e. une composition d'opérations d'édition élémentaires, à la PLI. Un parallèle a ainsi été fait entre une opération d'édition d'un script d'édition et un atome de ce type de clause, appelé *atome de transformation*. Afin d'outrepasser l'expressivité imposée par les scripts d'édition, nous avons introduit, dans ces dernières, la notion de contexte à l'aide d'atomes

appelés *atomes de contexte*. Ces atomes ont pour objectif de donner des informations sur les éléments manipulés. Ainsi, dans une clause de transformation, chaque élément susceptible d'être transformé est associé à un contexte fini le décrivant. Ces informations permettent de paramétrer dynamiquement les opérations de transformation définies par des atomes de transformation. Ces clauses se placent dans la lignée des programmes fonctionnels [Paulson 1991, Bergadano 1993b, Bergadano 1993a, Hernández-orallo 1999, Quinlan 1996, Santos 2009]. La représentation d'une transformation établie, nous avons ensuite proposé deux algorithmes d'apprentissage de clauses de transformation. Ces algorithmes sont basés sur l'exploration de l'espace de solutions par un algorithme A^* dans le but de trouver une clause de transformation. Le principe de recherche par l'algorithme A^* est assez courant en PLI, on le retrouve, par exemple, dans HOC-Learner [Santos 2009]. La recherche dans l'ensemble des solutions est guidée par heuristique. Dans la plupart des algorithmes de PLI, l'heuristique guidant la recherche correspond à une mesure spécifiant l'efficacité de la solution passée en revue. C'est notamment le cas du *gain* de FOIL [Quinlan 1990, Quinlan 1993, Quinlan 1995] . Nous nous différencions, sur ce point, en choisissant comme heuristique une distance d'édition. Ainsi, cette heuristique est propre à la nature des données manipulées et doit être adaptée en fonction du problème de transformation. Cependant, afin de rendre nos algorithmes les plus génériques possibles, nous avons prouvé qu'une distance d'édition sur un ensemble donné est monotone et admissible pour un problème défini sur cet ensemble. Ainsi, l'efficacité de la recherche est assurée. Le premier algorithme proposé s'appuie sur la recherche d'une transformation commune pour chaque exemple. Les transformations de chaque exemple sont ensuite généralisées afin d'en trouver une commune. Ce procédé est fréquent en PLI et est similaire à celui employé pour résoudre notre tâche de classification. Notre second algorithme consiste en une recherche dirigée par plusieurs exemples. Ainsi, la transformation trouvée n'est plus commune à un exemple mais à un ensemble d'exemples. Elle a pour intérêt de ne plus généraliser plusieurs transformations mais des étapes élémentaires d'une transformation afin d'orienter au mieux le choix des opérations de transformation suivantes. Ces deux méthodes constituent un framework d'apprentissage pour les transformations. Elles ont l'avantage de ne pas être propres à la transformation d'un objet en particulier comme les mots ou les arbres.

Cette thèse est organisée comme suit. Dans le chapitre suivant, nous présentons les notions de base nécessaires à la compréhension de la PLI. Nous retrouvons, entre autres, le vocabulaire de la logique du premier ordre, sa sémantique ainsi que le cas particulier des modèles de Herbrand. Enfin, nous exposons la plus naturelle des relations de subsomption qu'est l'implication logique puis décrivons la résolution et la SLD-résolution.

Nous continuons dans le chapitre 3 avec la présentation des structures de données que sont les mots, les arbres et les graphes. La tâche de classification de documents XML est ensuite introduite formellement puis située par rapport au cadre classique de la PLI [Muggleton 1991]. Enfin, nous présentons le modèle d'apprentissage de l'identification à la limite [Gold 1967] et donnons les critères d'évaluation utilisés lors de nos expériences.

La relation de subsomption est l'un des points critiques d'un système d'apprentissage. Ainsi, dans le chapitre 4, nous faisons un tour d'horizon des principales relations de subsomption utilisées en logique du premier ordre et susceptibles de nous intéresser. Nous les étudions et présentons leurs avantages ainsi que leurs inconvénients. Nous détaillons ensuite les principaux opérateurs de généralisation associés à ces relations. Puis, définissons plusieurs restrictions syntaxiques, couramment utilisées sur les clauses, rendant possible l'apprentissage de programmes logiques.

Le chapitre 5 traite de la tâche de classification d'arbres en PLI. L'étude de ce problème commence par la recherche de représentations logiques d'arbres. Nous présentons ainsi de nouvelles familles de clauses adaptées à la représentation d'arbres et de documents

XML. Pour chacune de ces classes est proposée une réécriture plus efficace des opérations de base de l'apprentissage relationnel que sont la θ-subsomption et le calcul de moindre généralisé. Ces opérations forment un framework d'apprentissage pour chaque classe de clauses et seront utilisées afin d'infirmer ou confirmer une possible identification à la limite [Gold 1967] de ces classes. Enfin, nous réalisons des expériences dans le but d'observer leurs comportements vis-à-vis de jeux de données réelles.

Nous terminons avec le chapitre 6 en présentant le problème de transformation adapté au cadre de la PLI. Nous précisons pour cela la forme des transformations apprises et introduisons la notion de script d'édition. Puis, présentons quelques biais présents dans la littérature dont l'emploi pour notre problème semble propice. Nous nous penchons ensuite sur un problème de transformation basique et montrons la difficulté que représente cette tâche de transformation. Suite à ces observations, nous proposons deux algorithmes qui permettent l'apprentissage de transformations en PLI et testons leurs efficacités expérimentalement. Enfin, nous présentons quelques travaux vis-à-vis desquels nous nous positionnons.

Chapitre 2

Introduction à la Programmation Logique Inductive

Sommaire

Dans ce chapitre, nous allons développer quelques notions de base de la programmation logique. Le lecteur habitué à la syntaxe ainsi qu'aux vocabulaires de la logique du premier ordre et de la programmation logique peut passer ce chapitre. Nous commençons par introduire le vocabulaire et les définitions nécessaires à la compréhension de la programmation logique inductive (PLI ou ILP en anglais pour Inductive Logic Programing). Nous nous focalisons plus particulièrement sur la logique du premier ordre. Historiquement, elle fut introduite par Gottlob Frege [Heijenoort 1967] puis approfondie par Alfred North Whitehead et Bertrand Russel [Alfred North Whitehead 1912]. Sa sémantique a ensuite été développée par Alfred Tarski [Tarski 1956]. Ces définitions sont pour la plupart inspirées et reformulées de [Lloyd 1987, Cornuéjols 2003]. Elles sont les bases de Prolog [Kowalski 1974] ainsi que de la plupart des systèmes de programmation logique inductive. Nous définissons ensuite, d'un point de vue théorique, la notion de modèle de Herbrand et introduisons une relation de subsomption : l'implication logique. Nous rappellons enfin une procédure la testant dans le cas précis des clauses définies : la SLD-résolution.

2.1 Syntaxe de la logique des prédicats

Avant de développer la notion de langage de clauses, il est nécessaire d'introduire un certain nombre de notations et définitions utiles à la syntaxe d'une clause. Ainsi, le vocabulaire de base utilisé en logique des prédicats est le suivant :

Une constante correspond généralement au nom d'un élément spécifique. On utilise la convention de nommage utilisée dans Prolog [Kowalski 1974]. Toute constante commence donc par une lettre minuscule. Ainsi, les constantes *annuaire, personne, td, tr* représentent des objets pouvant qualifier un annuaire, une personne et les balises HTML *td* et *tr*.

Une variable permet de nommer un élément sans l'identifier précisément, contrairement à une constante qui, elle, nomme l'élément que l'on manipule. Par convention, le nom d'une variable commence par une majuscule comme *X, Y, ...*

Un symbole de fonction est défini par un symbole *(f, g, ...)*, écrit en minuscule, ainsi que par une arité ($i, j, ... \in \mathbb{N}$). Ainsi, *f/i, g/j, ...* sont des symboles de fonction. L'arité d'un symbole de fonction est un nombre entier correspondant au nombre d'arguments que peut prendre la fonction. Une constante correspond à une fonction d'arité 0.

Un symbole de prédicat est, comme un symbole de fonction, défini par un symbole et une arité. Le symbole de ce dernier est en minuscule. On peut ainsi définir les symboles de prédicat suivants : *p/m, q/n, ...*

Cinq connecteurs : $\leftarrow$ l'implication (binaire), $\neg$ la négation (unaire), $\wedge$ le "*et*" logique (binaire), $\vee$ le "*ou*" logique (binaire) et $\leftrightarrow$ l'équivalence (binaire).

Deux quantificateurs : $\exists$ (il existe), est le quantificateur existentiel, et $\forall$ (quelque soit), est le quantificateur universel.

Des symboles de ponctuations : "(", ")", "." et ","

Par la suite, nous utilisons le connecteur $\rightarrow$ tel que les notations suivantes ($F \leftarrow G$) et ($G \rightarrow F$) sont équivalentes.

Définition 1 (Terme, Terme complexe et Terme fondé)**.** *Un* terme *est défini récursivement de la manière suivante :*

- *une variable est un terme ;*
- *une constante est un terme ;*
- *un symbole de fonction f/n associé à n arguments $t_1, \cdots, t_n$ étant des termes forment le terme $f(t_1, \cdots, t_n)$.*

Un terme constitué d'un symbole de fonction d'arité supérieure à 1 est appelé terme complexe*. Un terme ne contenant aucune variable est dit* fondé *(*ground *en anglais).*

Exemple 2. *Pour illustrer la définition de terme, nous reprenons l'exemple classique des listes en Prolog. Nous utilisons les symboles suivants :*

- *deux constantes o_1 et o_2, ils représentent deux objets distincts,*
- *la constante nil pour définir une liste vide,*
- *le symbole de prédicat $cons/2$ pour représenter une liste dont le premier argument est la tête de la liste et le second la queue.*
- *une variable X pour représenter un objet quelconque.*

À partir de ces symboles, nous pouvons construire, entre autres, les termes suivants :

- *$cons(o_1, cons(o_2, nil))$ est une liste contenant dans cet ordre les éléments o_1 et o_2,*
- *$cons(X, cons(o_2, nil))$ est une liste dont le premier élément est inconnu et le second est o_2,*
- *$cons(cons(o_1, cons(o_2, nil)), cons(cons(o_2, cons(o_1, nil)), nil))$ est une liste de listes.*

On remarque qu'à partir d'une variable ou/et d'une constante et d'un symbole de prédicat, on peut construire une infinité de nouveaux termes.

Nous notons $f^n(t)$, avec $n \in \mathbb{N}$, n applications de f sur t où t est un terme et f un symbole de fonction, c'est à dire $\underbrace{f(f(\cdots f(t)\cdots))}_{n \text{ fois}}$.

Définition 2 (Atome et Atome fondé)**.** *Si p/n est un symbole de prédicat n-aire et $t_1, \cdots, t_n$ des termes, alors $p(t_1, \cdots, t_n)$ est un* atome*. Un atome est* fondé *si ses termes sont fondés.*

Définition 3 (Littéral, Littéral fondé et Littéraux compatibles)**.** *Un* littéral *ℓ est un atome $A = p(t_1, \cdots, t_n)$, avec $t_1, \cdots, t_n$ des termes, ou bien la négation d'un atome, notée $\neg A$.*

Un littéral est fondé *si son atome est fondé. Deux littéraux ℓ_1 et ℓ_2 sont* compatibles *si et seulement s'ils ont les mêmes symboles de prédicat et sont tous les deux niés ou non niés.*

Rappelons que la double négation d'un atome $\neg\neg A$ est égale à A. Nous serons amenés à utiliser la lettre h pour nommer l'atome d'un littéral positif et la lettre b pour celui d'un littéral négatif.

Définition 4 (Clause, Clause vide, Clause unitaire et Clause fondée)**.** *Une* clause *est une disjonction (finie) de littéraux universellement quantifiés de la forme :*

$$\forall X_1, \cdots, \forall X_k \ (\ell_1 \vee \cdots \vee \ell_m)$$

où $\ell_1, \cdots, \ell_m$ sont des littéraux et $X_1, \cdots, X_k$ les variables apparaissant dans $\ell_1 \vee \cdots \vee \ell_m$. Une clause vide*, notée $\square$, est une clause ne contenant aucun littéral. Une* clause unitaire *est une clause comportant exactement un littéral (positif ou négatif). Une clause est* fondée *si chacun de ses littéraux est fondé.*

Chaque variable présente dans une clause est quantifiée universellement, il est alors possible de réécrire une clause :

$$C = \forall X_1, \cdots, \forall X_k \ (\ell_1 \vee \cdots \vee \ell_m)$$

sous une forme ensembliste :

$$\{\ell_1, \cdots, \ell_m\}$$

Cette représentation ensembliste à l'avantage de bénéficier des notations de la théorie des ensembles comme l'union, l'intersection, ... Une autre représentation intéressante et fortement utilisée est celle introduite par Kowalski dans Prolog [Kowalski 1974]. Cette dernière a l'avantage de distinguer les littéraux positifs de ceux négatifs. Ainsi, une clause :

$$C = \forall X_1, \cdots, \forall X_k \ (h_1 \vee \cdots \vee h_m \vee \neg b_1 \vee \cdots \vee \neg b_n)$$

peut être réécrite sous la forme d'une règle :

$$(h_1 \vee \cdots \vee h_m) \leftarrow (b_1 \wedge \cdots \wedge b_n)$$

Cette transformation est rendue possible grâce à la formule $a \leftarrow b$ correspondant à $a \vee \neg b$. Enfin, comme la notion de clause est destinée à être utilisée dans des algorithmes séquentiels, nous sommes amenés à introduire un ordre entre les littéraux d'une clause afin de la représenter par une séquence d'atomes :

$$h_1, \cdots, h_m \leftarrow b_1, \cdots, b_n.$$

Nous présentons maintenant une notation facilitant la manipulation des variables dans une clause, un littéral, un atome ou un terme.

Définition 5 (Ensemble de variables dans une clause, un littéral, un atome ou un terme)**.** *Soit un terme t, un atome $A = p(t_1, \cdots, t_n)$, un littéral ℓ et une clause $C = \{\ell_1, \cdots, \ell_m\}$. Les* ensembles de variables apparaissant dans *t, A, ℓ* et *C, respectivement notés $vars(t)$, $vars(A)$, $vars(\ell)$ et $vars(C)$, sont définis de la manière suivante :*

$$vars(t) = \begin{cases} \emptyset & \text{si } t \text{ est une constante,} \\ t & \text{si } t \text{ est une variable,} \\ \cup_{i=1}^{m} vars(t_i) & \text{si } t \text{ est un terme de la forme } f(t_1, \cdots, t_m). \end{cases}$$

$$vars(A) = vars(p(t_1, \cdots, t_n)) = \begin{cases} \emptyset & si\ n = 0, \\ \cup_{i=1}^{n} vars(t_i) & sinon. \end{cases}$$

$vars(\ell) = vars(A)$ *si* $\ell = A$ *ou* $\ell = \neg A$.

$$vars(C) = vars(\{\ell_1, \cdots, \ell_m\}) = \begin{cases} \emptyset & si\ m = 0, \\ \cup_{i=1}^{m} vars(\ell_i) & sinon. \end{cases}$$

Exemple 3. *Nous définissons, à l'aide du symbole de fonction* $cons/2$ *et de la constante nil de l'exemple 2, deux clauses* C *et* D *basées sur un prédicat* $last/2$. *Ces clauses représentent une opération retournant le dernier élément d'une liste.*

$$\begin{array}{lll} C & = & \forall X \quad last(cons(X, nil), X) \\ D & = & \forall X \forall Y \forall Z \quad last(cons(X, Y), Z) \vee \neg last(Y, Z) \end{array}$$

La représentation ensembliste de ces clauses est la suivante :

$$\begin{array}{lll} C & = & \{last(cons(X, nil), X)\} \\ D & = & \{last(cons(X, Y), Z), \neg last(Y, Z)\} \end{array}$$

et la représentation Prolog :

$$\begin{array}{lll} C & = & last(cons(X, nil), X) \leftarrow . \\ D & = & last(cons(X, Y), Z) \leftarrow last(Y, Z). \end{array}$$

Enfin, les ensembles de variables pour les clauses C *et* D *sont les suivants :*

$$vars(C) = \{X\} \text{ et } vars(D) = \{X, Y, Z\}$$

La présence de variables dans chaque littéral d'une clause permet d'établir des liens entre les différents littéraux de cette clause. On parle alors de *connexion* ou encore de *clause connectée*. Elle permet d'établir des relations entre littéraux servant parfois à les ordonner.

Définition 6 (Clause connectée)**.** *Deux littéraux* ℓ *et* ℓ' *d'une clause sont* connectés *s'ils partagent une variable commune ou s'il existe un littéral connecté à la fois à* ℓ *et à* ℓ'. *Une clause est* connectée *s'il existe un littéral connecté à tous les autres.*

Une clause est construite à partir d'un ensemble de constantes, d'un ensemble de symboles de fonction, d'un ensemble de prédicats ainsi que de variables. À partir d'un ensemble fini de ces symboles, il est possible de construire un nombre possiblement infini de clauses. Cependant, toutes ces clauses ont comme point commun d'avoir utilisé un ou plusieurs symboles imposés. On dit alors qu'elles appartiennent à un même *langage de clauses.*

Définition 7 (Langage de clauses)**.** *Soit un ensemble de symboles de prédicat, un ensemble de constantes, un ensemble de symboles de fonction et un ensemble infini de variables. Le* langage de clauses défini sur ces ensembles *est l'ensemble de toutes les clauses construites à partir de ces derniers.*

Exemple 4. *Les clauses* C *et* D *présentes dans l'exemple 3 appartiennent au langage de clauses* $\mathcal{L}$ *basé sur l'ensemble de constantes* $\mathcal{C} = \{nil\}$, *l'ensemble de symboles de fonction* $\mathcal{F} = \{cons/2\}$ *et l'ensemble des symboles de prédicat* $\mathcal{P} = \{last/2\}$.

L'arité des symboles utilisés pour un langage est importante. Elle permet de définir la taille d'un terme, d'un atome et d'une clause ainsi que celle d'un ensemble de clauses.

Définition 8 (Taille d'un terme, clause et ensemble de clauses)**.** *Soit un terme t, un atome A, un littéral ℓ, une clause C et un ensemble de clauses T. On note respectivement $|t|$, $|A|$, $|\ell|$, $|C|$ et $||T||$* les tailles d'un terme t, d'un atome A, d'un littéral ℓ, d'une clause C et d'un ensemble de clauses T *définies de la manière suivante :*

$$|t| = \begin{cases} 1 & \textit{si } t \textit{ est une constante ou une variable,} \\ 1 + \cup_{i=1}^{m} |t_i| & \textit{si } t \textit{ est un terme de la forme } f(t_1, \cdots, t_m). \end{cases}$$

$|A| = |p(t_1, \cdots, t_n)| = 1 + \sum_{k=1}^{n} |t_k|$ *si* $0 < n$ *sinon* 1.

$|\ell| = |\neg A| = |A|$

$|C| = |\{\ell_1, \cdots, \ell_n\}| = \sum_{k=1}^{n} |\ell_k|$ *si* $0 < n$*, sinon* 0.

$||T|| = ||\{C_1, \cdots, C_n\}|| = \sum_{k=1}^{n} |C_k|$ *si* $0 < n$*, sinon* 0.

Il est possible de restreindre la taille d'un terme, d'un atome ou d'une clause grâce aux notions de *profondeur* et d'arité.

Définition 9 (Profondeur)**.** *Soit une variable V, une constante a, un terme $f(t_1, \cdots, t_n)$, un atome $p(t_1, \cdots, t_n)$ et une clause $C = h \leftarrow \ell_1, \cdots, \ell_n$. On calcule* la profondeur (*depth*) d'une constante, d'une variable, d'un terme, d'un atome et d'une clause *de la manière suivante :*

$$\begin{array}{lcl} depth(a) & = & 1 \\ depth(V) & = & 0 \\ depth(f(t_1, \cdots, t_n)) & = & 1 + max(\{depth(t_1), \cdots, depth(t_n)\}) \\ depth(p(t_1, \cdots, t_n)) & = & max(\{depth(t_1), \cdots, depth(t_n)\}) \\ depth(C) & = & max(\{depth(h), depth(\ell_1), \cdots, depth(\ell_n)\}) \end{array}$$

Considérer l'arité des symboles d'un langage ainsi que la profondeur des atomes et des termes de ce langage comme constantes permet d'avoir la taille d'un atome ou d'un terme constante. La taille d'une clause correspond alors à son nombre de littéraux, à un facteur constant près. Nous prenons en compte cette simplification lorsque le contexte le permet.

La syntaxe d'une clause ainsi que la notion de langage de clauses définies, nous pouvons maintenant leur y associer un sens. Nous définissons ainsi dans la section suivante ce qu'Alfred Tarski a appelé la *sémantique*.

2.2 La sémantique de la logique des prédicats

Nous avons précédemment défini la syntaxe d'une clause ainsi que celle des éléments qui la composent comme les littéraux, les atomes, les termes, les constantes, les variables ainsi que les connecteurs. La signification de ces éléments est parfois connue comme, par exemple, le "et" logique $\wedge$ ou bien le "ou" logique $\vee$ issus de la logique Booléenne. Cela n'est cependant pas le cas de tous et nous sommes incapables pour le moment d'interpréter les clauses. Ainsi, Alfred Tarski [Tarski 1956] a introduit la notion de *sémantique*. Elle a pour but d'associer à chaque clause une valeur de vérité. Nous retrouvons, dans la notion de sémantique, les notions d'univers, de base et de modèle ainsi que certaines de leurs propriétés. Ces notions sont fortement présentes en ILP et sont utilisées à plusieurs reprises comme fondement de systèmes d'inférence. C'est notamment le cas du système Golem [Muggleton 1990]. Nous nous intéressons plus particulièrement à la notion de modèle, univers et base de Herbrand. Le lecteur intéressé par ces notions peut trouver des explications plus complètes dans [Lloyd 1987, Nienhuys-Cheng 1997a].

Définition 10 (Univers de Herbrand)**.** *Soit $\mathcal{L}$ un langage de clauses. L'*univers de Herbrand de $\mathcal{L}$*, noté $\mathcal{U}(\mathcal{L})$, est l'ensemble de tous les termes fondés construits à partir des symboles de fonction et des constantes*[1] *de $\mathcal{L}$.*

Définition 11 (Base de Herbrand)**.** *Soit $\mathcal{L}$ un langage de clauses. La* base de Herbrand de $\mathcal{L}$*, notée $\mathcal{B}(\mathcal{L})$, est l'ensemble de tous les atomes fondés construits à partir des symboles de prédicat de $\mathcal{L}$ et des termes de $\mathcal{U}(\mathcal{L})$. Ces atomes sont appelés* instanciations de Herbrand *de $\mathcal{L}$.*

Nous illustrons maintenant ces définitions avec l'exemple suivant :

Exemple 5. *Considérons l'ensemble T composé des clauses suivantes :*

$$\begin{array}{rcl} C & = & p(X,b) \leftarrow p(a,X) \\ et \quad D & = & p(X,Z) \leftarrow (p(X,Y) \wedge p(Y,Z)) \end{array}$$

L'univers de Herbrand de T est :

$$\{a,b\}$$

et sa base de Herbrand est :

$$\{p(a,a), p(a,b), p(b,a), p(b,b)\}$$

Définition 12 (Interprétation de Herbrand)**.** *Une* interprétation de Herbrand d'un langage de clauses $\mathcal{L}$ *est un sous-ensemble de $\mathcal{B}(\mathcal{L})$.*

Définition 13 (Valuation ou Assignation d'une valeur de vérité)**.** *Soit une interprétation de Herbrand $\mathcal{I}$ d'un langage de clauses $\mathcal{L}$, une clause C de $\mathcal{L}$, un ensemble G de toutes les clauses fondées obtenues en remplaçant les variables de C par des constantes de $\mathcal{L}$, un littéral fondé ℓ dont l'atome fondé $A \in \mathcal{B}(\mathcal{L})$. On note respectivement $val(C)$, $val(\ell)$ et $val(A)$ les* valuations *de la clause C, du littéral ℓ et de l'atome A définies de la manière suivante :*

$$val(A) = \begin{cases} vrai & si\ A \in \mathcal{I}, \\ faux & sinon. \end{cases}$$

$$val(\ell) = \begin{cases} vrai & si\ (\ell = A\ et\ A \in \mathcal{I})\ ou\ (\ell = \neg A\ et\ A \notin \mathcal{I}), \\ faux & sinon. \end{cases}$$

$$val(C) = \begin{cases} vrai & si\ \forall C' \in G\ avec\ C' = \{h_1, \cdots, h_m, \neg b_1, \cdots, \neg b_n\}, \\ & on\ a\ \forall h_i \in C'\ val(h_i) = vrai\ ou\ \forall b_j \in C'\ val(\neg b_j) = vrai, \\ faux & sinon. \end{cases}$$

Nous reprenons l'exemple 5 et illustrons la notion d'interprétation.

Exemple 6. *Les interprétations de Herbrand de l'ensemble T sont (par défaut, on ne met dans l'interprétation que les atomes évalués à vrai) :*

$\{\}$	$\{p(b,a), p(b,b)\}$
$\{p(a,a)\}$	$\{p(a,b), p(b,a)\}$
$\{p(a,b)\}$	$\{p(a,b), p(b,b)\}$
$\{p(b,a)\}$	$\{p(a,a), p(b,a), p(b,b)\}$
$\{p(b,b)\}$	$\{p(a,a), p(a,b), p(b,a)\}$
$\{p(a,a), p(a,b)\}$	$\{p(a,a), p(a,b), p(b,b)\}$
$\{p(a,a), p(b,a)\}$	$\{p(a,b), p(b,a), p(b,b)\}$
$\{p(a,a), p(b,b)\}$	$\{p(a,a), p(a,b), p(b,a), p(b,b)\}$

1. Si $\mathcal{L}$ ne contient pas de constante, on lui en ajoutera une arbitrairement afin de construire des termes fondés.

Définition 14 (Modèle de Herbrand)**.** *Soit une interprétation de Herbrand $\mathcal{I}$ d'un langage de clauses $\mathcal{L}$, une clause C de $\mathcal{L}$ et un ensemble de clauses T de $\mathcal{L}$. L'interprétation $\mathcal{I}$ est un* modèle de C *si et seulement si C est valuée à vrai par rapport à $\mathcal{I}$. L'interprétation $\mathcal{I}$ est un* modèle pour T *si et seulement si $\mathcal{I}$ est un modèle pour chaque clause de T.*

Exemple 7. *Soit les clauses C et D appartenant à l'ensemble de clauses T de l'exemple 5 ainsi que les interprétations suivantes :*

$$\begin{aligned} \mathcal{I}_1 &= \{p(b,b)\} \\ \mathcal{I}_2 &= \{p(a,a), p(a,b), p(b,a)\} \\ \mathcal{I}_3 &= \{p(a,a), p(a,b), p(b,a), p(b,b)\} \end{aligned}$$

$\mathcal{I}_1$ est un modèle pour la clause C mais pas pour D ni T. $\mathcal{I}_2$ est un modèle pour la clause D mais pas pour C ni T. Enfin, $\mathcal{I}_3$ est un modèle pour C, D et T.

Les clauses et les ensembles de clauses permettent la représentation d'informations, comme c'est le cas avec le concept de liste des exemples 2, 3 et 4. Il est parfois utile de vérifier si certaines suppositions, également représentées par des clauses, sont cohérentes avec nos connaissances existantes. La notion de *conséquence logique* le permet.

Définition 15 (Conséquence logique)**.** *Soit T et E deux ensembles de clauses d'un langage $\mathcal{L}$, et C une clause de $\mathcal{L}$. La clause C est une* conséquence logique d'un ensemble de clauses *T ou T implique logiquement C, notée $T \models C$, si et seulement si chaque modèle de T est un modèle de C. De la même manière, E est une conséquence logique de T, notée $T \models E$, si et seulement si $\forall C \in E,\ T \models C$.*

Il est impossible pour toutes les clauses de T d'être vraies sans que celles de E le soient. La notion de conséquence logique permet de regrouper les clauses par catégories vis-à-vis de leurs interprétations. Ainsi, on retrouve les notions de *validité* et de *satisfiabilité* le traduisant.

Définition 16 (Validité et Satisfiabilité)**.** *Soit une clause C d'un langage $\mathcal{L}$.*

1. *La clause C est* valide *ou bien tautologique, notée $\models C$, si chaque interprétation de $\mathcal{L}$ est un modèle de C. Auquel cas, C est* invalide*.*
2. *La clause C est* satisfiable *s'il existe une interprétation de $\mathcal{L}$ étant un modèle de C. Si aucune interprétation n'existe, elle est* insatisfiable *ce qui se note $C \models \square$ ou $C \models false$. La clause vide $\square$ représente la contradiction et est ainsi toujours insatisfiable.*
3. *La clause C est* contingente *si elle est satisfiable mais invalide.*

Les définitions établies ne sont, pour le moment, que d'ordre théorique et n'ont, à première vue, pas d'intérêt pratique. En effet, dans le cas des clauses, l'ensemble des termes pouvant constituer la base de Herbrand peut être infini. Dès lors, pour deux ensembles de clauses T_1 et T_2, il peut être impossible de vérifier en un temps fini si $T_1 \models T_2$ car une infinité d'interprétations peut exister. Ce constat est résumé par le théorème de Church [Church 1936] dont l'énoncé est le suivant :

Théorème 1. *Soit C une clause et T un ensemble fini de clauses, tester si $T \models C$ est indécidable.*

Malgré ce résultat d'indécidabilité, des méthodes semi-décidables permettent de tester si une clause est bien une conséquence logique d'un ensemble logique. L'une de ces méthodes est connue sous le nom de *résolution*. Afin de la présenter, il nous est nécessaire d'introduire une restriction sur la forme des clauses : les *clauses de Horn*.

2.3 Théorie des clauses de Horn

La théorie des clauses de Horn est une partie restreinte de la théorie des clauses. La restriction est ici faite sur la forme des clauses. Avant de donner la définition d'une clause de Horn, nous rappelons certaines notions utiles à celle-ci ainsi qu'à la notion de *résolution* introduite dans la section suivante.

Définition 17 (Clause définie, Programme défini et Fait)**.** *Une* clause définie *est une clause dont exactement un littéral est positif. Elle est ainsi de la forme :*

$$h \leftarrow b_1 \wedge \cdots \wedge b_n$$

L'atome h est appelé tête de la clause *et les atomes $b_1, \cdots, b_n$ forment le* corps de la clause. *Un* fait *est une clause définie sans corps de la forme :*

$$h \leftarrow$$

Un programme défini *est un ensemble fini de clauses définies.*

Un fait est une clause sans corps, l'atome de tête d'un fait ne dépend donc pas d'autres littéraux. Par conséquent, il est toujours vrai peu importe l'interprétation et correspond ainsi à une tautologie. La notion de clauses de Horn reprend celle de clauses définies.

Définition 18 (Clause de Horn, But et Sous-but)**.** *Une* clause de Horn *est une clause avec au plus un littéral positif. Un* but *est une clause de Horn n'ayant pas de littéral positif. Un* sous-but *est un littéral d'un but.*

Soulignons que la définition de connexion s'applique également aux clauses de Horn. Ainsi une clause de Horn est connectée si tous ses littéraux sont connectés.

Nous serons amenés à utiliser les termes *programme logique*, *programme* ou bien encore *théorie du domaine* pour nommer un ensemble de clauses définies constituant une base de connaissance. Dans le cadre de la programmation logique inductive, les clauses employées sont principalement des clauses définies. Ceci s'explique par leur forme qui permet la représentation de problèmes divers et variés comme par exemple celui des trains de Michalski [Larson 1977], celui des arches de Winston [Winston 1970] ou celui de la mutagénése de molécules cancérigènes [King 1995]. Le problème peut alors être nommé à l'aide du prédicat utilisé dans la tête des clauses manipulées. Le corps de ces clauses est une description du problème. Un ensemble de clauses décrivant un même prédicat est alors appelé *définition du prédicat.*

Définition 19 (Définition d'un prédicat)**.** *Une définition d'un prédicat p/n est un ensemble de clauses définies tel que chaque clause appartenant à cet ensemble a une tête dont le symbole de prédicat est p/n.*

Nous revenons momentanément sur les modèles de Herbrand afin de montrer l'intérêt des clauses Horn sur ce dernier. Il s'agit de la propriété d'intersection des modèles de Herbrand. Celle-ci nous informe que l'intersection de tous modèles de Herbrand d'un programme défini est un modèle de Herbrand de ce programme. Le modèle de Herbrand d'un programme défini est sa base de Herbrand. Cette dernière n'est pas toujours de taille finie. L'intersection de plusieurs modèles de Herbrand peut donc être de taille infinie. Nous nous intéressons plus particulièrement à l'un des modèles de Herbrand appelé *plus petit modèle de Herbrand.*

Définition 20 (Plus petit modèle de Herbrand)**.** *Le* plus petit modèle de Herbrand *d'un programme défini P, noté $\mathcal{M}_P$, est l'unique ensemble contenant tous les atomes fondés vrais pour n'importe quel modèle de P.*

Le plus petit modèle de Herbrand d'un programme logique P est l'intersection de tous les modèles de Herbrand de P. Ainsi, un programme P implique logiquement un atome fondé A si et seulement si $\mathcal{M}_P$ contient A. C'est une conséquence de la propriété d'intersection des modèles.

Dans le cas de l'exemple 5, le plus petit modèle de Herbrand est l'ensemble vide ($\{\}$) car cet exemple ne contient aucune tautologie. De ce fait, aucun atome fondé ne peut être valide pour n'importe quel modèle de P. En général, le plus petit modèle de Herbrand est de taille infinie. Dès lors, des techniques pour tester de manière efficace les conséquences logiques de programmes ont été développées. Ces techniques sont appelées *règles d'inférence*.

Avant de développer ces techniques de dérivation, nous présentons deux types de clauses issus des clauses définies et des clauses de Horn nommées *clauses sans fonction* et *clauses Datalog*.

Définition 21 (Clause sans fonction)**.** *Une* clause sans fonction *(function-free clause en anglais) est une clause ne contenant aucun symbole de fonction d'arité supérieure à 1.*

Un terme d'une clause sans fonction peut ainsi être une constante ou une variable, les termes complexes sont interdits. Ce type de clauses est un sous-ensemble syntaxique des clauses de Horn. L'association de cette propriété et des clauses de Horn permet de définir les clauses Datalog [Ceri 1989] comme suit :

Définition 22 (Clause Datalog)**.** *Une* clause Datalog *est une clause de Horn sans fonction.*

Les clauses Datalog bénéficient d'une évaluation polynomiale en temps d'un but dans son nombre de sous-buts [Becker 2011]. Elles sont, de ce fait, souvent utilisées dans le cadre d'apprentissage [Semeraro 1998, Esposito 1996]. La syntaxe et la sémantique des clauses maintenant bien définies, nous pouvons introduire d'une manière formelle la notion de *règle d'inférence*.

2.4 La résolution

Dans cette section, nous développons une technique de preuve parmi les plus connues dont l'algorithme a été introduit par Robinson : *la résolution* [Robinson 1965]. Cette méthode d'inférence ou technique de preuve permet d'effectuer un raisonnement à partir de clauses. Plus généralement, une technique de preuve est une procédure capable de générer une preuve ou d'en infirmer l'existence. Une *preuve* est un raisonnement affirmant qu'une clause C est bien une conséquence logique d'un ensemble de clauses T. Elle permet ainsi de déduire, mais également de vérifier, des informations induites par une clause vis-à-vis d'un ensemble de clauses. Supposons, par exemple, que nous disposons d'un ensemble de clauses T définissant un problème et d'une clause C correspondant à une supposition faite sur ce problème, il est possible de vérifier, à l'aide d'une technique de preuve, si la supposition C est vraie vis-à-vis de notre problème. Cet ensemble de clauses T est appelé *prémisse de la preuve* tandis que la clause C en est sa *conclusion*. Une preuve correspond alors à une séquence d'étapes. À chaque étape d'une preuve est dérivée une clause à partir de la clause précédemment dérivée et de la prémisse. La notion de dérivation fait ici référence à un mécanisme permettant de construire, à partir d'une clause et d'un ensemble de clauses, une nouvelle clause. L'une des règles de dérivation les plus connues et des plus utilisées, notamment en mathématique, s'appelle le *Modus ponens*. Son énoncé est le suivant :

Si A est vrai et $B \leftarrow A$ alors B est vrai.

noté :

$$A, B \leftarrow A \vdash B$$

Avant de définir cette technique de preuve, nous introduisons les notions de substitution et d'unificateur le plus général nécessaires à la résolution. Nous définissons ensuite la notion de résolution. Enfin, nous présentons une technique de preuve nommée la SLD-résolution. Il s'agit d'une adaptation de la résolution aux cas des clauses définies.

2.4.1 Substitution

La substitution permet le remplacement d'une variable par un terme dans une clause, un atome ou un terme. Son application sur ces derniers permet l'obtention d'une nouvelle clause, d'un nouvel atome ou d'un nouveau terme. Nous définissons la notion de substitution plus formellement de la manière suivante :

Définition 23 (Substitution et Substitution fondée)**.** *Une* substitution $\theta = \{X_1/t_1, \cdots, X_n/t_n\}$ *est une application d'un ensemble de variables* $\{X_1, \cdots, X_n\}$ *vers un ensemble de termes* $\{t_1, \cdots, t_n\}$ *telle que* t_i *est distinct de* X_i *pour* $1 \leqslant i \leqslant n$*. On note* $C\theta$*,* $A\theta$ *et* $t\theta$ *les applications respectives de* θ *à la clause* C*, à l'atome* A *et au terme* t*. Une application remplace simultanément chaque variable* X_i *par le terme* t_i *avec* $1 \leqslant i \leqslant n$*. Une substitution* $\theta = \{X_1/t_1, \cdots, X_n/t_n\}$ *est fondée si chaque terme* t_i *est fondé avec* $1 \leqslant i \leqslant n$*.*

La substitution donnée par un ensemble vide est appelée *substitution identité* ou *vide*.

Définition 24 (Composition de substitutions)**.** *Soit* $\theta_1 = \{X_1/s_1, \cdots, X_n/s_n\}$ *et* $\theta_2 = \{Y_1/t_1, \cdots, Y_m/t_m\}$ *deux substitutions, la* composition *de* θ_1 *et* θ_2*, notée* $\theta_1\theta_2$*, est une substitution obtenue à partir de l'ensemble :*

$$\{X_1/s_1\theta_2, \cdots, X_n/s_n\theta_2, Y_1/t_1, \cdots, Y_m/t_m\}$$

en supprimant les couples $X_i/s_i\theta_2$ *pour lesquels* $X_i = s_i\theta_2$ *et les couples* Y_i/t_i *tels que* $Y_i \in \{X_1, \cdots, X_n\}$*.*

On note θ^n, avec $n \in \mathbb{N}$, la composition de n substitutions θ, c'est à dire $\theta \cdots \theta$ (n fois).

Définition 25 (Variante Alphabétique)**.** *Deux clauses* C_1 *et* C_2 *sont des* variantes alphabétiques *ou* variantes*, notées* $C_1 \simeq C_2$*, s'il existe deux substitutions* θ_1 *et* θ_2 *telles que* $C_1\theta_1 = C_2$ *et* $C_1 = C_2\theta_2$*.*

Par la suite, nous considérons que des variantes alphabétiques sont une même clause. Une variante alphabétique peut-être obtenue à l'aide d'une *substitution de renommage*.

Définition 26 (Substitution de renommage)**.** *Soit une clause* C*, une* substitution de renommage *ou un* renommage *pour la clause* C *est une substitution étant une bijection d'un ensemble de variables* E *vers un ensemble de variables* F *telle que* $E \subseteq vars(C)$ *et* $(vars(C) \setminus E) \cap F = \emptyset$*.*

L'application d'une substitution à une clause permet l'obtention d'une nouvelle clause. Cette dernière a un lien de parenté avec la clause dont elle est issue. C'est ce que traduit la notion d'instance.

Définition 27 (Instance et Instance fondée)**.** *Un terme* t_1 *est une* instance d'un terme t_2 *s'il existe une substitution* θ *telle que* $t_1 = t_2\theta$*. Un atome* A_1 *est une* instance d'un atome A_2 *s'il existe une substitution* θ *telle que* $A_1 = A_2\theta$*. Un littéral* ℓ_1 *est une* instance d'un littéral ℓ_2 *s'il existe une substitution* θ *telle que* $\ell_1 = \ell_2\theta$*. Une clause* C_1 *est une* instance d'une clause C_2 *s'il existe une substitution* θ *telle que* $C_1 = C_2\theta$*. Une instance est* fondée *si le terme, l'atome, le littéral ou la clause correspondant à l'instance est fondé.*

Les notions de base nécessaires à l'unification posées, nous introduisons maintenant le procédé d'unification.

2.4.2 Unification

L'unification est un mécanisme de base nécessaire à la résolution. Elle repose principalement sur la notion de substitution introduite précédemment. Elle permet de démontrer que deux termes, en apparence différents, sont en fait identiques à un renommage de variable près. Nous définissons formellement l'unification et l'unificateur le plus général de la manière suivante :

Définition 28 (Unificateur et Unificateur le plus général)**.** *Une substitution θ est un* unificateur d'un ensemble $S = \{e_1, \cdots, e_n\}$ *si* $e_1\theta = \cdots = e_n\theta$ *tel que tous les éléments de S sont soit des termes, soit des atomes, soit des littéraux niés ou non niés. Un unificateur θ est un* unificateur le plus général *(ou most general unifier en anglais) d'un ensemble S de termes, d'atomes ou de littéraux, noté $mgu(S)$, si, pour chaque unificateur θ_1 de S, il existe une substitution θ_2 telle que $\theta_1 = \theta\theta_2$.*

Un ensemble fini de termes, d'atomes ou de littéraux sur lesquels a été appliqué un unificateur se réduit alors à un singleton. Nous utilisons la notation $A\theta = B\theta$ pour dire que la substitution θ est un unificateur de deux termes, de deux atomes ou de deux littéraux A et B.

2.4.3 La résolution

Nous ne donnons qu'une définition formelle de la résolution avant de l'adapter au cas des clauses définies.

Définition 29 (Facteur)**.** *Soit une clause C, une clause $S \subseteq C$ contenant au moins un littéral et un unificateur le plus général θ de S, on appelle $C\theta$ un* facteur *de C.*

Exemple 8. *Soit une clause $C = p(X) \leftarrow q(X) \wedge q(f(a))$. La clause $C\theta = p(f(a)) \leftarrow q(f(a))$, avec $\theta = \{X/f(a)\}$ un mgu de $\{q(X) \wedge q(f(a))\}$, est un facteur de C.*

Définition 30 (Résolution (règle de))**.** *Une clause R est un* résolvant de deux clauses C et D *si et seulement s'il existe trois substitutions $\theta_C, \theta_D, \theta$ et deux littéraux ℓ_C, ℓ_D tels que :*

1. *$C\theta_C$ est un facteur de C et $D\theta_D$ est un facteur de D ;*
2. *$vars(C\theta_C) \cap vars(D\theta_D) = \emptyset$;*
3. *ℓ_C est un littéral de $C\theta_C$ et ℓ_D un littéral de $D\theta_D$;*
4. *θ est un mgu de $\{\ell_C, \neg\ell_D\}$;*
5. *R est la clause $((C\theta_C \setminus \{\ell_C\}) \cup (D\theta_D \setminus \{L_D\}))\theta$.*

Les clauses C et D sont appelées clauses parentes *de R. R a donc été obtenu à partir de C et D à l'aide de la règle de résolution.*

Remarquons que les littéraux ℓ_C et ℓ_D ne sont pas compatibles puisque l'un des deux doit être nié et l'autre pas pour construire le *mgu* θ. Cette résolution permet à partir de deux clauses quelconques de générer une troisième clause. Nous pouvons maintenant définir la clôture par résolution d'un ensemble de clauses. Ceci correspond à l'ensemble des clauses construites à l'aide de la résolution à partir d'un ensemble de clauses.

Définition 31 (Clôture par résolution)**.** *Soit un ensemble de clauses T, la* clôture par résolution de T, *notée $Res^*(T)$, est définie de manière récursive :*

1. $Res^0(T) = T$,
2. $Res^{n>0}(T) = Res^{n-1}(T) \cup \{R \mid C, D \in Res^{n-1}(T)$ *et* R *est un résolvant de* C *et* $D\}$,
3. $Res^*(T) = Res^0(T) \cup Res^1(T) \cup Res^2(T) \cup \cdots$.

À l'aide de cette clôture par résolution est définie la notion de dérivation. Elle permet de tester si une clause appartient à la clôture par résolution d'un ensemble de clauses.

Définition 32 (Dérivation par résolution)**.** *Soit un ensemble de clauses* T*, une clause* C est dérivable de T par résolution*, notée* $T \vdash C$*, si et seulement si l'une des conditions suivantes est vérifiée :*

— C *est une tautologie*
— *ou il existe une clause* $D \in Res^*(T)$ *et une substitution* θ *telles que* $D\theta \subseteq C$.

Nous venons de définir la technique de preuve proposée par Robinson. Ce dernier a montré que cette technique est complète et correcte, d'où le théorème suivant :

Théorème 2. *Soit un ensemble de clauses* T *et une clause* C*,*
Correction : *si* $T \vdash C$*, alors* $T \models C$*,*
Complétude par réfutation : T *n'est pas satisfiable, c'est-à-dire* $T \models$ *false, si seulement si* $T \vdash$ *false*.

La résolution, comme technique de preuve, est inintéressante d'un point de vue computationnel. Pour cette raison, et grâce aux propriétés des clauses définies, une résolution plus efficace, appelée SLD-résolution, a été développée pour celles-ci.

2.4.4 Résolution-SLD

La SLD-résolution, dont nous allons détailler le fonctionnement, est une résolution propre aux clauses définies, voire aux clauses de Horn. Une de ses extensions, la SLDNF-résolution est notamment utilisée dans Prolog. Le terme SLD signifie que la résolution est linéaire, dirigée par une règle de sélection et s'applique à des programmes définis. On considère que les clauses sont ordonnées. La SLD-résolution d'un but G vis-à-vis d'un programme défini P tente d'unifier les clauses de P avec les sous-buts de G. Elle débute par l'unification du premier sous-but de G avec la tête d'une clause de P. Le choix de cette clause est réalisé de manière déterministe. On parcourt les clauses du programme défini dans l'ordre imposé avant le début de la résolution. L'atome de tête retenu doit être compatible avec le sous-but. Le choix de prendre le premier sous-but de G correspond à une règle de sélection. Il est montré dans [Nienhuys-Cheng 1997a] que le choix de la règle de sélection, c'est-à-dire l'ordre dans lequel sont choisis les buts, ne modifie pas la complétude ni la correction de la technique de preuve mais seulement son efficacité à construire la preuve. Il y est également prouvé que la SLD-résolution est l'une des règles les plus efficaces. Si une unification sans contradiction existe, alors un nouveau but est créé. Il est obtenu en remplaçant le sous-but dans le but actuel par les atomes du corps de la clause dont la tête a été unifiée avec ce sous-but et sur lesquels l'unificateur a été appliqué. La SLD-résolution est ensuite relancée de manière récursive sur le nouveau but. Si l'unification d'un sous-but échoue, alors l'unification de niveau supérieur, dont elle provient, échoue elle aussi. L'algorithme cherche alors une autre unification pour le premier sous-but du but. Cette étape s'appelle le *retour sur trace* (ou backtrack). Il peut se révéler très onéreux. À présent, définissons de manière plus formelle la SLD-résolution utilisant la notion d'unificateur le plus général. Il est induit dans les définitions suivantes que G, C et P ne partagent aucune variable.

Définition 33 (SLD-dérivé)**.** *Soit une clause $C = h \leftarrow b_1 \wedge \cdots \wedge b_m$ et un but $G = \leftarrow A_1 \wedge \cdots \wedge A_k \wedge \cdots A_n$, le but G' est un* SLD-dérivé *(ou juste dérivé) de G et C en utilisant l'unificateur le plus général θ si :*

1. *A_k est un atome appelé* atome sélectionné *;*
2. *θ est un upg de A_k et h ;*
3. *G' est de la forme $\leftarrow A_1\theta, \cdots, A_{i-1}\theta, b_1\theta, \cdots, b_m\theta, A_{i+1}\theta, \cdots, A_n\theta$.*

Le but G' un résolvant de G et C.

Définition 34 (SLD-dérivation)**.** *Soit un programme défini P et un but défini G, une* SLD-dérivation *de $P \cup \{G\}$ est un triplet contenant :*

— *une séquence (infinie ou finie) $G_0, G_1, \cdots$ de sous-buts avec $G_0 = G$,*
— *une séquence $C_1, C_2, \cdots$ de variantes des clauses de P*
— *et une séquence d'unificateur le plus général $\theta_1, \theta_2, \cdots$*

tel que chaque G_{i+1} est un SLD-dérivé de G_i et C_{i+1} avec pour mgu θ_{i+1}.

Définition 35 (SLD-réfutation)**.** *Soit P un programme défini et G un but défini, une* SLD-réfutation *de $P \cup \{G\}$ est une SLD-dérivation finie qui a pour dernier but, la clause vide (notée $\square$). Si G_n est la clause vide alors la SLD-réfutation est de taille n.*

Le fait d'obtenir une clause vide comme but indique que la résolution du but initial a réussi. On notera $P \vdash_{\mathsf{sld}} G$, le fait qu'il existe une SLD-réfutation de $P \cup \{G\}$. On dira alors que G est dérivable de P par SLD-résolution.

La *SLD*-résolution est un cas particulier de la résolution de Robinson, elle hérite ainsi des propriétés de complétude et de correction. Le théorème suivant, ré-établit par de nombreuses personnes comme Lee [Lee 1967], Kowalski [Kowalski 1970], ou bien encore Nienhuys-Cheng et de Wolf [hwei Nienhuys-cheng 1995], traduit ces propriétés et établit le lien entre l'implication logique et la SLD-résolution.

Théorème 3. *Soit un ensemble de clauses T et une clause C,*

Correction : *si $T \vdash_{\mathsf{sld}} C$, alors $T \models C$,*

Complétude par réfutation : *T n'est pas satisfiable, c'est-à-dire $T \models false$, si seulement si $T \vdash_{\mathsf{sld}} false$.*

Pour illustrer le fonctionnement d'une SLD-résolution, nous donnons le déroulement de la résolution d'un programme défini simple et de son but sous la forme d'un arbre SLD. Un arbre SLD est une représentation de toutes les dérivations obtenues avec la règle de sélection et se définit de la manière suivante :

Définition 36 (Arbre SLD)**.** *Soit un programme défini P et un but G, l'*arbre SLD *de $P \cup \{G\}$ est un arbre respectant les conditions suivantes :*

— *Chaque nœud est un but pouvant être vide,*
— *La racine de l'arbre est G,*
— *Un nœud N, différent de la racine, est le résolvant de son père et d'une clause de P.*
— *Les nœuds vides n'ont pas d'enfant.*

Exemple 9. *Soit le programme défini P contenant les clauses :*

$grand\_parent(C, G) \leftarrow parent(C, G), parent(P, G).$
$parent(C, P) \leftarrow pere(C, P).$
$parent(C, P) \leftarrow mere(C, P).$
$pere(charles, luc) \leftarrow .$

$pere(anna, george) \leftarrow .$
$mere(charles, anna) \leftarrow .$

et le but :

$\leftarrow grand\_parent(charles, X).$

On obtient l'arbre SLD de $P \cup \{G\}$ de la figure 2.1. Les numéros en bleu indiquent l'ordre dans lequel les noeuds sont ajoutés à l'arbre SLD. Ils indiquent ainsi l'ordre dans lequel les différentes dérivations ont été effectuées. Cet ordre est imposé par la règle de sélection de la SLD-résolution. On choisit le sous-but le plus à gauche dans le but puis on effectue une exploration en profondeur et ainsi de suite.

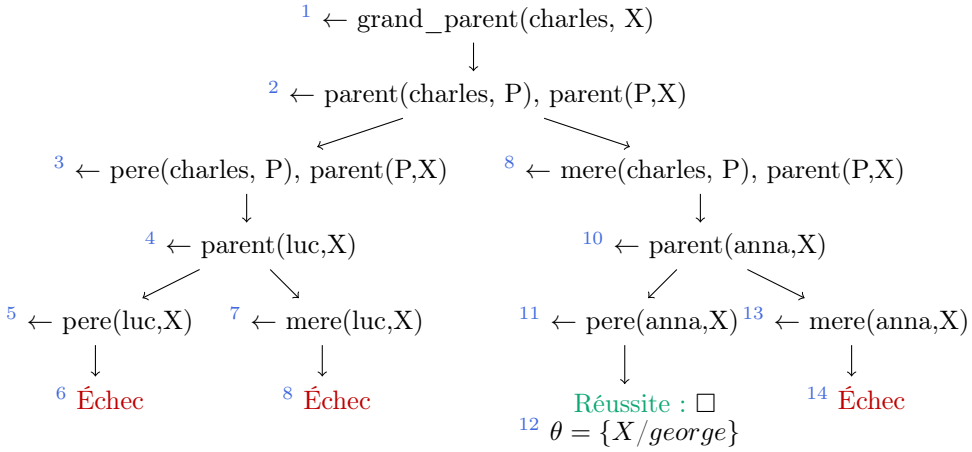


FIGURE 2.1 – Arbre SLD de $P \cup \{G\}$

L'ensemble de tous les atomes fondés dérivables à partir d'un programme défini P grâce à la SLD-résolution correspond au plus petit modèle de Herbrand de P. Rappelons qu'un plus petit modèle de Herbrand peut être infini. Ceci a pour conséquence l'existence de SLD-dérivations infinies. La SLD-résolution est donc, comme la résolution, semi-décidable dans le cas des clauses définies mais également des clauses.

Afin de bénéficier de modèle de Herbrand fini, les notions de *h-easy modèle* et *clause syntaxiquement générative* ont été définies. La notion de *h-easy modèle* est proposée pour la première fois par [Blum 1975] et a été adaptée aux problèmes d'apprentissage dans GOLEM [Muggleton 1990] et MIS [Shapiro 1981].

Définition 37 (h-easy)**.** *Soit un ensemble de clauses T, un atome A est un h-easy atome de T si $T \models A$ tel que A peut être obtenu à partir de T après h étapes de dérivation. Le h-easy modèle de Herbrand de T, noté $M_h(T)$, est l'ensemble des instanciations de Herbrand de h-easy atomes de T.*

Les clauses syntaxiquement génératives ont la propriété d'avoir un h-easy modèle fini pour une valeur de h donnée. Elles sont définies comme suit :

Définition 38 (Clause syntaxiquement générative)**.** *Une clause $h \leftarrow \ell_1, \cdots, \ell_n$ est* une clause syntaxiquement générative *si et seulement si $vars(h) \subseteq vars(\{\ell_1, \cdots, \ell_n\})$. Un ensemble de clauses est syntaxiquement génératif si et seulement si toutes ses clauses sont syntaxiquement génératives.*

Un h-easy modèle de clauses syntaxiquement génératives est alors l'ensemble des atomes fondés dérivables de ces clauses par résolution après, au plus, h étapes de dérivation.

Dans cette section, nous avons défini clairement ce que sont la résolution pour les clauses et la SLD-résolution pour les clauses de Horn. Nous avons remarqué qu'il existe, que ce soit pour la résolution ou la SLD-résolution, des cas entrainant des résolutions infinies. La résolution et la SLD-résolution sont des procédures complètes et correctes mais semi-décidables, il convient alors de s'intéresser à d'autres relations moins puissantes mais utilisables. Cela fera l'objet du chapitre 4 de cette thèse. Terminons ce chapitre par une opération appelée *aplatissement d'une clause* transformant des clauses définies en clauses Datalog.

2.5 Aplatissement d'une clause

Nous avons vu précédemment la notion de clause sans fonction et celle de clause Datalog. Dans certains cas, il est intéressant de transformer une théorie clausale en une théorie équivalente de clauses sans fonction. De nombreux systèmes en ILP ont utilisé ce procédé [Porter 1990, Rouveirol 1992, Krogel 2003]. Cela permet d'éviter la gestion de terme complexe. La notion d'aplatissement introduite en ILP dans [Rouveirol 1992, Rouveirol 1994] le permet. L'aplatissement d'une clause consiste à remplacer chaque terme $f(t_1, \cdots, t_n)$ d'une clause par une nouvelle variable X et à ajouter un littéral $p_f(t_1, \cdots, t_n, X)$ au corps de cette clause. Ce processus est répété jusqu'à ce que tous les termes complexes soient éliminés de la clause. L'algorithme 1 réalise l'aplatissement d'une clause définie C et retourne un couple contenant à la fois la clause aplatie $\mathsf{flat}(C)$, sans terme complexe ou constante, et un ensemble d'atomes $S = \mathsf{flatDefs}(C)$. L'ensemble S est utile pour la reconstruction de la clause avec fonction à partir de celle sans fonction. Ce dernier fait le lien entre les termes aplatis et les atomes ajoutés au corps de la clause.

Algorithme 1 Aplatissement d'une clause définie.

function flatten(C)
1: **let** $D := C$ **and** $D_{cst} := \emptyset$;
2: **let** $S_{cst} := \emptyset$ **and** $S_{pred} := \emptyset$;
3: **while** D contains a constant c **do**
4: **replace** all occurrences of c in D by a new variable X ;
5: **add** $P_c(X)$ **to** D_{cst} ;
6: **add** $P_c(c)$ **to** S_{cst} ;
7: **end while**
8: **while** D contains a term $t = f(X_1, \cdots, X_n)$ **do**
9: **replace** all occurrences of t in D by a new variable X ;
10: **add** $P_f(X_1, \cdots, X_n, X)$ **to** the body of D ;
11: **if** S_{pred} does not contain atom of the form $P_f(X_1, \cdots, X_n, f(X_1, \cdots, X_n))$ **then**
12: **add** $P_f(X_1, \cdots, X_n, f(X_1, \cdots, X_n))$ **to** S_{pred} ;
13: **end if**
14: **end while**
15: **let** $\mathsf{flat}(C) := D \cup D_{cst}$ **and** $\mathsf{flatDefs}(C) := S_{cst} \cup S_{pred}$;
16: **return** $(\mathsf{flat}(C), \mathsf{flatDefs}(C))$;
end function

L'algorithme présenté ne permet que la transformation d'une clause définie. Dans le cas d'un ensemble de clauses définies $T = \{C_1, \cdots, C_n\}$, le programme défini aplati correspon-

dant est construit de la manière suivante :

$$\mathsf{flat}(T) = \mathsf{flat}(C_1) \cup \cdots \cup \mathsf{flat}(C_n)$$

$$\mathsf{flatDefs}(T) = \mathsf{flatDefs}(C_1) \cup \cdots \cup \mathsf{flatDefs}(C_n)$$

Illustrons maintenant le fonctionnement de cet algorithme :

Exemple 10. *Considérons les clauses définies suivantes appartenant à une théorie T :*

$$\begin{array}{lcl} C & = & append(nil, X, X). \\ D & = & append(cons(H,T), Y, cons(H,W)) \leftarrow append(T, Y, W). \end{array}$$

Les clauses aplaties sont :

$$\begin{array}{lcll} \mathsf{flat}(C) & = & append(X,Y,Y) & \leftarrow \; nil(X). \\ \mathsf{flat}(D) & = & append(C1,Y,C2) & \leftarrow \; append(T,Y,W) \\ & & & \quad \wedge \; cons(H,T,C1) \wedge cons(H,W,C2). \\ \mathsf{flatDefs}(C) & = & \{nil(nil)\} & \\ \mathsf{flatDefs}(D) & = & \{cons(X1, X2, cons(X1, X2))\} & \end{array}$$

L'aplatissement d'une clause résulte en une théorie contenant plusieurs clauses. En effet, l'ensemble des atomes $\mathsf{flatDefs}(C)$ doit être pris en compte sous la forme de faits lorsque la clause $\mathsf{flat}(C)$ est considérée. Dans le cas des clauses définies, [Rouveirol 1994, Nienhuys-Cheng 1997a, Hirata 1999] ont montré le théorème suivant :

Théorème 4. *Soit un programme défini T et une clause définie C,*

1. *$T \models C$ si et seulement si $\mathsf{flatDefs}(T) \cup \mathsf{flat}(T) \models \mathsf{flat}(C)$*
2. *si $\mathsf{flat}(T) \models \mathsf{flat}(C)$, alors $T \models C$.*

La réciproque de ce théorème n'est pas vraie pour n'importe quelle clause définie. Nous introduisons plusieurs restrictions sur la forme des clauses utilisées afin de l'établir.

Définition 39 (Clause récursive et individuellement récursive)**.** *Une clause est* récursive *si elle contient un littéral positif et un littéral négatif ayant le même symbole de prédicat. Sinon, la clause est dite non-récursive. Une clause est* individuellement récursive *(*singly recursive *en anglais) si un littéral de son corps est unifiable avec sa tête.*

Exemple 11. *Soit les clauses suivantes :*

$$\begin{array}{lcl} C & = & p(f(x)) \leftarrow p(x). \\ D & = & (p(a), q(x))) \leftarrow p(f(y)). \\ E & = & p(a) \leftarrow q(x,a). \\ F & = & (p(x,y) \vee p(y,x)) \leftarrow . \end{array}$$

Les clauses C et D sont récursives, tandis que les clauses E et F ne le sont pas.

Certaines clauses peuvent être également résolues par une variante alphabétique d'elle-même. Le terme anglais utilisé pour nommer ces clauses est : *self-resolving clauses*. Cette notion est fortement liée à celle de récursivité. Or, cette dernière peut entrainer des résolutions infinies rendant impossible l'utilisation des techniques de preuves précédemment développées. Voilà un exemple de clause self-resolving :

Exemple 12. *Soit les clauses suivantes :*

$$\begin{array}{lcl} C & = & marier(julie, jean). \\ D & = & marier(X,Y) \leftarrow marier(Y,X). \textit{ (clause self-resolving)} \end{array}$$

Avec ce programme, le but $(\leftarrow marier(jean, julie))$ *est satisfait. Néanmoins, le but* $(\leftarrow marier(julie, robert))$ *entraîne un nombre infini d'évaluations dû à la présence de la clause self-resolving. Cette clause self-resolving est une clause récursive, l'inverse est souvent vrai. Cependant, il ne faut pas en déduire qu'une clause récursive est forcément self-resolving comme démontrer ici :* $p(X, a) \leftarrow p(X, b)$.

Les clauses self-resolving sont l'une des raisons de ce problème et font l'objet d'études comme dans [Ohlbach 1998]. Ces travaux sont motivés par la découverte de solutions permettant de détecter, voire contourner, les évaluations infinies.

Les restrictions nécessaires à la réciproque du précédemment théorème posées, nous pouvons l'énoncer.

Théorème 5. *Soit deux clauses définies* C *et* D*, si :*

1. C *n'est pas self-resolving et* D *non tautologique,*
2. *ou* D *n'est pas récursive*
3. *ou* C *n'est pas singly récursive*

alors la proposition suivante est vraie :

$$C \models D \text{ si et seulement si } \mathsf{flat}(C) \models \mathsf{flat}(D)$$

La classe des clauses définies pour laquelle l'aplatissement préserve l'implication correspond à celle pour laquelle l'implication est décidable. Celle des clauses dont l'aplatissement ne préserve pas l'implication correspond à celle pour laquelle ce problème est indécidable [Hirata 1999].

Finalement, remarquons qu'il est possible d'obtenir la clause originale C à partir d'une clause aplatie $\mathsf{flat}(C)$ et d'un ensemble $\mathsf{flatDefs}(C)$. Cette opération consiste en la SLD-résolution de $\mathsf{flat}(C)$ avec pour théorie les atomes présents dans $\mathsf{flatDefs}(C)$. Plus de détails sur cette opération peuvent être trouvés dans [Nienhuys-Cheng 1997a].

2.6 Conclusion

Nous avons ainsi vu les notions de base nécessaires à la compréhension de la programmation logique ainsi que de la PLI. Différents types de clauses ont été définis et une méthode de transformation d'une clause définie en une clause définie sans symbole de fonction présentée. L'accent fut ensuite porté sur la notion d'implication logique et sur la difficulté d'un apprentissage utilisant cette dernière, de par sa complexité et son indécidabilité. Dans la continuité de cela, une présentation de la résolution ainsi qu'une adaptation de cette dernière aux clauses définies, correspondant à une alternative semi-décidable à la résolution, a été faite.

Les bases de la PLI posées, nous présenterons dans le chapitre suivant le problème de classification ainsi que la nature des données traitées. Nous introduirons ensuite le cadre d'apprentissage utilisé ainsi que le mode d'évaluation de nos expériences.

Chapitre 3

La PLI pour la classification de documents

Sommaire

De jour en jour, l'échange d'informations entre des périphériques variés, comme les ordinateurs ou les smartphones, ne cesse de soulever de nombreux problèmes par le volume des données transmises et acheminées entre ces derniers ainsi que par l'hétéroclisme des applications utilisées. Afin d'être capable de traiter ces flux de données et de les représenter, un certain nombre de standards ont été développés. Parmi ces standards figurent le XML [Bray 2008]. Il a su s'imposer par rapport aux autres formats grâce sa représentation des données semi-structurées utilisées et utilisables par la plupart des outils de communication. En général, ces logiciels combinent des données semi-structurées à des langages permettant d'effectuer des traitements intelligents. Du fait du nombre croissant de documents, certains traitements comme la classification supervisée de documents XML sont devenus des prérequis pour utiliser le potentiel de ces données. Cette tâche a pour but d'apprendre à regrouper les documents traitant d'une même thématique.

Nous commençons ce chapitre par définir quelques structures de données issues de la théorie des graphes. Nous présentons ensuite le standard XML et faisons le lien avec les structures de données précédemment introduites. Puis, nous définissons la tâche de classification de documents XML et la situons dans le cadre classique de la PLI [Muggleton 1991]. Afin de cadrer notre tâche d'apprentissage, nous précisons un modèle d'apprentissage : l'identification à la limite [Gold 1967]. Enfin, nous donnerons les critères d'évaluation d'un apprentissage.

3.1 Structures de données

Les structures de données que nous allons manipuler sont les mots, les graphes et les arbres. Ces structures de données sont fréquentes en Informatique et permettent une représentation des documents XML. Nous rappelons ici leurs définitions ainsi que quelques problèmes de base leurs étant associés.

3.1.1 Mot

Un *alphabet* Σ est un ensemble totalement ordonné de symboles. La taille d'un alphabet Σ, notée $|\Sigma|$, est le nombre de symboles contenus dans cet alphabet. Un *mot* $w = w_0 w_1 \cdots w_k$ est une séquence finie de symboles telle que $\forall i \in [0..k],\ w_i \in \Sigma$. Le symbole w_i se situe à la $i+1^{\text{e}}$ position dans le mot w. On note $|w|$ la taille du mot w, c'est-à-dire $|w| = (k+1)$, et $|w|_{l \in \Sigma}$ le nombre de fois que le symbole l apparait dans le mot w. Le symbole ε représente le *mot vide*, c'est-à-dire le mot sans symbole. L'ensemble des mots construits à partir de l'alphabet Σ est noté $\Sigma^\star$. Enfin, nous utilisons la notation $w_1 w_2$ pour représenter la concaténation de deux mots $w_1, w_2 \in \Sigma^\star$.

3.1.2 Graphe

Définition 40 (Graphe et Digraphe étiquetés)**.** *Un* graphe étiqueté G *est défini par un tuple* $G = (N_G, E_G, lab_\Sigma^G, lab_\Gamma^G)$ *où :*

- N_G *est l'ensemble de nœuds,*
- $E_G \subseteq \{\{n_i, n_j\} \mid n_i, n_j \in N_G\}$ *l'ensemble des arêtes,*
- $lab_\Sigma^G : N_G \to \Sigma$ *est une fonction d'étiquetage des nœuds de* N_G *par des symboles de l'alphabet* Σ
- *et* $lab_\Gamma^G : E_G \to \Gamma$ *est une fonction d'étiquetage des arêtes de* E_G *par des symboles de l'alphabet* Γ.

Un digraphe étiqueté G *est un graphe* $G = (N_G, E_G, lab_\Sigma^G, lab_\Gamma^G)$ *tel que* $E_G \subseteq N_G \times N_G$, *c'est-à-dire* $E_G \subseteq \{(n_i, n_j) \mid n_i, n_j \in N_G\}$, *le reste est inchangé. Une arête est alors orientée et s'appelle un* arc.

Une étiquette attache une information à un nœud ou à une arête. Cette information peut-être omise pour une partie ou la totalité des nœuds et arêtes. Un tel graphe ou digraphe est ainsi partiellement étiqueté ou non étiqueté. Les fonctions lab_Σ et lab_Γ sont, en fonction de cela, omises du tuple le définissant.

Définition 41 (Sous-graphe)**.** *Soit deux graphes ou digraphes* $G_1 = (N_{G_1}, E_{G_1}, lab_\Sigma^{G_1}, lab_\Gamma^{G_1})$ *et* $G_2 = (N_{G_2}, E_{G_2}, lab_\Sigma^{G_2}, lab_\Gamma^{G_2})$, *on dit que* G_1 *est un* sous-graphe de G_2 *si :*

1. $N_{G_2} \subseteq N_{G_1}$,
2. $E_{G_2} \subseteq E_{G_1}$,
3. $\forall n \in N_{G_2},\ lab_\Sigma^{G_1}(n) = lab_\Sigma^{G_2}(n)$,
4. $\forall e \in E_{G_2},\ lab_\Gamma^{G_1}(e) = lab_\Gamma^{G_2}(e)$.

Nous continuons nos définitions en établissant des propriétés sur un graphe ou un digraphe $G = (N_G, E_G, lab_\Sigma^G, lab_\Gamma^G)$. La *taille* de G, notée $|G|$, est le nombre d'arêtes de G, c'est-à-dire $|E_G|$. Elle est bornée par $|N_G|^2$ dans le cas d'un digraphe et par $|N_G| \times (|N_G| - 1)/2$ pour un graphe où $|N_G|$ est le nombre de nœuds de G. Un *chemin* dans G est une séquence $(n_0, n_1, \cdots, n_i, \cdots, n_m)$ de nœuds telle que pour tout $0 \leqslant i < m$, on a $(n_i, n_{i+1}) \in E_G$ si G est un digraphe ou $\{n_i, n_{i+1}\} \in E_G$ si G est un graphe. Un *cycle* C dans G est une séquence de nœuds distincts $C = (n_0, \cdots, n_k)$, à l'exception de n_0 et n_k, telle

que pour tout $i \in [0..k-1]$, on a $(n_i, n_{i+1}) \in E_G$ si G est un digraphe ou $\{n_i, n_{i+1}\} \in E_G$ si G est un graphe. Un *self-loop* est une arête connectant un nœud à lui-meme et est un cas particulier des cycles. Un graphe ou un digraphe sans cycle est dit *acycle*. Deux nœuds sont *connectés* dans un graphe ou un digraphe s'il existe un chemin entre ces nœuds. Le graphe ou digraphe G est *connecté* ou *connexe* si et seulement s'il existe pour tous nœuds $n_i, n_j \in N_G$ un chemin entre n_i et n_j dans G. Un graphe non connexe peut être décomposé en une union de graphes connexes. L'union de deux graphes ou deux digraphes, $G_1 = (N_{G_1}, E_{G_1}, lab_\Sigma^{G_1}, lab_\Gamma^{G_1})$ et $G_2 = (N_{G_2}, E_{G_2}, lab_\Sigma^{G_2}, lab_\Gamma^{G_2})$ avec $N_{G_1} \cap N_{G_2} = \emptyset$, est respectivement un graphe ou un digraphe $G_3 = (N_{G_3}, E_{G_3}, lab_\Sigma^{G_3}, lab_\Gamma^{G_3})$ où :

- $N_{G_3} = N_{G_1} \cup N_{G_2}$
- $E_{G_3} = E_{G_1} \cup E_{G_2}$
- $\forall n \in N_{G_3},\ lab_\Sigma^{G_3}(n) = \begin{cases} lab_\Sigma^{G_1}(n) & \text{si } n \in N_{G_1} \\ lab_\Sigma^{G_2}(n) & \text{sinon.} \end{cases}$
- $\forall e \in E_{G_3},\ lab_\Gamma^{G_3}(e) = \begin{cases} lab_\Gamma^{G_1}(e) & \text{si } e \in E_{G_1} \\ lab_\Gamma^{G_2}(e) & \text{sinon.} \end{cases}$

Les deux derniers points sont à omettre lorsque les fonctions d'étiquetage ne sont pas définies. Un graphe ou un digraphe G non connexe est alors décomposable sous la forme d'une union respectivement de graphes ou de digraphes connexes $G = G_1 \cup \cdots \cup G_m$ telle que pour tout $i, j \in [1..m]$, G_i et G_j ont des nœuds et des arêtes disjointes. Chaque composante G_i est *maximalement connectée* puisqu'elle est connectée et est indépendante des autres composantes.

Illustrons ces définitions avec l'exemple suivant :

Exemple 13. *Soit un graphe $G = (N_G, E_G, lab_\Sigma^G, lab_\Gamma^G)$ dont les nœuds et les arêtes sont étiquetés et deux digraphes $D_1 = (N_{D_1}, E_{D_1}, lab_\Sigma^{D_1})$ et $D_2 = (N_{D_2}, E_{D_2}, lab_\Sigma^{D_2})$ où :*

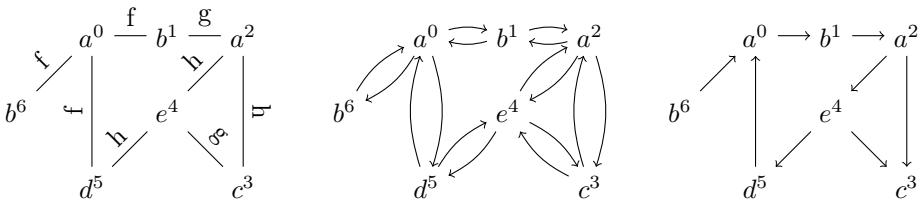
$$\begin{array}{lcl} N_G & = & \{0,\ 1,\ 2,\ 3,\ 4,\ 5,\ 6\} \\ & = & N_{D_1} \\ & = & N_{D_2} \\ E_G & = & \{\{0,6\},\ \{0,1\},\ \{1,2\},\ \{2,3\},\ \{2,4\},\ \{3,4\},\ \{4,5\},\ \{0,5\}\} \\ E_{D_1} & = & \{(0,6),\ (6,0),\ (0,1),\ (1,0),\ (1,2),\ (2,1),\ (2,3),\ (3,2), \\ & & (2,4),\ (4,2),\ (3,4),\ (4,3),\ (4,5),\ (5,4),\ (0,5),\ (5,0)\} \\ E_{D_2} & = & \{(6,0),\ (0,1),\ (1,2),\ (2,3),\ (2,4),\ (3,4),\ (4,5),\ (5,0)\} \\ \Sigma & = & \{a,b,c,e,d\} \\ \Gamma & = & \{f,g,h\} \end{array}$$

avec :

$$lab_\Sigma^X(0) = a, \quad lab_\Sigma^X(1) = b, \quad lab_\Sigma^X(2) = a, \quad lab_\Sigma^X(3) = c, \quad lab_\Sigma^X(4) = e,$$
$$lab_\Sigma^X(5) = d, \quad lab_\Sigma^X(6) = b$$

où X peut-être G, D_1 ou D_2 et :

$$lab_\Gamma^G(\{0,1\}) = f, \quad lab_\Gamma^G(\{1,2\}) = g, \quad lab_\Gamma^G(\{2,4\}) = h, \quad lab_\Gamma^G(\{2,3\}) = h,$$
$$lab_\Gamma^G(\{3,4\}) = g, \quad lab_\Gamma^G(\{4,5\}) = h \quad lab_\Gamma^G(\{5,0\}) = f, \quad lab_\Gamma^G(\{0,6\}) = f$$



Le graphe G contient le cycle $C = (2,\ 3,\ 4,\ 2)$. Ce cycle est également présent dans D_1 mais pas dans D_2. On peut aussi observer la présence du chemin $P = (0, 1, 2, 4)$ dans chaque graphe. Remarquons enfin que le digraphe D_1 est obtenu à partir du graphe G en définissant $E_{D_1} = \{(n_i, n_j), (n_j, n_i) \mid \{n_i, n_j\} \in E_G\}$. Cela permet la transformation d'un graphe en un digraphe.

La notion de graphe posée, intéressons nous maintenant à un cas particulier : les arbres.

3.1.3 Arbre

Un arbre est un graphe connexe et acyclique. Nous nous intéressons plus particulièrement au cas des arbres issus de graphes dirigés et donnons une définition formelle de ce concept.

Définition 42 (Arbre enraciné, étiqueté, ordonné et d'arité non bornée)**.** *Un* arbre t enraciné, ordonné et d'arité non bornée dont les nœuds et les arêtes *sont respectivement étiquetés sur un alphabet Σ et un alphabet Γ est défini par un tuple $(N_t, root_t, child_t, ns_t, lab^t_\Sigma, lab^t_\Gamma)$ où :*

- *N_t est l'ensemble des nœuds de t appelé* domaine de l'arbre. *Cet ensemble est un sous-ensemble non vide et préfixe clos de $\mathbb{N}$ [1] tel que pour deux éléments $ni,\ nj \in \mathbb{N}^\star$ avec $i, j \in \mathbb{N}$ et $i < j$, on a $nj \in N_t$ si et seulement si $ni \in N_t$.*
- *$root_t \in N_t$ est la* racine *de t et est représentée par le mot vide ε.*
- *$child_t \subset N_t \times N_t$ est l'ensemble des couples de nœuds tel que :*

$$child_t = \{(m, mi) \mid m, mi \in N_t \text{ et } i \in \mathbb{N}\}$$

Il décrit une relation de parenté entre les nœuds. Un nœud n est le père d'un nœud m ou a pour fils m si $(n, m) \in child_t$

- *$ns_t \subset N_t \times N_t$ est un ensemble de couples de nœuds tel que :*

$$ns_t = \{(mi, m(i+1)) \mid mi, m(i+1) \in N_t \text{ et } i \in \mathbb{N}\}$$

Il désigne une relation de fraternité entre les fils d'un même nœud, elle est nommée next-sibling. *Un nœud mi est le frère de gauche du nœud $m(i+1)$ ou a pour frère de droite $m(i+1)$ si $(mi, m(i+1)) \in ns_t$.*

- *$lab^t_\Sigma : N_t \to \Sigma$ est une fonction d'étiquetage des nœuds par des symboles de l'alphabet Σ.*
- *$lab^t_\Gamma : E_t \to \Gamma$ est une fonction d'étiquetage des arcs par des symboles de l'alphabet Γ.*

Il est d'usage de distinguer les nœuds d'un arbre en fonction de leur emplacement dans celui-ci. Un nœud sans fils est appelé *feuille*. Un nœud ayant au moins un fils est nommé *nœud interne*. Enfin, un nœud sans père est qualifié de *racine* de l'arbre, celui-ci est unique.

1. Un ensemble $S \subseteq \mathbb{N}^\star$ est préfixe clos si pour tout $n = mk \in S$ avec $m, k \in \mathbb{N}^\star$, on a $m \in S$.

Nous différencions également certains arcs présents dans $child_t$ afin de former l'ensemble fc_t correspondant à une relation *first-child* (premier fils). Cet ensemble est défini ainsi :

$$fc_t = \{(n, n0) \in child_t\}$$

et correspond aux arcs liant un nœud à son premier fils.

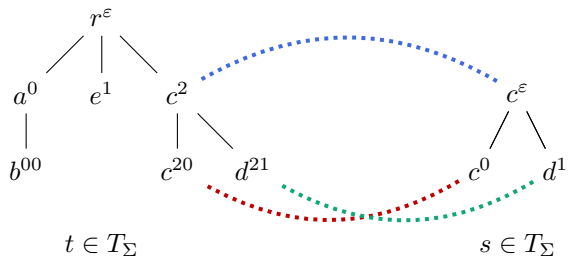
Un arbre étant un graphe, il est possible d'adapter la notion de *sous-graphe* à celle d'arbre afin d'obtenir celle de *sous-arbre*.

Définition 43 (Sous-arbre)**.** *Un* sous-arbre s d'un arbre $t = (N_t, root_t, child_t, ns_t, lab^t_\Sigma, lab^t_\Gamma)$ *est un arbre* $s = (N_s, root_s, child_s, ns_s, lab^s_\Sigma, lab^s_\Gamma)$ *tel qu'il existe un nœud* $r \in N_t$ *pour lequel les conditions suivantes sont vérifiées :*

1. $\forall n \in N_s,\ rn \in N_t$,
2. $\forall (n, m) \in child_s,\ (rn, rm) \in child_t$,
3. $\forall (n, m) \in ns_s,\ (rn, rm) \in ns_t$,
4. $\forall n \in N_s,\ lab^s_\Sigma(n) = lab^t_\Sigma(rn)$,
5. *et* $\forall (n, m) \in E_s,\ lab^s_\Gamma((n, m)) = lab^t_\Gamma((rn, rm))$.

Nous notons $subtree_t(n)$ *avec* $n \in N_t$ *le sous-arbre enraciné au nœud* n *dans l'arbre* t.

Exemple 14. *Soit deux arbres* $t, s \in T_\Sigma$ *définis par les tuples* $t = (N_t, \varepsilon, child_t, ns_t, lab^t_\Sigma)$ *et* $s = (N_s, \varepsilon, child_s, ns_t, lab^s_\Sigma)$ *tels que :*



$\Sigma = \{r, a, b, c, d, e\}$

$$\begin{array}{lcl} N_t & = & \{\varepsilon, 0, 1, 2, 00, 20, 21\} \\ child_t & = & \{(\varepsilon, 0),\ (\varepsilon, 1),\ (\varepsilon, 2,),\ (0, 00),\ (2, 20), (2, 21)\} \\ ns_t & = & \{(0, 1),\ (1, 2),\ (20, 21)\} \end{array}$$

$lab^t_\Sigma(\varepsilon) = r, \quad lab^t_\Sigma(0) = a, \quad lab^t_\Sigma(1) = e, \quad lab^t_\Sigma(2) = c, \quad lab^t_\Sigma(00) = b,$
$lab^t_\Sigma(20) = c, \quad lab^t_\Sigma(21) = d$

$$\begin{array}{lcl} N_s & = & \{\varepsilon, 0, 1\} \\ child_s & = & \{(\varepsilon, 0),\ (\varepsilon, 1)\} \\ ns_s & = & \{(0, 1)\} \end{array}$$

$lab^s_\Sigma(\varepsilon) = c, \quad lab^s_\Sigma(0) = c, \quad lab^s_\Sigma(1) = d$

Dans cet exemple, l'arbre s *est un sous-arbre de l'arbre* t.

La taille d'un arbre t, notée $|t|$, correspond au nombre de nœuds de l'arbre, c'est-à-dire $|N_t|$. Il est possible de borner la taille d'un arbre en limitant sa *profondeur* et sa *largeur*.

Définition 44 (Profondeur et Largeur)**.** *Soit un arbre* $t = (N_t, root_t, child_t, ns_t, lab^t_\Sigma, lab^t_\Gamma)$ *et un nœud* $n \in N_t$*. On note respectivement* $depth(n)$ *et* $depth(t)$ *la* profondeur *de* n *et celle de* t *définies comme suit :*

$$\begin{array}{lcl} depth(n) & = & \begin{cases} 0 & si\ n = \varepsilon, \\ 1 + depth(m) & tel\ que\ (m,n) \in child_t\ sinon. \end{cases} \\ depth(t) & = & \max(\{depth(n) \mid n \in N_t\}) \end{array}$$

et $width(n)$ *et* $width(t)$ *la* largeur *de* n *et celle de* t *:*

$$\begin{array}{lcl} width(n) & = & \begin{cases} 0 & si\ n\ est\ une\ feuille, \\ |\{m \mid (n,m) \in child_t\}| & sinon. \end{cases} \\ width(t) & = & \max(\{depth(n) \mid n \in N_t\}) \end{array}$$

Exemple 15. *L'arbre* t *de l'exemple 14 a une profondeur de* 3*. Cette profondeur correspond au chemin le plus long reliant un nœud de l'arbre avec la racine de cet arbre. Ainsi plusieurs nœuds de* t *ont une profondeur de* 3 *comme* 00, 20 *et* 21*. Le nœud* 2 *de* t *a une largeur de* 2 *puisqu'il a deux fils.*

Un arbre est dit d'*arité bornée* si la largeur de chacun de ses nœuds est bornée par une constante donnée, auquel cas il est dit *non borné*. Enfin, un arbre est dit *partiellement étiqueté* si les fonctions d'étiquetage sont partielles sur l'ensemble de définition. Dans ce cas, certains nœuds de l'arbre n'ont pas d'étiquette.

Comme dans le cas des graphes, il est possible d'assouplir la définition d'arbre en omettant les étiquettes présentes sur les nœuds ou bien sur les arcs. De la même manière, on peut supprimer l'ordre imposé entre les nœuds par la relation ns, les arbres ainsi définis sont *non ordonnés*. Dans la suite, on note T_Σ l'ensemble des arbres enracinés, ordonnés et non bornés dont les nœuds sont étiquetés sur un alphabet Σ. Une *forêt d'arbres de* T_Σ est une séquence $(t_1, \cdots, t_n)$ avec $1 \leqslant n$ telle que pour tout $1 \leqslant i \leqslant n$, on a $t_i \in T_\Sigma$.

Nous terminons cette section en faisant le lien entre la notion d'arbre et celle de terme en programmation logique. La notion d'arbre est une notion élémentaire dans de nombreux domaines. On la retrouve également en programmation logique sous la forme de termes. Il est ainsi possible de représenter un arbre $t \in T_\Sigma$ par un unique terme fondé. Dans cette représentation, une constante c correspond alors à un feuille de t dont l'étiquette est le symbole $c \in \Sigma$. Un terme $t = p(t_1, \cdots, t_n)$ est, quant-à-lui, un arbre dont la racine ε a pour étiquette p et pour sous-arbres, les arbres issus des termes $t_1, \cdots, t_n$. Nous emploierons parfois cette représentation d'arbre en terme logique pour plus de commodité dans nos définitions à venir.

Exemple 16. *L'arbre* t *de l'exemple 14 est représenté par le terme suivant :*

$$r(a(b), e, c(c, d))$$

où $r/3,\ a/1, c/2$ *sont des symboles de fonction et* $b,\ e,\ c,\ d$ *des constantes.*

3.1.4 Codages d'arbre

De nombreux codages et représentations pour les arbres ordonnés non bornés existent dans la littérature. Nous nous pencherons particulièrement sur deux d'entres eux dont la particularité est d'être sans perte. Un *codage sans perte* est une fonction prenant en entrée des données qu'elle transforme ensuite en un autre format. L'application de la fonction inverse permet de retrouver à l'identique les données originales à partir de celles codées. Ainsi les données codées et celles originales contiennent les mêmes informations sous une forme différente.

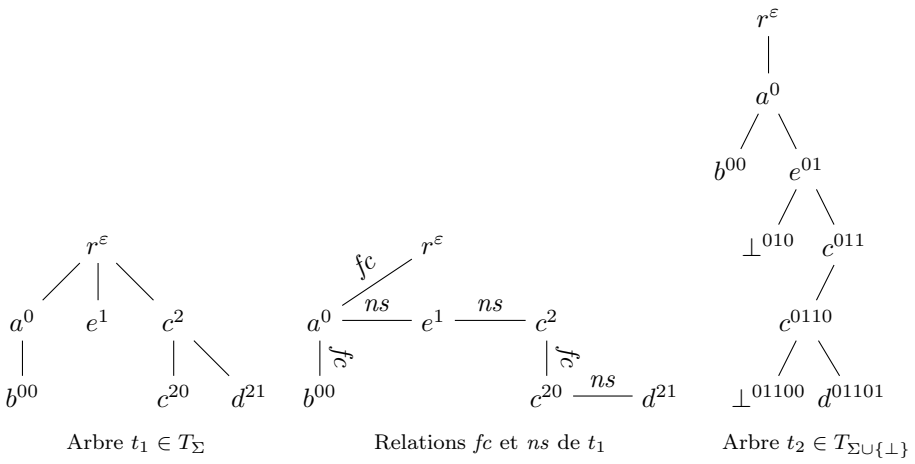
Codage *Firstchild-nextsibling*

Le codage *first-child next-sibling* [Rabin 1969, Comon 1997] a pour but de transformer des arbres ordonnés, étiquetés et d'arité non bornée en arbres ordonnés, étiquetés et d'arité bornée. Ce codage conserve l'étiquetage des nœuds. L'arbre obtenu après codage est binaire, ainsi chaque nœud a, au plus, deux fils. Cette transformation construit l'arbre binaire à partir des arcs des ensembles *fc* et *ns* de l'arbre d'origine. Elle omet ainsi une partie des arcs de l'ensemble *child*. Afin de simplifier la définition de ce codage, nous utilisons la représentation d'arbres sous forme de termes logiques. Le codage d'un arbre $t \in T_\Sigma$ en un arbre respectant le codage *fcns* est réalisé par les fonctions $fcns : T_\Sigma \to T_{\Sigma\cup\{\bot\}}$ et $fcns' : (T_\Sigma)^\star \to T_{\Sigma\cup\{\bot\}}$ définies de la manière suivante :

$$fcns(a(t_1, \cdots, t_n)) = a(fcns'(t_1, \cdots, t_n), \bot) \text{ avec } a, \bot \in \Sigma \cup \{\bot\}$$
$$fcns'((a(t_1^1, \cdots, t_1^m), t_2, \cdots, t_n)) = a(fcns'(t_1^1, \cdots, t_1^m), fcns'(t_2, \cdots, t_n))$$
$$fcns'(()) = \bot$$

La création du nouvel arbre passe par l'ajout de nœuds étiquetés par un symbole $\bot$, non présent dans l'alphabet Σ, afin d'obtenir un arbre ordonné binaire. Nous notons T_Σ^{fcns} l'ensemble des arbres obtenus suite au codage de l'ensemble des arbres T_Σ par la fonction *fcns*. L'exemple 17 illustre l'utilisation de ce codage.

Exemple 17. *Soit deux arbres $t_1 \in T_\Sigma$ et $t_2 \in T_{\Sigma\cup\{\bot\}}$. L'arbre t_2 est obtenu par application du codage* first-child next-sibling *à l'arbre t_1.*



Le passage de l'arbre t_1 à l'arbre t_2 est d'autant plus visible avec la figure 3.1. Celle-ci met en évidence les arcs appartenant aux ensembles fc et ns.

Linéarisation des arbres

Un autre codage possible consiste à transformer un arbre en un mot, on parle alors de *linéarisation d'arbre*. Le mot ainsi obtenu est appelé *mot imbriqué (bien formé)* ou *(well) nested word*. La linéarisation d'un arbre $t \in T_\Sigma$ est un mot de $\hat{\Sigma}^\star$ où $\hat{\Sigma} = \{open, close\} \times \Sigma$. Elle est obtenue par la fonction $lin : T_\Sigma \to \hat{\Sigma}^\star$ suivante :

$$lin(a()) = (open, a)(close, a) \text{ avec } a \in \Sigma$$

$$lin(a(t_1, \cdots, t_n)) = (open, a)lin(t_1) \cdots lin(t_n)(close, a)$$

qui prend en entrée un arbre de T_Σ sous la forme d'un terme logique et retourne un mot de $\hat{\Sigma}^\star$. Nous notons $W_{nested}^{T_\Sigma}$ l'ensemble des mots obtenus par application de la fonction *lin* sur les arbres de T_Σ. L'exemple 18 illustre l'utilisation de la fonction *lin*.

Exemple 18. *Soit le terme représentant l'arbre t issu de l'exemple 16 :*

$$r(a(b), e, c(c, d))$$

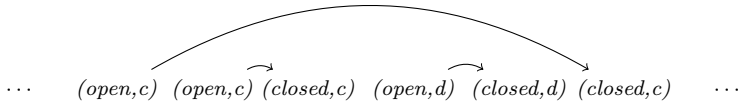
Pour linéariser cet arbre, on commence par définir l'alphabet $\hat{\Sigma}$ *tel que :*

$$\begin{aligned}\hat{\Sigma} \quad = \quad & \{(open, a), (close, a), (open, b), (close, b), (open, c), (close, c),\\ & (open, d), (close, d),\ (open, e), (close, e),\ (open, r), (close, r)\}\end{aligned}$$

On peut alors construire le nested word *suivant :*

$$\begin{aligned}&(open,r)(open,a)(open,b)(close,b)(close,a)(open,e)(close,e)\ \cdots\\ &\cdots\ (open,c)(open,c)(close,c)(open,d)(close,d)(close,c)(close,r)\end{aligned}$$

Ce codage est sans perte, il est donc possible de retrouver les paires de lettres correspondant aux différents nœuds et ainsi reconstruire l'arbre dont est issu le mot. Le schéma suivant représente une partie des liens présents entre les lettres du mot.



Deux symboles reliés par un arc correspondent à un nœud de l'arbre d'origine. Un arc couvert par un autre arc signifie que le nœud de l'arc couvert est un fils du nœud de l'arc couvrant.

3.1.5 Morphisme

Donnons maintenant les définitions de morphisme pour les mots, les graphes et les arbres.

Définition 45 (Homomorphisme et Isomorphisme de mots)**.** *Soit un alphabet* Σ*, un* homomorphisme *h est une fonction* $h : \Sigma^\star \to \Sigma^\star$ *telle que* $\forall x, y \in \Sigma^\star,\ h(xy) = h(x)h(y)$*. Un* isomorphisme *de mots est un homomorphisme bijectif de mots .*

Un arbre étant un graphe particulier, la définition d'homomorphisme d'arbres sera identique à celle de graphes.

Définition 46 (Homomorphisme et Isomorphisme de graphes)**.** *Soit deux graphes ou deux digraphes :* $G_1 = (N_{G_1}, E_{G_1}, lab_\Sigma^{G_1}, lab_\Gamma^{G_1})$ *et* $G_2 = (N_{G_2}, E_{G_2}, lab_\Sigma^{G_2}, lab_\Gamma^{G_2})$*. Un* homomorphisme *entre* G_1 *et* G_2 *est une fonction* $f : N_{G_1} \to N_{G_2}$ *nécessaire à la définition d'une fonction* $f' : E_{G_1} \to E_{G_2}$ *telle que :*

1. $\forall e \in E_{G_1}, f'(e) = f'((n_i, n_j)) = (f(n_i), f(n_j))$ *si* G_1 *et* G_2 *sont des digraphes ou bien* $\forall e \in E_{G_1}, f'(e) = f'(\{n_i, n_j\}) = \{f(n_i), f(n_j)\}$ *si ce sont des graphes.*
2. $\forall e \in E_{G_1},\ f'(e) \in E_{G_2}$,
3. $\forall n \in N_{G_1},\ lab_\Sigma^{G_1}(n) = lab_\Sigma^{G_2}(f(n))$,
4. $\forall e \in E_{G_1},\ lab_\Gamma^{G_1}(e) = lab_\Gamma^{G_2}(f'(e))$.

Un isomorphisme entre deux graphes G_1 et G_2 *est un homomorphisme entre* G_1 *et* G_2 *tel que* f *et* f' *sont des bijections.*

Dans le cas de graphes non étiquetés, les points 3 et 4 de la définition d'homomorphisme n'ont pas de sens et sont donc à omettre.

Exemple 19. *À partir des digraphes* D_1 *et* D_2 *présents dans l'exemple 13, il est possible de définir l'isomorphisme* f *de* D_2 *vers* D_1 *suivant :*

$$f(0) = 0, \quad f(1) = 1, \quad f(2) = 2, \quad f(3) = 3, \quad f(4) = 4,$$
$$f(5) = 5, \quad f(6) = 6$$

La recherche d'un homomorphisme dans le cas général des graphes est un problème NP-complet [Hell 2004]. Le problème d'isomorphisme de sous-graphe, c'est-à-dire décider s'il existe pour deux graphes G_1 et G_2 un sous-graphe de G_1 pour lequel il y a ait un isomorphisme de ce sous-graghe vers G_2, est lui aussi un problème NP-complet [Chartrand 1977].

Les structures de données définies, nous pouvons maintenant présenter le format XML qui tirera profit de ces dernières.

3.2 Documents XML

Le XML [Bray 2008], pour *eXtensible Markup Language*, est un format textuel promu par le World Wide Web Consortium [2] (W3C). Il a été développé dans le but de faciliter le stockage ainsi que l'échange de données structurées ou semi-structurées en champs arborescents, rendant possible une automatisation de ces processus.

Exemple 20 (Document XML représentant un annuaire)**.**

```
<Annuaire>
        <Amis>
             <Personne>
                     <Nom> Dixon  </Nom>
                     <Prénom> Daryl </Prénom>
             </Personne>
             <Personne>
                      <Nom> Grimes </Nom>
                      <Prénom> Rick </Prénom>
             </Personne>
        </Amis>
        <Collègues>
             <Personne>
                     <Nom> Dixon  </Nom>
                     <Prénom> Daryl </Prénom>
             </Personne>
        </Collègues>
</Annuaire>
```

La syntaxe du XML est facilement reconnaissable par son usage de chevrons permettant de former des balises ouvrantes (*<balise>*) ainsi que des balises fermantes (*</ balise>*). On peut notamment observer dans l'exemple 20 la présence de balises comme `<Annuaire>` et `</Annuaire>` mais également celles de données textuelles comme *Daryl* et *Dyxon*. Chaque

2. `http://www.w3.org`

balise, ouvrante ou fermante, possède un nom également appelé *élément*. L'exemple 20 contient ainsi les éléments suivants : `Annuaire`, `Amis`, `Collègues`, `Personne`, `Nom` et `Prénom`.

Les balises ouvrantes et fermantes présentes dans un document XML sont liées deux par deux afin de former des couples contenant une balise ouvrante et une balise fermante. Ainsi, une balise ouvrante est associée à une unique balise fermante du même élément et réciproquement. Une balise ouvrante peut éventuellement contenir des attributs ainsi que des données textuelles. Un *attribut* est une paire clé/valeur écrite sous la forme *clé="valeur"* dénotant des propriétés de l'élément. Une balise $<balise\ attr1 = "val1"\ attr2 = "val2">$ a ainsi un élément *balise* et deux attributs, *attr1* et *attr2*, dont les valeurs sont respectivement *val1* et *val2*.

Les paires de balises ouvrantes et fermantes sont imbriquées sans possibilité de chevauchement comme cela est le cas pour les *nested words*. Il est ainsi possible de représenter un document XML sous la forme d'un arbre ordonné, non borné et étiqueté [Suciu 2002, Neven 2002]. Chaque paire constituée d'une balise ouvrante et d'une fermante correspond alors à un nœud de l'arbre XML étiqueté par l'élément de ces balises. Les éventuelles paires de balises présentes entre ces deux balises forment, quant-à-elle, les sous-arbres des fils de ce nœud. Les attributs ainsi que les données textuelles sont alors représentés dans l'arbre XML par des nœuds. La figure 3.1 correspond à la représentation arborescente du document XML de l'exemple 20.

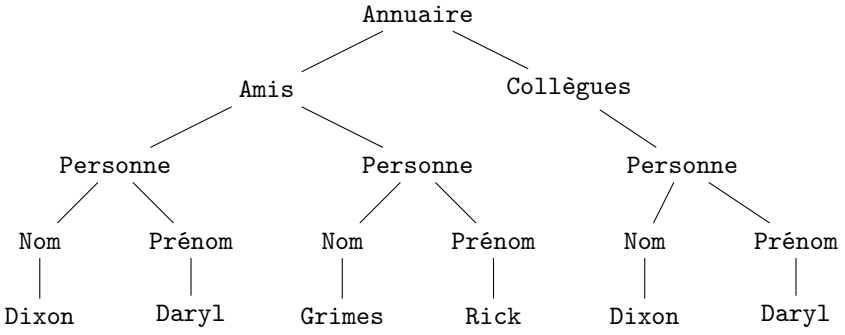


FIGURE 3.1 – Arbre XML construit à partir du document XML de l'exemple 20.

Contrairement au HTML [W3C 1997, W3C 2011], le langage XML a l'avantage d'être qualifié d'extensible. Il permet ainsi à l'utilisateur de définir ses propres balises à la différence des balises statiques du HTML où $<p>$ est la balise représentant un paragraphe et $<li>$ celle d'un élément d'une liste. L'utilisateur peut donc multiplier le nombre de balises à sa guise et emprunter celles des autres utilisateurs en y associant une sémantique pouvant être différente. Il peut également spécifier, sans y être obligé, le schéma de documents XML qu'il définit à l'aide d'un langage de schéma comme XML Schema [Van der Vlist 2002] ou bien encore une DTD. Le but d'un schéma est de définir une classe de documents XML. Il permet de décrire les autorisations d'imbrication et l'ordre d'apparition des éléments et de leurs attributs. Malgré l'intérêt des schémas pour la compréhension et la structuration des documents XML, ces derniers sont rarement spécifiés ou bien encore consultables. Cette remarque est d'autant plus vraie lorsque les documents XML regardés proviennent de sources réelles comme Internet ou appartiennent à de grandes collections de documents XML comme c'est le cas pour ceux du challenge INEX [Alexander 2011]. Ainsi, nous faisons le choix de ne prendre en compte que le document XML en lui même sans prêter attention aux schémas.

En plus d'une facilité d'utilisation, d'extension et de compréhension du format XML, notre attrait pour celui-ci est renforcé par les recommandations du W3C. Ce dernier en-

courage l'usage de la syntaxe XML pour exprimer des langages de balisage spécifique faisant du format XML un standard pour l'échange et le stockage de données. Ainsi, de nombreux langages respectent la syntaxe du XML. C'est le cas du XHTML [W3C 2000, W3C 2010], une adaptation du HTML au XML, pour des données de page Web, du RSS [W3C 2002] pour des données de syndication de contenu, XSLT [W3C 2007c] pour des programmes de transformation d'un document XML, XPATH [W3C 1999, W3C 2007a, Benedikt 2009] et XQUERY [W3C 2007b] comme langage de requêtes ou bien encore les DTDs et XML Schema comme langages de schémas.

Nous terminons avec les documents XML en faisant le lien entre les notions d'arbres et de graphes. Dans l'exemple 20, on remarque qu'une même personne peut être à la fois présente en tant qu'*Amis* mais également *Collègues*. Une telle information peut être représentée par un arbre contenant des variables (X et Y) comme dans la figure 3.2. Ceci permet d'établir des relations entre deux nœuds ne pouvant être liés dans une structure arborescente.

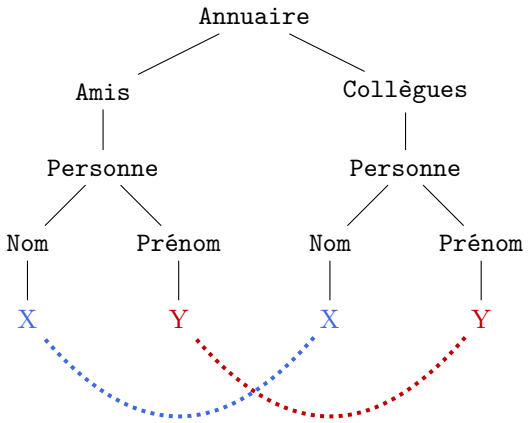


FIGURE 3.2 – Liens implicites présents dans l'arbre XML construit à partir du document XML de l'exemple 20.

L'usage de variables a pour conséquence de changer la structure d'arbre en graphe. Il peut alors être intéressant de bénéficier d'un langage capable de gérer ce genre de généralisation. Cela permettrait de tirer profit des liens implicites existant au sein des données. La PLI est un langage relationnel, il autorise donc ce genre de représentation. Cependant, il ne faut pas perdre de vue qu'une représentation trop riche risque de compliquer l'apprentissage. Représenter toutes les relations implicites dans un arbre n'est donc pas forcément une excellente idée. Désireux d'apprendre à classer et à transformer des documents XML, il convient donc de chercher une représentation de ces documents exploitant une partie ou la totalité de ces informations sans en subir les conséquences. Nous étudierons cela dans les chapitres 5 et 6.

3.3 Apprentissage automatique en PLI

Dans cette section, nous présentons un sous-domaine de l'Intelligence Artificielle qu'est l'apprentissage automatique. Nous introduisons ensuite de manière générique le problème d'apprentissage en PLI puis imposons certaines restrictions sur ce dernier afin de définir différents cadres d'apprentissage.

3.3.1 Apprentissage automatique et supervisé

L'*apprentissage* est une tâche pouvant se résumer à l'application du *principe d'induction*, c'est-à-dire à généraliser un raisonnement à partir de cas singuliers. Il en est de même pour la tâche d'apprentissage automatique. L'objectif de l'apprentissage automatique est d'extraire automatiquement des connaissances à partir d'un ensemble E de données appelé *exemples*. Ces données font ici référence aux cas singuliers nécessaires à l'apprentissage et dépendent d'une représentation devant être homogène entre les exemples. En PLI, on peut, par exemple, considérer qu'un élément de E est un fait ou une clause. L'atome de tête d'un exemple serait alors le concept à apprendre et ceux du corps de la clause des propriétés respectées par cet exemple. Nous notons dorénavant $\mathcal{L}_e$ le langage permettant de décrire les exemples, par conséquent $E \subseteq \mathcal{L}_e$.

Ce problème d'apprentissage automatique se décline en plusieurs variantes dont l'apprentissage supervisé [Mitchell 1997, Cornuéjols 2003]. Les données fournies pour un apprentissage supervisé correspondent à des couples constitués d'un exemple et d'une étiquette. L'étiquette d'un exemple est également appelée *classe*. Elle appartient à un ensemble de classes $\mathcal{C}$ défini pour le problème. Le problème d'apprentissage supervisé consiste alors à apprendre, à partir de ces couples, une fonction de l'ensemble des exemples vers les classes. Comme pour les exemples, il est nécessaire de définir une représentation des fonctions effectuant l'étiquetage. On note $\mathcal{L}_h$ le langage d'hypothèses permettant de décrire l'ensemble de toutes ces fonctions et $h(e) \in \mathcal{C}$ la classe prédite par une hypothèse $h \in \mathcal{L}_h$ et un exemple $e \in \mathcal{L}_e$.

Lorsqu'un apprentissage supervisé ne dispose que de deux classes, on parle d'*apprentissage supervisé de concept* (*concept learning* en anglais), auquel cas il s'agit d'une tâche de *classification*. Nous nous intéressons plus particulièrement à l'apprentissage supervisé de concept. Ainsi, nous employons, par la suite, le terme de classification pour faire référence à la tâche d'apprentissage supervisé de concept. Lors d'une tâche de classification, l'ensemble des couples fournis pour l'apprentissage décrit une hypothèse $c \in \mathcal{L}_h$ appelée *concept*. Le but est alors d'apprendre une hypothèse $h \in \mathcal{L}_h$ telle que pour tout $e \in \mathcal{L}_e,\ h(e) = c(e)$.

Afin de faciliter l'approche de ce problème, il est possible de considérer une hypothèse non plus comme une fonction de l'ensemble des exemples vers l'ensemble des classes mais plutôt comme une représentation d'un ensemble d'exemples. On établit ainsi un mapping des hypothèses de $\mathcal{L}_h$ vers des sous-ensembles d'exemples de $\mathcal{L}_e$. La tâche d'apprentissage consiste alors à trouver une hypothèse décrivant les mêmes exemples que ceux du concept. Ce mapping permet de définir une relation, appelée *relation de couverture*, visant à tester si une hypothèse couvre bien un exemple. On note $couv(h)$ l'ensemble des exemples de $\mathcal{L}_e$ couverts par une hypothèse $h \in \mathcal{L}_h$. Deux hypothèses ayant la même couverture seront *équivalentes vis-à-vis de la relation de couverture*.

Ce problème de classification comporte uniquement deux classes. On peut, sans perte de généralité, fixer l'ensemble des classes $\mathcal{C}$ à $\{+, -\}$. L'association d'une classe à un exemple dépend du concept à apprendre. L'ensemble des exemples E est alors divisé en deux sous-ensembles, E^+ et E^-, en fonction d'un concept $c \in \mathcal{L}_h$ tels que :

— l'ensemble E^+ contient les exemples, appelés *exemples positifs*, couverts par c :

$$E^+ = \{e \in E \mid e \in couv(c)\}$$

— l'ensemble E^- contient les exemples, appelés *contre-exemples*, non couverts par c :

$$E^- = \{e \in E \mid e \notin couv(c)\}$$

où $E = E^+ \cup E^-$. Cette remarque est également vraie pour toute hypothèse $h \in \mathcal{L}_h$,

l'ensemble $\mathcal{L}_e$ peut ainsi être divisé de la même manière :

$$\mathcal{L}_h^+ = \{e \in \mathcal{L}_e \mid e \in couv(h)\}$$
$$\mathcal{L}_h^- = \{e \in \mathcal{L}_e \mid e \notin couv(h)\}$$

avec $\mathcal{L}_e = \mathcal{L}_h^+ \cup \mathcal{L}_h^-$.

La relation de couverture a été étendue afin de définir une relation entre hypothèses, appelée *relation de subsomption*, que l'on note $\succeq$. Cette relation permet de hiérarchiser l'ensemble des hypothèses. Elle indique si l'ensemble des exemples couverts par une hypothèse est égal, inclus ou incomparable à l'ensemble des exemples couverts par une autre hypothèse. Cette hypothèse est ainsi plus générale (ou moins spécifique), égale ou incomparable à cette autre hypothèse. Deux hypothèses h_1 et h_2 sont équivalentes si $h_1 \succeq h_2$ et inversement. Elles couvrent ainsi le même ensemble d'exemples. Lorsque l'ensemble des hypothèses et celui des exemples sont confondus, la relation de subsomption permet également de dire si une hypothèse couvre un exemple.

Par la suite, nous considérons que l'ensemble des couples fournis en entrée de l'apprentissage ne sont pas *bruités*, c'est-à-dire que chaque association classe/exemple vérifie bien le concept à apprendre. On a ainsi $E^+ \cap E^- = \emptyset$. Chaque exemple de l'ensemble E^+ est alors correctement associé à la classe $+$ tandis que les exemples de E^- sont associés à la classe $-$. Il en est de même pour toute hypothèse $h \in \mathcal{L}_h$, c'est-à-dire $\mathcal{L}_h^+ \cap \mathcal{L}_h^- = \emptyset$.

Nous terminons en faisant le lien entre une tâche d'apprentissage supervisé de concept et celle d'exploration d'un espace de recherche. Une tâche de classification peut être vue comme la recherche d'une hypothèse dans un ensemble d'hypothèses satisfaisant un certain nombre de propriétés [Mitchell 1982]. L'ensemble de ces hypothèses est alors nommé *espace de recherche*. Ce problème d'exploration se présente sous la forme d'un tuple $(\succeq, \mathcal{L}_h, \mathcal{L}_e)$ où :

— $\mathcal{L}_e$ est le langage des exemples contenant l'ensemble des exemples E,
— $\mathcal{L}_h$ est le langage des hypothèses
— et $\succeq$ est une relation de subsomption sur $\mathcal{L}_h$

comme définis pour l'apprentissage supervisé de concept. Il consiste à explorer l'*espace de recherche* en tirant partie de l'ordre imposé par $\succeq$ afin de trouver une hypothèse couvrant les exemples de E.

Après avoir défini les notions d'apprentissage supervisé et d'apprentissage supervisé de concept, nous présentons maintenant un cadre générique d'un problème d'apprentissage en PLI, puis faisons le lien entre ce dernier et l'apprentissage supervisé de concept.

3.3.2 L'apprentissage automatique en PLI

Le terme de programmation logique inductive a été pour la première fois mis en avant par Muggleton dans [Muggleton 1991]. Il a, par la suite, proposé une définition de la PLI dans [Muggleton 1994b] reprise et adaptée plusieurs fois dans la littérature [Kietz 1994a, Kietz 1994b]. Un problème d'apprentissage de concept en PLI est défini de la manière suivante :

Définition 47 (Problème d'apprentissage de concept en PLI)**.** *Un* problème d'apprentissage en PLI *est un tuple* $(\vdash, \mathcal{L}_t, \mathcal{L}_e, \mathcal{L}_h)$ *où* $\vdash$ *est une règle d'inférence correcte définie sur un langage de clauses* $\mathcal{L}$*, c'est-à-dire* $\forall C, D \in \mathcal{L}$*, si* $C \vdash D$ *alors* $C \models D$*,* $\mathcal{L}_t \subseteq \mathcal{L}$ *le langage de clauses d'une théorie du domaine* T*,* $\mathcal{L}_e \subseteq \mathcal{L}$ *le langage de clauses d'un ensemble d'exemples* $E = E^+ \cup E^-$ *et* $\mathcal{L}_h \subseteq \mathcal{L}$ *le langage de clauses des hypothèses.*

Un algorithme d'apprentissage en PLI pour $(\vdash, \mathcal{L}_t, \mathcal{L}_e, \mathcal{L}_h)$ *est un algorithme qui prend en entrée une théorie du domaine* T *et un ensemble d'exemples* E *et qui retourne une hypothèse* $h \in \mathcal{L}_h$ *telle que les conditions suivantes sont vérifiées :*

1. *satisfiabilité a priori :* $T \nvdash e$ *pour tout* $e \in E^-$;
2. *satisfiabilité a posteriori (correction) :* $h \wedge T \nvdash e$ *pour tout* $e \in E^-$;
3. *nécessité a priori :* $T \nvdash e$ *pour tout* $e \in E^+$;
4. *condition a posteriori (complétude) :* $h \wedge T \vdash e$ *pour tout* $e \in E^+$.

si elle existe ou null dans le cas contraire.

La satisfiabilité a priori (condition 1) demande à ce que les exemples négatifs ne soient pas déductibles de la théorie du domaine. La satisfiabilité a posteriori (condition 2), également appelée *correction*, exige que l'union du programme logique appris et de la théorie du domaine ne permette pas la déduction des exemples négatifs à l'aide d'une méthode d'inférence. La nécessité a priori (condition 3) impose que les exemples positifs ne peuvent être déduits de la théorie du domaine. Dans le cas contraire, il n'est pas nécessaire de chercher le programme cible puisque la théorie couvre déjà le concept à apprendre. Enfin, la condition a posteriori (condition 4), ou *complétude*, signifie que l'hypothèse associée à la base de connaissances permet de déduire l'ensemble des exemples positifs à couvrir.

De nombreuses contraintes ont été appliquées à la définition 47 afin de proposer des cadres d'apprentissage variés. Ces contraintes portent sur la nature de $\mathcal{L}_h$, $\mathcal{L}_e$ et $\mathcal{L}_t$ ainsi que sur la règle d'inférence $\vdash$. Parmi les sémantiques les plus connues figurent la sémantique normale et la sémantique définie [Muggleton 1994b]. La sémantique normale utilise comme règle d'inférence l'implication logique. Elle permet l'apprentissage de clause et de programme logique comme hypothèse. La sémantique définie est une restriction de la sémantique normale qui n'autorise comme clauses que les clauses définies. Cette restriction syntaxique permet de bénéficier du lien existant entre les clauses définies et les modèles de Herbrand. Elle rend ainsi possible l'utilisation de règles d'inférences propres aux clauses définies comme la SLD-résolution. Pour cette raison, nous la choisissons comme cadre d'apprentissage. Tout comme la sémantique normale, la sémantique définie a permis la naissance d'autres sémantiques plus restrictives. C'est le cas de l'Example Setting proposée par Muggleton dans [Muggleton 1994b]. Cette sémantique reprend la sémantique définie et ajoute la contrainte suivante : l'ensemble des exemples ne doit contenir que des clauses unitaires fondées, c'est-à-dire des clauses ne contenant qu'un seul littéral fondé. Une liste non exhaustive de résultats théoriques sur différents cadres d'apprentissage se trouve dans [Kietz 1994b].

3.3.3 À la recherche d'une hypothèse

Le problème d'apprentissage en PLI peut être vu comme une adaptation du problème d'apprentissage supervisé de concept à la PLI. Ainsi, comme pour un problème d'apprentissage supervisé de concept, il est possible de le ramener à la recherche d'une hypothèse dans un espace de recherche [Mitchell 1997]. Une telle approche repose principalement sur deux points. Le premier point correspond à la règle d'inférence. Il est clair que l'utilisation d'une règle d'inférence intractable n'est pas un choix judicieux. Il convient donc d'en utiliser une dont la complexité est raisonnable. Le second point est la définition des langages de clauses pour les exemples et les hypothèses. L'espace de recherche correspond à un ensemble d'hypothèses, il convient donc de définir un langage d'hypothèses se limitant au mieux à la représentation des concepts à apprendre. Il en est de même pour le langage des exemples et celui de la théorie du domaine. Remarquons que les choix de la règle d'inférence et des langages d'hypothèses, d'exemples et de la théorie sont liés. Il convient donc de trouver un bon compromis entre ces différents points.

Le parcours de cet espace est rendu possible grâce à des opérations de raffinement (de généralisation et/ou de spécialisation) basées sur une relation de subsomption. La recherche

d'une hypothèse dans cet ensemble peut être réalisée de plusieurs manières. Une première, appelée *recherche descendante* (ou top-down), consiste à partir de l'hypothèse la plus générale, la clause vide, et à se diriger vers des hypothèses de plus en plus spécifiques respectant l'ensemble des exemples et la théorie. De nombreux systèmes utilisent cette approche, notamment CIGOL [Muggleton 1988], CLINT [De Raedt 1992], GOLEM [Muggleton 1990] et ALEPH [Srinivasan 2004]. Une seconde approche consiste à partir d'une hypothèse la plus spécifique pour un exemple, appelée *bottom clause*, et à se diriger vers des hypothèses de plus en plus générales. La *clause la plus spécifique (ou bottom clause) d'une clause C par rapport à une théorie T* est une clause $\bot(C)$, la plus spécifique, telle que $\bot(C) \cup T \models C$. On parle alors de *recherche ascendante* (ou bottom-up). Cette approche est utilisée par les systèmes en PLI : FOIL [Quinlan 1990, Quinlan 1993, Quinlan 1995], MOBAL [Kietz 1992], CLAUDIEN [Raedt 1997] et PROGOL [Muggleton 1995]. Enfin, une dernière, appelée *approche mixte*, consiste à mélanger les deux précédentes. On la retrouve notamment dans le système PROGOLEM [Muggleton 2010].

3.4 Identification à la limite

Dans cette section, nous présentons le paradigme de l'identification à la limite introduit par Gold [Gold 1967]. Ce paradigme nécessite un langage d'exemples $\mathcal{L}_e$ dénombrable, un langage d'hypothèses $\mathcal{L}_h$ et une relation de subsomption ou de couverture, notée $\succeq$, entre hypothèses et exemples. Le langage d'exemples contient un ensemble d'exemples E regroupant exemples et contre-exemples d'un concept c à apprendre.

L'identification à la limite a pour but de donner un cadre formel aux algorithmes d'apprentissage, c'est-à-dire aux processus prenant en entrée des exemples et retournant une hypothèse, en leur imposant l'obligation de converger vers le concept à apprendre au bout d'un nombre fini d'exemples.

Commençons par rappeler les notions de base nécessaires à la définition d'apprentissage à la limite. Le vocabulaire suivant diffère légèrement des définitions données par [Gold 1967] pour ne pas rester spécifique à l'inférence grammaticale mais demeure fortement inspiré par [de la Higuera 2010].

Définition 48 (Présentation positive, négative et complète)**.** *Soit un concept $c \in \mathcal{L}_h$, une séquence infinie d'exemples $S_c = e_1, \cdots, e_n, \cdots$ est appelée respectivement* présentation positive, négative ou complète *si chaque élément de S_c est un élément de $\mathcal{L}_c^-$, de $\mathcal{L}_c^+$ ou bien de $\mathcal{L}_e$ et réciproquement.*

Par la suite, nous notons $S_c = e_1, ..., e_n, ..., e_m, ...$ une présentation d'un concept cible $c \in \mathcal{L}_h$. Nous notons également $S_c[1..m]$ la séquence finie issue de S_c contenant du premier au $m^{\text{ième}}$ élément de S_c et $||S_c[1..m]||$ la taille de $S_c[1..m]$. La taille d'une séquence est la somme des tailles des exemples qui la composent.

Revenons maintenant à la notion d'apprenant. Un *apprenant* $\mathcal{A}$ prend en entrée une séquence d'exemples $e_1, \cdots, e_n$ et retourne une hypothèse $h = \mathcal{A}(e_1, \cdots, e_n)$. Cette hypothèse peut, en fonction de la définition de l'apprenant, couvrir ou non les exemples de la séquence donnée en entrée. On parle alors de *correction*. On dit qu'une hypothèse h est correcte vis-à-vis de la séquence $S_c[1..m]$ appartenant à la présentation S_c si pour tout $e \in S_c[1..m]$:

— $h \succeq e$ si $e \in \mathcal{L}_c^+$;

— $h \not\succeq e$ si $e \in \mathcal{L}_c^-$;

L'ensemble des définitions nécessaires à l'identification à la limite posé, nous pouvons maintenant la présenter.

Définition 49 (Identification à la limite)**.** *Un ensemble d'hypothèses $\mathcal{L}_h$ est* identifiable à la limite *si et seulement s'il existe un algorithme d'apprentissage $\mathcal{A}$ pour $\mathcal{L}_h$ tel que, pour chaque concept $c \in \mathcal{L}_h$ et pour toute présentation $S_c = e_1, ..., e_n, ..., e_m, ...$, il existe une borne $n \in \mathbb{N}$ tel que quelque soit $m \in \mathbb{N}$ avec $m \geqslant n$, $\mathcal{A}(S_c[1...m]) = \mathcal{A}(S_c[1...n]) = c$ (modulo l'équivalence).*

Une identification à la limite est par positifs seuls, négatifs seuls ou positifs et négatifs si S_c est une présentation respectivement positive, négative et complète.

Une manière de prouver qu'un ensemble d'hypothèses est apprenable à la limite pour un algorithme donné consiste à définir, pour chaque hypothèse, un plus petit ensemble d'exemples appelé échantillon caractéristique. Une fois donné en entrée de l'apprenant, il permet de retrouver l'hypothèse pour laquelle il a été construit.

Définition 50 (Echantillon caractéristique)**.** *Un ensemble d'exemples EC_c est un* échantillon caractéristique *pour une hypothèse $c \in \mathcal{L}_h$ et un algorithme d'apprentissage $\mathcal{A}$ si pour chaque présentation S_c incluant EC_c, $\mathcal{A}(S_c) = c$ (modulo l'équivalence).*

La taille d'un échantillon caractéristique est un des critères qui permet de déterminer la difficulté d'apprentissage d'un ensemble d'hypothèses $\mathcal{L}_h$ à l'aide d'un algorithme $\mathcal{A}$. La définition d'identification à la limite proposée par Gold ne pose aucune contrainte sur la complexité de l'algorithme, ni sur la taille de l'échantillon caractéristique. C'est pour cette raison qu'a été introduite la notion d'identification polynomiale à la limite en temps et en données.

Définition 51 (Identification polynomiale à la limite en temps et données)**.** *Un ensemble d'hypothèses $\mathcal{L}_h$ est* polynomialement identifiable à la limite en temps et en données *si et seulement s'il existe un algorithme d'apprentissage $\mathcal{A}$ pour $\mathcal{L}_h$ tel que, pour chaque concept $c \in \mathcal{L}_h$ et pour toute présentation $S_c = e_1, ..., e_n, ..., e_m, ...$, il existe deux polynômes $p()$ et $q()$ permettant la vérification des conditions suivantes :*

- *$\mathcal{A}(S_c[1..m])$ retourne une hypothèse $h \in \mathcal{L}_h$ en temps $\mathcal{O}(p(||S_c[1..m]||))$ t.q. h est correcte vis-à-vis de $S_c[1..m]$;*
- *il existe un échantillon caractéristique EC_c de taille inférieure ou égale à $q(|c|)$ t.q. si $EC_c \subseteq S_c$ alors $\mathcal{A}(S_c) = \mathcal{A}(EC_c) = c$ (modulo l'équivalence).*

Cette définition d'identification à la limite peut être rendue moins stricte en supprimant la condition de correction présente dans le premier point de la définition d'identification polynomiale à la limite.

3.5 Évaluation d'un apprentissage

Nous allons maintenant nous intéresser à la manière dont une tâche de classification peut être évaluée. Cette phase d'évaluation est importante puisqu'elle permet de juger de la qualité de l'apprentissage réalisé par un algorithme pour une représentation donnée des exemples et des hypothèses.

3.5.1 Erreur réelle, Erreur empirique et Validation croisée

Nous nous replaçons dans le cadre d'un apprentissage supervisé de concept complet et sans bruit. Rappelons que les données fournies pour un tel apprentissage se présentent sous la forme d'un ensemble S de couples exemple/classe (e_i, c_i), appelés *exemples étiquetés*, étant un sous-ensemble de $\mathcal{L}_e \times \mathcal{C}$. Ainsi, pour tout $(e_i, c_i) \in S$, $c_i = +$ si $e_i \in E^+$, sinon $-$. Le but de cet apprentissage est alors d'induire, d'après un ensemble d'exemples étiquetés,

une hypothèse $h : \mathcal{L}_e \to \mathcal{C}$ s'approchant le plus possible de la fonction cible c (inconnue). L'écart entre l'hypothèse apprise h et le concept cible c est évalué par une fonction de perte l. Une fonction de perte, pour un problème de classification, est une fonction $l : \mathcal{C} \times \mathcal{C} \to \{0, 1\}$ telle que pour deux classes $c_1, c_2 \in \mathcal{C}$, on ait :

$$l(c_1, c_2) = \begin{cases} 0 & \text{si } \ c_1 = c_2, \\ 1 & \text{sinon.} \end{cases}$$

Cette fonction est utilisée pour donner un coût à l'étiquetage d'un exemple par une hypothèse apprise vis-à-vis du concept à apprendre. On pose ainsi généralement $c_1 = h(e)$ et $c_2 = c(e)$ où h est une hypothèse apprise et c le concept à découvrir. Elle a alors pour but de sanctionner l'hypothèse apprise lorsque la classe prédite par h n'est pas celle indiquée par c. Dans le cas d'une classification binaire, la sanction est classiquement de 0 en cas de correspondance et 1 en cas de divergence. On parle d'une *fonction de coût* 0-1.

L'évaluation d'une hypothèse pour un exemple vis-à-vis d'un concept maintenant définie, nous devons la définir pour l'ensemble des exemples. L'idéal est d'évaluer h sur l'ensemble de tous les exemples de $\mathcal{L}_e$. Cette évaluation, appelée *erreur réelle*, est définie de la manière suivante :

$$R_{\text{réelle}}(h) = \mathbb{E}[l(h(e_i), c_j)] = \int_{e_i \in \mathcal{L}_e, c_j \in \mathcal{C}} l(h(e_i), c_j) \times \mathrm{p}_{\mathcal{L}_e \mathcal{C}} \, \mathbf{d}e_i \mathbf{d}c_j$$

où $P_{\mathcal{L}_e \mathcal{C}}$ est la loi de probabilité à deux variables dont les domaines sont respectivement l'ensemble d'exemples et l'ensemble des classes.

En pratique, l'ensemble de tous les exemples peut être infini et la loi de distribution de ces derniers est méconnue, l'erreur réelle ne peut donc être calculée. L'évaluation d'une hypothèse est alors réalisée à partir d'un ensemble fini d'exemples étiquetés S défini précédemment et est appelée *erreur empirique*. L'*erreur empirique* d'une hypothèse h par rapport à un concept c sur un ensemble d'apprentissage $S = \{(e_1, c_1), \cdots, (e_m, c_m)\}$ se calcule ainsi :

$$R_{\text{emp}}(h) = \frac{1}{m} \sum_{k=1}^{m} l(h(e_i), c_i)$$

L'erreur empirique permet ainsi d'obtenir une approximation imparfaite de l'erreur réelle.

Cependant, avant d'être capable de calculer l'erreur empirique, il est nécessaire d'effectuer l'apprentissage à partir d'un ensemble d'exemples étiquetés. Une première approche consiste à utiliser, pour l'apprentissage, l'ensemble de tous les exemples étiquetés et connus, ces derniers étant en nombre fini. Cette approche, appelée *méthode de resubstitution*, a généralement pour effet de diminuer l'erreur empirique au détriment de l'erreur réelle. Ceci s'explique par un surapprentissage [Cornuéjols 2010], c'est-à-dire l'obtention d'une hypothèse ayant trop exploitée les données fournies pour l'apprentissage rendant ainsi l'hypothèse apprise plus spécifique aux données qu'au concept à apprendre.

Afin d'éviter cela, l'ensemble des exemples étiquetés est divisé en deux. On dispose alors d'un *ensemble d'apprentissage* à partir duquel sera réalisé l'apprentissage et d'un *ensemble de test* à partir duquel sera calculée l'erreur. Il existe un nombre exponentiel de couples contenant l'ensemble d'apprentissage et l'ensemble de test. Dans le but de prendre en compte l'erreur empirique sur tous ces couples, la méthode de *validation croisée* a été établie. L'erreur considérée correspond alors à la moyenne de toutes les erreurs empiriques calculées pour les différents couples d'ensemble d'apprentissage et de test. Cependant, tester et évaluer des ensembles d'apprentissage se révèle coûteux, même en se restreignant aux cas du *leave-one-out*, c'est-à-dire aux couples dont l'ensemble d'apprentissage contient tous les exemples à l'exception d'un formant l'ensemble de test [Guyon 2003].

Pour remédier à ce problème, la notion de validation croisée à n plis (n-fold cross-validation) est utilisée. Cette dernière consiste à :

- diviser l'ensemble des exemples en plusieurs sous-ensembles tels que $S = S_1 \cup \cdots \cup S_n$ et $|S| = |S_1| + \cdots + |S_n|$.
- réaliser l'apprentissage sur l'ensemble $S \setminus S_i$.
- évaluer le résultat d'apprentissage sur l'ensemble S_i afin d'obtenir un score jaugeant la qualité de l'apprentissage.
- effectuer ces opérations une fois pour chaque ensemble S_i avec $1 \leqslant i \leqslant n$ et faire la moyenne des scores.

Il a été constaté empiriquement que cette technique d'évaluation donne de meilleurs résultats que la validation *leave-one-out* [Breiman 1992]. Plus de détails sur la technique de validation croisée se trouvent dans [Kohavi 1995, Cornuéjols 2003].

3.5.2 Rappel, précision et F-mesure

L'erreur empirique donne le pourcentage d'erreur sur la classification effectuée. Cependant cela n'est pas toujours représentatif d'un apprentissage. En effet, il peut être utile de savoir si un taux d'erreur élevé provient d'exemples positifs reconnus comme négatifs, d'exemples négatifs considérés comme positifs ou des deux. Ainsi, d'autres mesures qualitatives ont été proposées. Nous allons maintenant définir les notions de *rappel* et de *précision* qui sont des mesures statistiques issues du domaine de la *Recherche d'Information*. Elles sont encore récemment utilisées pour la classification de documents XML [Wu 2008, Wu 2012] et restent des mesures très populaires permettant d'évaluer une tâche de classification. Ces mesures portent sur une tâche de classification binaire. On a ainsi deux classes $\mathcal{C} = \{+, -\}$. On définit tout d'abord quelques notions nécessaires à la précision et au rappel.

On note :

- VP les *vrais positifs*, c'est-à-dire le nombre d'exemples de l'ensemble de test et de classe positive correctement assignés à la classe positive par l'hypothèse,
- FP les *faux positifs*, c'est-à-dire le nombre d'exemples de l'ensemble de test et de classe négative assignés à la classe positive par l'hypothèse,
- FN les *faux négatifs*, c'est-à-dire le nombre d'exemples de l'ensemble de test et de classe positive assignés à la classe négative par l'hypothèse,
- VN les *vais négatifs*, c'est-à-dire le nombre d'exemples de l'ensemble de test et de classe négative correctement assignés à la classe négative par l'hypothèse.

La précision et le rappel peuvent être maintenant définis.

$$\text{rappel } = \frac{VP}{VP + FN}$$

Le *rappel* a pour but d'indiquer la proportion d'exemples positifs classés par l'hypothèse apprise comme positifs sur l'ensemble de tous les exemples positifs qui auraient dus être classés comme positifs. Cette mesure permet d'évaluer la capacité du système à identifier un maximum d'exemples positifs comme tel. Plus le rappel est élevé, moins il y a de risque que le système ait oublié des exemples positifs.

$$\text{précision } = \frac{VP}{VP + FP}$$

La *précision* indique la proportion de vrais positifs, c'est-à-dire d'exemples correctement assignés à la classe positive, sur l'ensemble des exemples considérés comme positifs par l'hypothèse apprise. Cette mesure donne une idée de la confiance que l'on peut avoir dans

l'attribution de la classe positive aux exemples à classer comme tel. Plus la précision est élevée, plus l'attribution d'une classe à un exemple a de chance d'être correcte.

Le but d'un apprentissage est donc d'obtenir à la fois un bon rappel et une bonne précision. Ce constat fait, une mesure, appelée la F-mesure, a été proposée par [Rijsbergen 1979] afin d'évaluer un système en prenant en compte à la fois le rappel et la précision. La *F-mesure* est une moyenne harmonique du rappel et de la précision.

$$\text{F-mesure } = \frac{(1+\beta^2) \times \text{rappel} \times \text{précision}}{\beta^2 \times \text{rappel} + \text{précision}} \qquad \text{avec } \beta > 0$$

On fixe généralement la valeur de β à 1. Ceci permet d'accorder un poids égal à la précision et au rappel. La F-mesure est maximisée lorsque le rappel et la précision sont égaux.

Dans le but d'évaluer la classification de plusieurs classes, la stratégie du *one-vs-all*, encore appelée *one-vs-rest*, a été proposée [Rifkin 2004]. Elle consiste en l'apprentissage d'une hypothèse pour chaque classe. L'ensemble des exemples étiquetés nécessaire à l'apprentissage binaire d'une classe est alors obtenu en considérant que les exemple positifs sont ceux couverts par la classe traitée. Les autres exemples forment l'ensemble des négatifs. Disposant d'une hypothèse pour chaque classe, on peut alors calculer pour chacune d'entre elle une F-mesure. La moyenne, pondérée ou non par le poids de chaque classe, est alors calculée afin de jauger la qualité de l'apprentissage. Dans le cas d'une moyenne pondérée, le poids d'une classe peut être, par exemple, le nombre d'exemples présents dans cette classe ou bien son importance par rapport au problème étudié. La moyenne non pondérée sera appelée *macro-moyenne* tandis que la moyenne pondérée sera nommée *micro-moyenne*. Cette dernière a l'avantage de prendre en compte l'importance d'une classe contrairement à la première qui accorde autant d'importance à chaque classe. Le choix de l'une ou l'autre dépendra donc de l'application considérée.

3.6 Conclusion

Dans ce chapitre, nous avons rappelé les définitions de différentes structures de données que sont les mots, les graphes et les arbres. Un codage d'arbres en mots imbriqués, les *nested words*, et un autre, le codage *first-child next-sibling*, pour la représentation d'arbres d'arité non bornée en arbres d'arité bornée ont été définis. Puis, nous avons présenté le format bien connu de documents semi-structurés que constitue le XML.

Le problème de classification a ensuite été posé et le lien avec la programmation logique établi. Une résolution de ce problème par l'exploration d'un espace de recherche a été introduite. Il a été mis l'accent sur l'importance des choix de représentation ainsi que celui de la relation de subsomption guidant la recherche. Un cadre formel, l'identification à la limite, a ensuite été exposé afin d'être utilisé pour la classification de documents XML. Enfin, les mesures permettant l'évaluation de nos résultats d'apprentissage ont été définies.

Ces notions de bases maintenant posées, nous pouvons nous atteler à une étude préliminaire portant sur les relations de subsomption ainsi que sur les différentes restrictions syntaxiques développées dans la littérature. Ceci nous permettra d'entrevoir les différents problèmes existants pour une tâche de classification et ainsi proposer une solution adéquate à notre problème de classification de documents XML.

Chapitre 4

Relations de subsomption et généralisations

Sommaire

Un problème de classification en PLI peut être vu comme la recherche d'une hypothèse dans un espace de recherche. Ce dernier est l'ensemble de toutes les hypothèses possibles autorisées. Il est fréquent que cet espace soit très vaste, voire infini. Ainsi, il est généralement créé au fur et à mesure de l'induction de nouvelles hypothèses. Le défi d'un bon système de PLI est alors de parcourir le moins d'hypothèses possibles dans l'espace de recherche avant d'en trouver une satisfaisante.

Afin de simplifier la recherche dans cet espace, plusieurs biais peuvent être envisagés et combinés. Ils ont pour intérêt la définition de l'espace de recherche en imposant des contraintes sur les exemples et les hypothèses que l'algorithme peut considérer. Ils orientent également le choix de la relation de subsomption qui est définie pour un ensemble d'exemples et un d'hypothèses. Le rôle du biais est double :

— réduire la taille de l'espace de recherche afin d'améliorer le temps de réponse du système ;
— garantir une certaine qualité de l'apprentissage en interdisant au système de considérer certaines hypothèses inutiles.

L'un des premiers biais, lors de la conception d'un système en PLI, est le choix des langages d'exemples et d'hypothèses. On peut ensuite appliquer à ces langages d'autres biais, appelés *biais déclaratifs* ou *de langage*, que sont les biais syntaxiques et les biais sémantiques :

— Un *biais syntaxique* est un ensemble de contraintes sur la forme des hypothèses : leur taille, le nombre maximal de variables dans chaque hypothèse, le nombre maximal de littéraux, les prédicats autorisés dans les hypothèses, ...
— *Un biais sémantique* contraint le sens et l'utilisation des hypothèses. Il peut consister en un typage des arguments des atomes, une déclaration de modes (entrée/sortie) (cf. Aleph [Srinivasan 2004]), ...

L'adéquation des biais d'un système de PLI conditionne le succès de l'apprentissage. Si les contraintes imposées par les biais sont trop fortes ou mal ciblées, la (ou les) hypothèse(s) décrivant le concept cible peuvent être exclues de l'espace de recherche ou rendues difficiles à trouver. L'expression de biais adaptés à un problème spécifique d'apprentissage est donc une étape cruciale et non triviale lors de l'application d'un système d'apprentissage. Ceci est d'autant plus vrai en PLI, où la taille de l'espace de recherche potentiel peut aboutir à un échec de l'apprentissage si le biais est trop faible. Il en est de même du choix des langages d'exemples et d'hypothèses et de la relation de subsomption. Cette dernière a pour objectif de guider la recherche dans l'espace de recherche en l'organisant. Son choix est donc également critique et dépend de ces langages.

Un ensemble d'exemples pour notre problème de classification ou celui de transformation sera constitué d'instances du concept à apprendre. Pour cette raison, nous nous plaçons dans la sémantique définie [Muggleton 1994b] et représentons nos exemples par des faits fondés. Notre apprentissage consistera alors à trouver, dans l'espace de recherche, une hypothèse qui généralise ces exemples. Il est alors nécessaire de bénéficier d'un opérateur de généralisation définis pour la relation de subsomption choisie.

Dans ce chapitre, nous nous intéressons en premier lieu aux principales relations de subsomption utilisées en logique du premier ordre. Nous les étudierons et présenterons leurs avantages ainsi que leurs inconvénients. Nous détaillerons ensuite les principaux opérateurs de généralisation associés à ces relations. Enfin, nous définirons des restrictions syntaxiques, couramment utilisées sur les clauses, rendant possible l'apprentissage de programmes logiques.

4.1 La relation de subsomption

Une relation de subsomption entre hypothèses permet d'établir une hiérarchie entre les hypothèses. En PLI, les relations de subsomption sont généralement des quasi-ordres sur l'ensemble des hypothèses, ce dernier inclut l'ensemble des exemples. Les relations de couverture et de subsomption sont alors les mêmes. Une relation de subsomption permet de savoir si une clause est plus générale, équivalente ou incomparable à une autre clause. Rappelons qu'une hypothèse h est plus générale qu'une hypothèse h' ou h' est plus spécifique que h, notée $h \succeq h'$, si l'ensemble des exemples couverts par h inclut ceux couverts par h'. Si les ensembles d'exemples sont équivalents, elles sont équivalentes, sinon incomparables. La couverture des exemples dépend de la sémantique utilisée. Dans le cas de la sémantique normale (cf. chapitre 3), les hypothèses et les exemples sont des clauses et l'implication logique ($\models$) est utilisée comme relation de couverture et de subsomption.

Il peut exister, au sein de l'ensemble des hypothèses et pour une relation de subsomption donnée, plusieurs hypothèses équivalentes sous cette relation. L'ensemble de toutes les hypothèses équivalentes à une même hypothèse forme alors *une classe d'équivalence*. Ce regroupement des hypothèses en ensembles permet ainsi de transformer l'espace de recherche et de le réduire. Cependant, une telle transformation rend inutilisable la relation de subsomption qui n'est possible qu'entre deux hypothèses. Une classe d'équivalence contenant un ensemble d'hypothèses équivalentes, on peut alors rechercher dans ce dernier une hypo-

thèse correspondant à une représentation canonique de cet ensemble. Cette hypothèse est appelée *hypothèse réduite*. La construction de ces formes canoniques a cependant un coût dépendant de la relation de subsomption, du langage d'hypothèses et de celui d'exemples. Elle peut donc conférer un avantage ou un inconvénient lors d'un apprentissage.

Il existe de nombreuses relations de subsomption en PLI. Certaines sont naturelles comme l'implication logique, d'autres sont intéressantes d'un point de vue computationnel comme la subsomption stochastique [Sebag 1997b]. Ainsi, nous allons passer en revue certaines relations de subsomption, plus ou moins connues, que sont : l'implication logique, la théta-subsomption, l'OI-subsomption, ... ainsi que les formes canoniques associées.

4.1.1 Implication

L'implication est la relation de couverture et de subsomption la plus intuitive pour hiérarchiser les clauses.

Définition 52 (Implication)**.** *Une clause C* implique *(logiquement) une clause D, notée $C \models D$, si et seulement si chaque modèle de C est un modèle de D, c'est à dire $\{C\} \models D$. Deux clauses C et D sont (logiquement) équivalentes , notées $C \equiv D$, si $C \models D$ et $D \models C$.*

Cette relation est réflexive et transitive ce qui nous fait bénéficier d'un quasi-ordre sur les clauses et par conséquent de classes d'équivalence sous ce quasi-ordre. La forme canonique d'une classe de clauses équivalentes est appelée : *clause réduite sous implication*. On peut définir la notion de clause réduite sous équivalence logique comme suit.

Définition 53 (Réduction sous implication)**.** *Un littéral ℓ d'une clause C est redondant sous implication logique si $C \models C \setminus \{\ell\}$. Une clause est* réduite sous implication *logique si C ne contient aucun littéral redondant.*

S'agissant d'une forme canonique, une clause réduite ne contient aucun littéral inutile qui serait équivalent à un autre littéral de cette même clause.

Le domaine d'application de l'implication ainsi que sa présence dans la définition d'un problème en PLI (cf. chapitre 3) nous indiquent que cette dernière est un bon candidat pour la subsomption. Cependant, il a été prouvé que l'implication entre clauses est un problème indécidable [Schmidt-Schauß 1988]. Cela signifie que pour deux clauses C et D choisies arbitrairement, il n'existe pas de procédure permettant de savoir si $C \models D$. On pourrait penser que la classe des clauses est trop générale et qu'une restriction à une sous-classe des clauses permettrait d'outrepasser ce problème de décidabilité. La classe des clauses de Horn a notamment été étudiée dans ce but. Or, le résultat de [Marcinkowski 1992] a clôturé cette piste en prouvant que l'implication entre clauses de Horn est également un problème indécidable. Cela se traduit dans la procédure de SLD-résolution par l'existence d'un arbre SLD infini.

De nombres propositions ont été faites afin d'apporter des solutions plus ou moins satisfaisantes à ce problème. Ainsi, on retrouve des restrictions sur la forme de clauses assimilables à des biais syntaxiques dont nous allons rappeler quelques résultats théoriques. Ces résultats sont en partie tirés de [Nienhuys-Cheng 1996b] et utilisent la notion de skolémisation :

Définition 54 (Skolémisation)**.** *Soit C une clause et S un ensemble fini de clauses, $\{X_1, \cdots, X_n\}$ l'ensemble des variables apparaissant dans C et $\{a_1, \cdots, a_n\}$ un ensemble de constantes n'apparaissant pas dans S. On appelle substitution de skolémisation de C respectant S une substitution $\theta = \{X_1/a_1, \cdots, X_i, /a_i, \cdots X_n, a_n\}$. Si $S = \emptyset$, alors θ est une* substitution de skolémisation *de C.*

Le respect d'une substitution de skolémisation peut être facilement étendu à un ensemble de clauses en considérant que cette substitution respecte toutes les clauses de cet ensemble. Enfin, l'application d'une substitution de skolémisation à une clause permet de rendre fondée la clause. C'est sur ce principe que reposent les résultats suivants de [Nienhuys-Cheng 1996a] :

Lemme 1. *Soit T un ensemble de clauses, C une clause et θ une substitution de skolémisation de C respectant T, on a $T \models C$ si et seulement si $T \models C\theta$.*

Lemme 2. *Soit T un ensemble de clauses fondées et C une clause fondée, le problème $T \models C$ est décidable.*

Cette restriction à des clauses fondées s'explique par le lien qu'a l'implication avec les modèles de Herbrand. En effet, on sait que pour un ensemble de clauses T et une clause fondée C, si $T \models C$ alors il existe un ensemble fini de clauses fondées T' étant des instances de T tel que $T' \models C$. Une preuve de ceci peut être trouvée dans [Nienhuys-Cheng 1996a]. D'autres restrictions ont été considérées sur la forme des clauses manipulées. On retrouve notamment une restriction limitant l'arité des symboles de fonction utilisés dans les clauses.

Lemme 3. *Soit une ensemble de clauses sans fonction T et une clause C, le problème $T \models C$ est décidable.*

Ces résultats nous montrent que l'utilisation de l'implication peut être rendue possible en restreignant le type de clauses utilisées. Dans le premier cas, il s'agit de ne considérer que des théories et des clauses libres de variables tandis que, dans l'autre cas, on interdit l'utilisation de fonctions. Ces résultats sont intéressants d'un point de vue théorique puisqu'ils montrent que l'implication peut être, lorsque des restrictions sont faites sur la nature des clauses, rendue décidable. Cependant, malgré ces restrictions, l'usage de l'implication n'en demeure pas moins très coûteux comme l'ont montré [Boytcheva 2000, Nienhuys-Cheng 1996b]. Ainsi dans le cas des clauses Datalog, c'est-à-dire des clauses de Horn sans fonction, le problème d'implication consistant à décider si une clause Datalog en implique une autre est EXPTIME-complet [Gottlob 2003]. Le choix de l'implication logique n'est donc pas à conseiller pour un usage expérimental mais n'en demeure pas moins intéressant d'un point de vue théorique.

Les résultats donnés précédemment ne prennent pas en compte la présence d'une théorie du domaine. Afin d'y remédier, la notion d'implication relative à une théorie a été définie.

Définition 55 (Implication relative à une théorie)**.** *Une clause C* implique *une clause D* relativement à une théorie *T, noté $C \models_T D$, si $\{C\} \cup T \models D$. Deux clauses C et D sont équivalentes relativement à une théorie T, notées $C \equiv_T D$, si $C \models_T D$ et $D \models_T C$.*

Tout comme l'implication, l'implication relative est réflexive et transitive. Il s'agit donc d'un quasi-ordre. On peut alors définir, comme précédemment, la notion de clause réduite sous implication relative.

Définition 56 (Réduction sous implication relative)**.** *Soit une théorie T, un littéral ℓ d'une clause C est* redondant sous implication logique relativement à *T si $C \wedge T \models C \setminus \{\ell\}$. Une clause est* réduite sous implication logique relativement à une théorie *T si C ne contient aucun littéral redondant.*

Malgré l'attrait de la notion de clause réduite, le résultat de décidabilité de l'implication rend inintéressante, d'un point de vue pratique, l'implication relative. Nous continuons cependant à préciser le lien entre l'implication et l'implication relative. Il est facile de voir que si $C \models D$ alors $C \models_T D$ où T est choisi arbitrairement. Néanmoins, la réciproque n'est pas vraie comme nous le montre l'exemple suivant :

Exemple 21. *Soit deux clauses* $C = p(a) \leftarrow p(b)$ *et* $D = p(a) \leftarrow$ *et une théorie* T *contenant l'unique clause* $E = p(b) \leftarrow$*, on a* $C \models_T D$ *mais* $C \not\models D$.

L'implication logique semblait donc être un bon candidat comme relation de subsomption. Cette dernière est notamment utilisée pour la définition d'un problème de classification en PLI. Cependant, les résultats d'indécidabilité et de décidabilité portant sur cette dernière nous montrent l'utilité de restrictions sur la forme des clauses manipulées.

4.1.2 θ-subsomption

Nous allons maintenant parler d'une relation, très populaire en PLI, proposée par Plotkin : la *θ-subsomption* [Robinson 1965, Plotkin 1970].

Définition 57 (θ-subsomption et θ-équivalence)**.** *Une clause* C θ-subsume *une clause* D*, notée* $C \succeq_\theta D$*, si et seulement s'il existe une substitution* θ *telle que* $C\theta \subseteq D$*. Deux clauses* C *et* D *sont* θ-équivalentes *ou* équivalentes sous θ-subsomption*, notées* $C \sim_\theta D$*, si et seulement si* $C \succeq_\theta D$ *et* $D \succeq_\theta C$.

La θ-subsomption est une relation réflexive et transitive. Elle correspond de ce fait à un quasi-ordre et bénéficie de classes d'équivalence de clauses. Ce point est intéressant car deux clauses peuvent être équivalentes sous θ-subsomption sans être des variantes alphabétiques.

Exemple 22. *Considérons les clauses suivantes :*

$$\begin{array}{lcl} C & = & c(X_1) \leftarrow fc(X_1, X_2) \wedge ns(X_2, X_3) \wedge fc(Y_1, Y_2) \wedge ns(Y_2, Y_3). \\ D & = & c(Z_1) \leftarrow fc(Z_1, Z_2) \wedge ns(Z_2, Z_3). \\ E & = & c(X_1) \leftarrow fc(X_1, X_2). \end{array}$$

On peut voir que $C \succeq_\theta D$ *avec* $\theta = \{X_1/Z_1, X_2/Z_2, X_3/Z_3, Y_1/Z_1, Y_2/Z_2, Y_3/Z_3\}$ *mais également* $D \succeq_\theta C$ *avec* $\theta = \{Z_1/X_1, Z_2/X_2, Z_3/X_3\}$*. On a alors* $C \sim_\theta D$*. Enfin, on remarque que* $E \succeq_\theta D$ *avec* $\theta = \{X_1/Z_1, X_2/Z_2\}$ *mais que* $D \not\succeq_\theta E$ *et que* $E \succeq_\theta C$ *avec* $\theta = \{X_1/X_1, X_2/X_2\}$ *mais que* $C \not\succeq_\theta E$.

Intéressons nous maintenant à l'utilisation possible d'une telle relation. Le théorème 6 donne un résultat de décidabilité pour la θ-subsomption entre clauses. Ce dernier a été prouvé par Robinson dans [Robinson 1965].

Théorème 6. *Soit deux clauses* C *et* D*, il existe une procédure qui décide si* $C \succeq_\theta D$.

Le choix de cette opération comme test de subsomption se justifie par son lien avec l'implication logique. Ce lien est décrit par le théorème qui suit.

Théorème 7. *Soit deux clauses* C *et* D*, si* $C \succeq_\theta D$ *alors* $C \models D$

Cependant la réciproque n'est pas vraie et il existe certains cas où $C \not\succeq_\theta D$ alors que $C \models D$. Ce genre de situations arrive lorsque D est une clause self-résolvante (self-resolving) ou bien que C est une tautologie [Kietz 1994b]. Prenons l'exemple suivant :

$$p(s(X)) \leftarrow p(X).$$

$$p(s(s(X))) \leftarrow p(Y).$$

On peut facilement montrer qu'une des deux clauses implique l'autre alors que ce n'est pas le cas avec la θ-subsomption. On en déduit donc que, si l'utilisation de la θ-subsomption est restreinte à des clauses n'étant ni self-resolvantes, ni des tautologiques, alors le théorème 7 peut être étendu de la manière suivante :

$$C \succeq_\theta D \text{ si et seulement si } C \models D$$

comme c'est le cas dans [Gottlob 1987, Muggleton 1992, Kietz 1994a]. Pour résumer, la θ-subsomption est donc correcte et complète pour des clauses de Horn non récursives.

Nous allons maintenant donner les algorithmes qui constituent cette procédure de décision. Le test de subsomption d'une clause C à une clause D revient à tester chaque fonction totale des littéraux de C vers les littéraux compatibles de D jusqu'à trouver celle autorisant l'inclusion. L'assignation des littéraux de C vers ceux de D se fait en commençant de la gauche vers la droite et à l'aide d'un parcours en profondeur. L'ordre des littéraux des clauses a ainsi une importance sur l'efficacité ou l'inefficacité de la θ-subsomption.

Algorithme 2 θ-subsomption.

```
function sub_atom(ℓ, ℓ', var θ)
 1: let ℓ = R(t_1, ..., t_n);
 2: let ℓ' = P(s_1, ..., s_m);
 3: if R ≠ P or n ≠ m
        or have different signs then
 4:     return false;
 5: θ' := θ;
 6: for i ∈ {1, ..., n} do
 7:     if sub_term(t_i, s_i, θ) then
 8:         continue;
 9:     θ := θ';
10:     return false;
11: end for
12: return true;
end function

function sub_clause(C, D, θ)
13: if C = ∅ then
14:     return true;
15: choose ℓ ∈ C;
16: for ℓ' ∈ D do
17:     θ' := θ;
18:     if sub_atom(ℓ, ℓ', θ') and
19:         sub_clause(C \ {ℓ}, D, θ')
20:     then return true;
21: end for
22: return false;
end function

function sub_term(t, s, var θ)
23: if t = s then
24:     return true;
25: if t = X then
26:     if ∄v v/X ∈ θ then
27:         θ := θ ∪ {s/X};
28:     if s/X ∈ θ then;
29:         return true;
30: if t = R(t_1, ..., t_n)
31: and s = R(s_1, ..., s_n) then
32:     for i ∈ {1, ..., n} do
33:         if not(sub_term(t_i, s_i, θ)) do
34:             return false;
35:     end for
36:     return true;
37: return false;
end function

function sub(C, D)
38: return sub_clause(C, D, ∅);
end function
```

Pour deux clauses C et D, l'algorithme 2 a une complexité en temps de $\mathcal{O}(|D|^{|C|})$ puisque l'on doit, au pire, construire toutes les fonctions des littéraux de C vers ceux de D en respectant la contrainte de compatibilité des littéraux. Le théorème suivant donne plus d'indications sur la difficulté d'un test de subsomption et a été prouvé dans [Kietz 1994b, Karp 1972b].

Théorème 8. *La θ-subomption entre deux clauses de Horn est un problème NP-complet.*

La complexité de la θ-subsomption est ainsi, dans le pire des cas, exponentielle dans la taille des clauses. Ce résultat n'est pas étonnant puisque la θ-subsomption est équivalente à

un problème d'homorphisme de graphes [Plotkin 1970]. Celui-ci est également un problème NP-complet. Cette complexité ainsi que l'usage fréquent dans les systèmes de PLI (MIS [Shapiro 1991], PROGOLEM [Muggleton 2010]) de la subsomption ont motivé la recherche d'algorithme efficace. C'est le cas de Django [Maloberti 2004], Resumer2 [Kuželka 2009] ou Subsumer [Santos 2010].

La θ-subsomption est un quasi-ordre. On peut donc, comme pour l'implication logique, définir une représentation canonique pour chaque classe d'équivalence de clauses sous θ-subsomption. Ce type de clauses a été identifié par Plotkin et nommé *clause réduite sous θ-subsomption.*

Définition 58 (Réduction sous θ-subsomption)**.** *Un littéral ℓ d'une clause C est* redondant sous θ-subsomption *si et seulement si $C \succeq_\theta C \setminus \{\ell\}$. Une clause* réduite sous θ-subsomption *est une clause ne contenant aucun littéral redondant sous θ-subsomption.*

Gottlob et Fermüller [Gottlob 1993] ont démontré que décider si une clause est réduite ou non est un problème co-NP-complet. Cela s'explique en présentant la réduction comme une succession de plusieurs tests de θ-subsomption, comme le montre l'algorithme 3 (reduce).

Algorithme 3 Réduction.

```
function reduce(C)
1: R := C ;
2: for ℓ ∈ C do
3:     if sub(R, R \ {ℓ}) then
4:         R := R \ {ℓ} ;
5: end for
6: return R ;
end function
```

La complexité de reduce dépend en effet directement de la complexité de la θ-subsomption. Ainsi, dans le cas de clauses de Horn, le test de subsomption est en $|C|^{\mathcal{O}(|C|)}$ (exponentiel). Malgré ce problème de complexité, la réduction permet une meilleure lisibilité des clauses manipulées, un gain computationnel que ce soit en temps ou en espace lors de plusieurs tests θ-subsomption successifs et une réduction de la taille de l'espace de recherche.

La présence d'une théorie est exclue de la définition de θ-subsomption. Elle a ainsi été étendue par [Plotkin 1971b] pour en prendre une en compte. Cette opération s'appelle la θ-subsomption relative à une théorie. Elle repose sur la θ-subsomption dont l'utilisation est complétée à l'aide d'une technique de dérivation appelée C-dérivation. Une *C-dérivation de R à partir de C et T* est une résolution de R à partir de C et T où la clause C est utilisée au plus une fois. Il n'est pas rare que cette définition soit rendue plus stricte en limitant l'usage de la clause C a exactement une fois comme dans [Raedt 1996, Muggleton 1990].

Définition 59 (θ-subsomption relative)**.** *Une clause C θ-subsume une clause D* relativement à une théorie T*, noté $C \succeq_{\theta,T} D$, si et seulement s'il existe une C-dérivation d'une clause R à partir de C et T telle que R θ-subsume D.*

Remarquons que si une clause C θ-subsume D alors la clause C θ-subsume D relativement à une théorie T en choisissant $R = C$. La θ-subsomption relative conserve les propriétés de réflexivité et de transitivité issues de la θ-subsomption. On peut donc, comme pour la θ-subsomption, construire un ensemble de classes d'équivalence permettant de restreindre la taille de l'espace de recherche. Avant de montrer le lien qu'elle a également avec la l'implication logique, nous donnons un théorème décrivant sa relation avec le θ-subsomption sans théorie pour une classe restreinte de clauses : les clauses de Horn.

Théorème 9. *Soit C et D deux clauses de Horn et $T = \{\ell_1, \cdots, \ell_n\}$ un ensemble fini de littéraux fondés, C θ-subsume D relativement à T, noté $C \succeq_{\theta,T} D$, si $C \succeq_\theta (D \cup \{\neg\ell_1, \cdots, \neg\ell_n\})$.*

Connaissant le lien existant entre la θ-subsomption et l'implication, nous pouvons maintenant établir une relation entre l'implication et la θ-subsomption relative à une théorie.

Théorème 10. *Si une clause C θ-subsume une clause D relativement à une théorie T alors $T \cup \{C\} \models D$.*

Cette propriété permet l'utilisation de la θ-subsomption relative à une théorie dans le cadre de système d'inférence. La θ-subsomption relative utilise une dérivation, cette dernière peut donc être coûteuse. Afin de réduire ce coût, la base de connaissances est réduite en pratique à un ensemble de faits fondés. Cela permet d'exploiter le théorème 9. Malgré cette restriction, la θ-subsomption relative a été utilisée pour constituer la base d'un système d'inférence connu : GOLEM [Muggleton 1990].

Dans le cas de clauses non récursives, la θ-subsomption est équivalente à l'implication. De plus, et comme l'implication, elle peut également bénéficier de la prise en compte d'une théorie du domaine. Elle correspond donc à une bonne alternative à l'implication logique. Cependant, la complexité de la θ-subsomption n'en demeure pas moins élevée puisqu'il s'agit d'un problème NP-complet dans le cas général des clauses. Nous continuons donc notre investigation à la recherche d'une relation de subsomption respectant au mieux l'implication logique et offrant un coût d'utilisation raisonnable.

4.1.3 OI-subsomption

Afin de réduire la complexité de la θ-subsomption, de nombreuses contraintes lui ont été appliquées et des alternatives définies. Parmi ces alternatives, on retrouve l'*OI-subsomption* ou *θ-subsomption sous Identité d'Objet*.

La θ-subsomption sous Identité d'Objet est fondée sur le principe que, dans certains problèmes, il est utile de différencier les termes d'une clause par leur symbole. Cette contrainte, non prise en compte dans la θ-subsomption originale, lui a donc été ajoutée afin de constituer la θ-subsomption sous Identité d'Objet. Il s'agit d'une restriction de la θ-subsomption de Plotkin demeurant elle-aussi un quasi-ordre. Elle interdit que des variables de noms différents soient substituées par le même terme lors d'un test de θ-subsomption. L'OI-subsomption est utilisée dans plusieurs travaux en ILP [Esposito 1996, Semeraro 1998, Ferilli 2002] ainsi que dans plusieurs systèmes d'apprentissage comme TRITOP [Geibel 1997] ou FOCL-OI [Semeraro 1994, Esposito 1994].

Nous donnons maintenant la définition de θ-subsomption sous Identité d'Objet pour deux clauses de Horn.

Définition 60 (OI-subsomption et OI-équivalence)**.** *Soit C et D deux clauses de Horn, la clause C θ_{OI}-subsume D, notée $C \succeq_{OI} D$, si et seulement s'il existe une substitution θ telle que $C\theta \subseteq D$ où θ est injectif. Deux clauses C et D sont* OI-équivalentes *ou* équivalentes sous OI-subsomption*, notées $C \sim_{OI} D$, si et seulement si $C \succeq_{OI} D$ et $D \succeq_{OI} C$.*

Illustrons maintenant cette définition.

Exemple 23. *Soit les clauses suivantes :*

$$\begin{aligned} C &= p(X, X). \\ D &= p(X, Y). \\ E &= p(a, a). \end{aligned}$$

La clause C θ_{OI}-subsume E avec $\theta = \{X/a\}$ tandis que la clause D ne θ_{OI}-subsume pas E, un terme de la clause à subsumer ne peut être associé, dans la substitution, à au plus une variable de la clause subsumant.

Sous θ-subsomption de Plotkin, la clause C θ-subsume E avec $\theta = \{X/a\}$ et la clause D θ-subsume également E avec $\theta = \{X/a,\ Y/a\}$.

L'exemple précédent démontre que la θ-subsomption sous Identité d'Objet est une restriction de la θ-subsomption de Plotkin. Ainsi, pour deux clauses C et D, si C θ_{OI}-subsume D alors C θ-subsume D. La réciproque n'est pas vraie, comme nous le montre cet exemple. La θ-subsomption sous Identité d'Objet n'est donc pas équivalente à l'implication logique dans le cas de clauses, ni à la θ-subsomption dans le cas de clauses non récursives.

Les restrictions imposées par la θ-subsomption sous Identité d'Objet ont cependant l'avantage de réduire la complexité d'un test d'OI-équivalence entre clauses. Ceci s'explique principalement par la contrainte d'injectivité imposée à la substitution pour une OI-subsomption. Ainsi, dans le cas de clauses Datalog, une clause C est OI-équivalente à une clause D si C et D sont des variantes alphabétiques. Ce constat permet, dans le cas de clauses Datalog, de simplifier le problème de réduction d'une clause. La complexité de la réduction d'une clause sous Identité d'Objet est alors quadratique dans le nombre de littéraux de la clause [Semeraro 1996].

Une autre propriété intéressante issue de l'Identité d'Objet provient d'une observation faite sur la taille des clauses. Pour deux clauses C et D si $C \succeq_{OI} D$ alors $|C| \leqslant |D|$, la réciproque n'est pas vraie. Cette propriété limite l'usage d'un test de OI-subsomption et réduit la complexité de cette subsomption. En effet, cette dernière dépend directement du nombre de variables à substituer.

Malgré ces points positifs, la θ-subsomption sous Identité d'Objet n'en demeure pas moins un problème pouvant se révéler difficile. En effet, l'OI-subsomption est équivalente à un problème d'isomorphisme de sous-graphes [Malerba 2001, Nijssen 2003, Yan 2003, Kuznetsov 2005, Douar 2012, C Garriga 2012]. Elle constitue donc également un problème NP-complet.

Tout comme la θ-subsomption, l'OI-subsomption peut être une bonne alternative à l'implication logique là où l'Identité d'Objet a du sens. Elle a notamment été étudiée et testée avec succès sur des tâches concrètes dans de nombreux travaux [Semeraro 1994, Ferilli 2002, Nijssen 2003, C Garriga 2012]. Cependant, bien que le test d'OI-équivalence soit quadratique et qu'il soit possible de réduire le nombre de tests de subsomption grâce la taille des clauses manipulées, l'OI-subsomption est, comme la θ-subsomption, un problème NP-complet. Nous continuons donc à rechercher d'autres relations de subsomption bénéficiant d'une complexité plus attrayante.

4.1.4 Subsomption stochastique

La subsomption stochastique, autrement appelée *stochastic matching* ou *sampling of substitutions*, est une technique de subsomption utilisée notamment dans [Sebag 1997b, Sebag 1997a]. La subsomption stochastique est motivée par le constat que la θ-subsomption entre deux clauses C et D est un processus combinatoire pouvant se révéler coûteux (complexité en temps $\mathcal{O}(|C|^{|D|})$) dans le cas où les littéraux de ces clauses disposent d'un même symbole de prédicat. Cela s'explique par le fait qu'un test de θ-subsomption consiste à tester tous les appariements possibles entre les littéraux des deux clauses, ceci dans le but de vérifier l'existence d'une possible substitution. Afin d'éviter ce problème, un test de subsomption stochastique consiste à ne considérer qu'un échantillon des appariements possibles entre les littéraux des deux clauses. Plus précisément, seules k substitutions possibles sont

considérées. Les littéraux de la clause D utilisés pour construire ces différentes substitutions sont choisis avec une probabilité uniforme.

Définition 61 (Subsomption stochastique)**.** *Soit une constante $k \in \mathbb{N}$, une clause C k-subsume stochastiquement une clause D si parmi les k substitutions tirées aléatoirement, il existe une substitution θ_i telle que $C\theta_i \subseteq D$ avec $1 \leqslant i \leqslant k$.*

Cette subsomption stochastique a une complexité en $\mathcal{O}(k \times |C|)$. Elle est incomplète puisqu'au bout du nombre d'essais autorisés, il peut demeurer une substitution permettant à la clause C de subsumer la clause D. Dans le cas où la constante $k = \infty$, la subsomption stochastique devient cependant équivalente à la θ-subsumption. Cette approche se révèle intéressante dans le cas où les littéraux des clauses envisagées utilisent un grand nombre de fois un même symbole de prédicat rendant la θ-subsomption plus difficile. Ainsi, la subsomption stochastique a notamment été utilisée dans [Sebag 1997b] pour le problème de mutagenèse [King 1995] consistant en la détection de molécules cancérigènes. Cette subsomption propose donc une complexité intéressante et exploitable dans des problèmes réels. Cependant avant de l'utiliser, il convient de considérer si cette approche probabiliste est envisageable vis-à-vis d'un problème de classification de documents XML mais également s'il n'existe pas une subsomption complète dont le coût serait similaire.

4.1.5 Arc Consistency

Les relations de subsomption vues précédemment portent sur les clauses et souffrent, pour la plupart, de complexités élevées. D'autres approches, utilisant des représentations différentes, ont été proposées pour ce problème de subsomption. On retrouve notamment l'Arc Consistency [Liquiere 2007, Douar 2011, Douar 2012] basée sur la projection entre graphes. Une projection d'un graphe G_1 vers G_2 est un mapping de chaque nœud du graphe G_1 vers un ou plusieurs nœuds de G_2. Il constitue ainsi une relation de généralité pour les graphes. Cette notion englobe, entre autre, celle d'homomorphisme de graphes et d'isomorphisme de sous-graphes. Un parallèle peut donc être fait avec les notions de θ-subsomption et d'OI-subsomption.

Présentons ici l'AC-projection pour des digraphes définis dans [Liquiere 2007]. Pour un digraphe G, nous notons N_G l'ensemble des nœuds de G, E_G l'ensemble des arcs de G, Σ_G l'ensemble des labels de G et $lab : N_G \to \Sigma_G$ une fonction qui associe à un nœud de G un label de G.

Définition 62 (Labelling)**.** *Un* labeling *d'un digraphe G_1 dans un digraphe G_2 est une fonction $I : N_{G_1} \to 2^{N_{G_2}}$ (où $2^{N_{G_2}}$ est l'ensemble des parties de N_{G_2}) telle que $\forall x \in N_{G_1},\ \forall y \in I(x),\ lab(x) = lab(y)$.*

Définition 63 (Cohérence d'un arc)**.** *Soit deux digraphes G_1 et G_2, le labeling $I : N_{G_1} \to 2^{N_{G_2}}$ est* cohérent avec l'arc $(x, y) \in E_{G_1}$ *si et seulement si :*

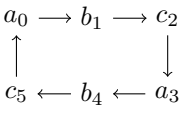
$$\left.\begin{array}{ll} 1. & \forall x_1 \in I(x)\ \exists y_1 \in I(y) \text{ tel que } (x_1, y_1) \in E_{G_2}, \\ 2. & \forall y_2 \in I(y)\ \exists x_2 \in I(x) \text{ tel que } (x_2, y_2) \in E_{G_2}. \end{array}\right\} \quad \begin{array}{l} \text{on note alors :} \\ I(x) \sim\!> I(y) \end{array}$$

La cohérence d'un arc assure, pour un labeling donné de deux nœuds x et y liés par un arc dans un graphe de départ, l'existence, dans le graphe d'arrivée, d'une image de cet arc telle que les nœuds de cette image soient étiquétés par le labelling. La notion d'AC-Projection généralise cette définition à l'ensemble des arcs d'un digraphe. Elle définit un quasi-ordre entre les digraphes et correspond ainsi à une relation de subsomption pour ces derniers.

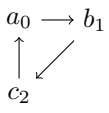
Définition 64 (AC-Projection et AC-equivalence)**.** *Un labeling I d'un digraphe G_1 vers un digraphe G_2 est une* AC-Projection*, notée $G_1 \rightharpoonup G_2$, si et seulement si I est cohérent pour tout $e \in E_{G_1}$. Deux digraphes G_1 et G_2 sont* AC-équivalents*, notés $G_1 \rightleftharpoons G_2$, si et seulement si $G_1 \rightharpoonup G_2$ et $G_2 \rightharpoonup G_1$.*

Illustrons ces propos à l'aide de l'exemple suivant.

Exemple 24. *Soit deux digraphes G_1 et G_2 :*



G_1



G_2

avec :

$$\begin{array}{lcl} N_{G_1} & = & \{0,1,2,3,4,5\} \\ E_{G_1} & = & \{(0,1),(1,2),(2,3), \\ & & (3,4),(4,5),(5,0)\} \\ \Sigma_{G_1} & = & \{a,b,c\} \end{array}$$

$$\begin{array}{llllll} lab(0) & = & lab(3) & = & a \\ lab(1) & = & lab(4) & = & b \\ lab(2) & = & lab(5) & = & c \end{array}$$

avec :

$$\begin{array}{lcl} N_{G_2} & = & \{0,1,2\} \\ E_{G_2} & = & \{(0,1),(1,2),(2,0)\} \\ \Sigma_{G_2} & = & \{a,b,c\} \end{array}$$

$$\begin{array}{lcl} lab(0) & = & a \\ lab(1) & = & b \\ lab(2) & = & c \end{array}$$

On peut construire le labeling I de G_1 vers G_2 tel que :

$$I(a_0) = \{a_0\},\ I(b_1) = \{b_1\},\ I(c_2) = \{c_2\},\ I(a_3) = \{a_0\},\ I(b_4) = \{b_1\},\ I(c_5) = \{c_2\}$$

On a ainsi :

$$I(a_0) \sim> I(b_1),\ I(b_1) \sim> I(c_2),\ I(a_3) \sim> I(b_4),\ I(b_4) \sim> I(c_5),\ I(c_5) \sim> I(a_0)$$

On en déduit que : $G_1 \rightharpoonup G_2$. De la même manière, on peut constuire le labeling J de G_2 vers G_1 tel que :

$$J(a_0) = \{a_0, a_3\},\ J(b_1) = \{b_1, b_4\},\ J(c_2) = \{c_2, c_5\}$$

et constater que :

$$J(a_0) \sim> J(b_1),\ J(b_1) \sim> J(c_2) \text{ et } J(c_2) \sim> J(a_0)$$

On a donc également : $G_2 \rightharpoonup G_1$. De ce fait, G_1 et G_2 sont équivalents vis-à-vis de l'AC-équivalence.

Tout comme Plotkin l'a fait pour les littéraux d'une clause sous θ-subsomption, la redondance et l'équivalence sous AC-Projection de nœuds dans un digraphe ont été établies. Elles permettent la définition de digraphe réduit sous AC-Projection.

Définition 65 (Nœud AC-redondant et Nœuds AC-équivalents)**.** *Un nœud $x \in N_{G_1}$ d'un digraphe G_1 est* AC-redondant *s'il existe un digraphe G_2 tel que :*

1. $N_{G_2} = N_{G_1} \setminus \{x\}$,
2. $E_{G_2} = E_{G_1} \setminus \{(y,z) \mid y = x \text{ ou } z = x\}$,
3. $G_1 \rightleftharpoons G_2$.

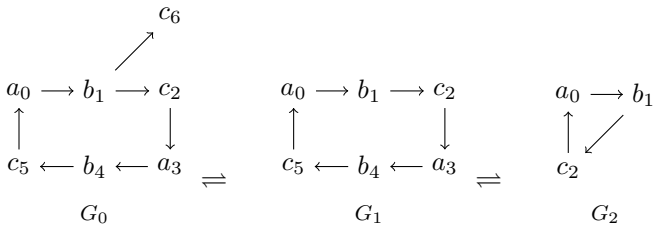
Deux nœuds $x, y \in N_G$ d'un digraphe G sont AC-équivalents *si et seulement s'il existe une AC-Projection $I : G \to G$ telle que $I(x_1) = I(x_2)$.*

Le digraphe réduit sous AC-Projection d'un digraphe G, noté $\mathcal{R}(G)$, est obtenu par fusion des nœuds AC-équivalents puis suppression des nœuds AC-redondants de G.

Définition 66 (Digraphe réduit sous AC-Projection)**.** *Soit un digraphe G, $\mathcal{R}(G)$ est le* digraphe réduit *de G si $\mathcal{R}(G)$ ne contient aucun nœud AC-redondant ni AC-équivalent et $\mathcal{R}(G) \rightharpoonup G$.*

Le détail de l'algorithme permettant la construction d'un digraphe réduit sous AC-Projection à partir d'un digraphe se trouve dans [Liquiere 2007].

Exemple 25. *Soit trois digraphes G_0, G_1 et G_2 :*



Le digraphe G_1 est obtenu par suppression des nœuds AC-redondants de G_0. G_2 est obtenu par fusion des nœuds AC-équivalents de G_1. On a donc $G_2 = \mathcal{R}(G_0)$.

Contrairement à la θ-subsomption, l'AC-Projection n'est pas équivalente à la notion d'homomorphisme entre graphes [Douar 2011]. Le lien existant entre ces deux notions est précisé par le lemme suivant, la réciproque de ce dernier n'est pas vraie :

Lemme 4. *Soit deux digraphes G_1 et G_2, s'il existe un homomorphisme de G_1 vers G_2, alors il existe un labeling I étant une AC-Projection de G_1 vers G_2.*

Tester s'il existe une AC-projection d'un graphe vers un autre peut-être réalisé en temps polynomial. Plus précisément, l'algorithme d'AC-Projection a une complexité en temps de $\mathcal{O}(ed^2)$ où e est le nombre d'arcs et d est la taille du plus grand sous-ensemble de nœuds ayant le même label. Une version détaillée de cet algorithme se trouve dans [Liquiere 2007]. L'AC-Projection bénéficie donc d'une complexité plus attrayante que la θ-subsomption. Elle dispose également d'une forme réduite et offre ainsi un compromis entre la complexité de la subsomption et les représentations qu'elle autorise. Cependant, désirant rester équivalent à l'implication logique à la base de la définition de notre problème, nous penchons davantage pour la θ-subsomption. Toutefois, nous garderons l'AC-Projection comme élément de comparaison, la complexité de sa subsomption étant du même ordre que celle dont nous désirons bénéficier pour notre problème.

4.2 Opérateur de généralisation

Dans cette section, nous nous intéressons à la notion de quasi-ordre et à l'obtention d'opérateurs de généralisation. Nous avons décrit précédemment plusieurs relations de subsomption qui sont, à nos yeux, les plus connues et les plus utilisées. Ces relations de subsomption sont des quasi-ordres qui disposent de forme réduite pour chaque classe d'équivalence d'hypothèses. L'existence d'un quasi-ordre permet de bénéficier des notions de borne supérieure et inférieure également appelées *spécialisation* et *généralisation*.

Définition 67 (Généralisation, Généralisation minimale et Généralisation la plus spécifique)**.** *Soit une relation $\geq$ étant un quasi-ordre (i.e. une relation transitive et réflexive) sur un ensemble S et $S' \subseteq S$ un sous-ensemble de S.*

1. *Un élément $x \in S$ est une* généralisation *de S' sous $\geq$ si pour tout élément $y \in S'$, on a $x \geq y$.*
2. *Une généralisation $x \in S$ de S' est une* généralisation la plus spécifique *(également appelée moindre généralisé, least general generalization ou encore lgg) de S' sous $\geq$ si pour toute généralisation $y \in S$ de S' sous $\geq$, on a $y \geq x$.*
3. *Une généralisation $x \in S$ de S' est une* généralisation minimale *de S' si pour toute généralisation $y \in S$ de S' tel que $x \geq y$, on a $x \sim y$, i.e. $x \geq y$ et $y \geq x$.*

En opposition à cela, existent les notions de spécialisation, de spécialisation maximale et de spécialisation la plus générale.

Définition 68 (Spécialisation, Spécialisation maximale et Spécialisation la plus générale)**.** *Soit une relation $\geq$ étant un quasi-ordre sur un ensemble S et $S' \subseteq S$ un sous-ensemble de S.*

1. *Un élément $x \in S$ est une* spécialisation *de S' sous $\geq$ si pour tout élément $y \in S'$, on a $y \geq x$.*
2. *Une spécialisation $x \in S$ de S' est une* spécialisation la plus générale *de S' sous $\geq$ si pour toute spécialisation $y \in S$ de S' sous $\geq$, on a $x \geq y$.*
3. *Une spécialisation $x \in S$ de S' est une* spécialisation maximale *de S' sous $\geq$ si pour toute spécialisation $y \in S$ de S' tel que $y \geq x$, on a $x \sim y$.*

Ces définitions n'imposent ni l'existence, ni l'unicité d'un moindre généralisé ou d'une spécialisation maximale. Leurs présences pour chaque couple d'éléments d'un ensemble permettent de définir la notion de *treillis*.

Définition 69 (Treillis)**.** *Soit une relation $\geq$ étant un quasi-ordre sur un ensemble S, si pour tous éléments $x, y \in S$ il existe un moindre généralisé de x et y ainsi qu'une spécialisation maximale alors S est un* treillis *selon $\geq$.*

On peut remarquer que si deux éléments $x, y \in S$ ont plus d'un moindre généralisé, ces derniers sont équivalents sous le quasi-ordre utilisé. Ainsi, si un moindre généralisé existe, alors il est unique modulo l'équivalence de classes. De la même manière, si un moindre généralisé existe, alors il est également une généralisation minimale. En effet, contrairement au moindre généralisé, il peut exister plusieurs généralisations minimales non-équivalentes vis-à-vis d'un quasi-ordre. Il arrive parfois qu'en PLI la définition de moindre généralisé fasse référence en réalité à une généralisation minimale. Dans ce cas, l'unicité ou non de ce dernier, modulo équivalence du quasi ordre, est précisée comme dans [Semeraro 1998, Esposito 2000, Decoster 2010].

Notre intérêt pour la généralisation, et plus particulièrement le moindre généralisé, s'explique par le fait que la résolution de notre problème en PLI passe par la recherche d'un concept couvrant un ensemble d'exemples formé d'instances du concept. Comme le moindre généralisé d'un ensemble d'exemples est une généralisation couvrant les autres généralisations de cet ensemble, nous sommes sûrs que ce dernier contient toute l'information de l'ensemble des exemples à généraliser. Le moindre généralisé semble donc intéressant pour trouver une représentation d'un concept à apprendre. Nous allons donc, par la suite, préciser la notion de moindre généralisé pour les différentes relations de subsomption détaillées précédemment et présenter leurs avantages ainsi que leurs inconvénients.

4.2.1 Généralisation sous implication

L'implication logique est un quasi-ordre pour les clauses, nous pouvons donc définir un moindre généralisé sous implication logique en instanciant la définition 67 telle que le

quasi-ordre $\geq$ est remplacé par l'implication logique $\models$ et l'ensemble S par un ensemble de clauses. Le moindre généralisé sous Implication est alors appelé *LGGI* (pour *least general generalization under implication*). Donnons maintenant un exemple de moindre généralisé sous implication.

Exemple 26. *Soit les clauses suivantes :*

$$\begin{array}{lcl} C & = & p(f(a)) \leftarrow p(a). \\ D & = & p(f^2(b)) \leftarrow p(b). \\ E & = & p(f(X)) \leftarrow p(X). \end{array}$$

La clause E est un moindre généralisé sous implication de C et de D car $E \models C$ et $D \models C$.

L'existence d'un moindre généralisé sous implication n'est pas toujours assurée comme nous le montre l'exemple 27 emprunté à [Muggleton 1994a].

Exemple 27. *Soit les clauses de Horn suivantes :*

$$\begin{array}{lcl} C & = & p(f^2(X)) \leftarrow p(X). \\ D & = & p(f^3(X)) \leftarrow p(X). \\ E & = & p(f(X)) \leftarrow p(X). \\ F & = & p(f^2(X)) \leftarrow p(X). \end{array}$$

On a $E \models C$ et $E \models D$ ainsi que $F \models C$ et $F \models D$. Or, on ne peut pas trouver de clause plus spécifique que E et F puisqu'aucun résolvant de G avec lui-même n'impliquera la clause D et aucune clause moins générale que G n'impliquera également C et D. Le même problème peut être observé pour H. Nous avons donc deux clauses impliquant C et D mais ne s'impliquant pas l'une l'autre. Il n'y a donc pas de LGGI pour C et D.

L'existence d'un moindre généralisé pour les clauses de Horn ayant une réponse négative, de nombreux travaux ont eu pour but de trouver des sous-classes de clauses dont l'existence d'un moindre généralisé sous implication est vérifiée. Nous avons ainsi le résultat suivant :

Théorème 11. *Soit une théorie du domaine T de clauses sans fonction (les termes sont uniquement des variables ou des constantes) d'un langage de clauses $\mathcal{L}$, il existe un LGGI de T dans $\mathcal{L}$.*

Un algorithme permettant de calculer le LGGI d'un ensemble de clauses sans fonction se trouve dans [Nienhuys-Cheng 1996b]. Cet algorithme est néanmoins intractable, ce dernier utilise l'implication logique et le calcul de moindre généralisé sous θ-subsomption sur des ensembles dont la taille est en $\mathcal{O}(2^{2^{2^{|A|}}})$ où A est l'ensemble de tous les atomes distincts construits à partir des constantes et symboles de prédicat de T.

Nous remplaçons maintenant le quasi-ordre de la définition 67 par l'implication logique relative à une théorie et obtenons un *moindre généralisé sous implication relative à une théorie* noté RLGGI. Le RLGGI est lié au LGGI par le lien qui existe entre l'implication logique et l'implication logique relative. Il n'existe ainsi pas de RLGGI pour une théorie donnée dans le cas des clauses de Horn. Cependant, une réponse positive est donnée à l'existence du RLGGI pour une classe particulière de clauses sans fonction comme nous l'indique le théorème suivant :

Théorème 12. *Soit une théorie du domaine T formée de clauses fondées et sans fonction d'un langage de clauses $\mathcal{L}$, chaque sous-ensemble fini S de $\mathcal{L}$ contenant au moins une clause D, telle que $D \cup \neg T$ est non tautologique et sans fonction, a un RLGGI dans $\mathcal{L}$.*

Des extensions de ces définitions sont données dans [Boytcheva 2000, Boytcheva 2002] ainsi qu'un algorithme permettant de calculer le *RLGGI* dans le cas de clauses définies sans fonction.

La complexité du calcul d'un LGGI et d'un RLGGI rend ces opérateurs difficilement utilisables en pratique. Notre intérêt pour ces derniers n'est donc que théorique, comme cela l'était déjà pour l'implication logique et l'implication logique relative. Ce constat est renforcé par le résultat de non-existence d'un moindre généralisé pour les clauses de Horn. Ainsi, nous continuons l'étude de la notion de moindre généralisé et passons aux autres relations de subsomption précédemment décrites.

4.2.2 Généralisation sous θ-subsomption

Le moindre généralisé sous θ-subsomption, appelé least general generalization (lgg), a été introduit par Plotkin [Plotkin 1970, Plotkin 1971a, Plotkin 1971b]. Il est obtenu en instanciant la définition 67 telle que le quasi-ordre $\geq$ soit remplacé par la θ-subsomption et l'ensemble S par un ensemble de clauses. Son existence a été prouvée par Plotkin pour tout ensemble de clauses. Nous illustrons nos propos avec l'exemple suivant :

Exemple 28. *Considérons les clauses suivantes :*

$$\begin{array}{lcll} C & = & c(a) & \leftarrow \quad fc(a, X_2), fc(X_2, b). \\ D & = & c(X_1) & \leftarrow \quad fc(X_1, b), fc(b, X_3). \\ E & = & c(Z_1) & \leftarrow \quad fc(Z_1, Z_2), fc(Z_2, Z_3), fc(Z_4, b). \\ F & = & c(Z_1) & \leftarrow \quad fc(Z_1, Z_2), fc(Z_2, Z_3), fc(Z_4, b), fc(Z_5, Z_6), fc(Z_6, Z_7). \end{array}$$

Les clauses E et F sont toutes les deux des moindres généralisés des clauses C et D, la clause E est obtenue suite à la réduction de la clause F.

Le moindre généralisé de clauses sous θ-subsomption n'est pas unique. L'exemple 28 le montre. Cependant, Plotkin [Plotkin 1970] a démontré, modulo θ-équivalence, l'unicité de ce moindre généralisé pour les clauses et donc par la même occasion les clauses de Horn. Il a également prouvé que l'ensemble des clauses de Horn est clos par moindre généralisation sous θ-subsomption. Le calcul du moindre généralisé de clauses peut-être réalisé par les fonctions lgg_atom et lgg présentes dans l'algorithme 4. Cet algorithme n'est pas de complexité exponentielle mais peut donner comme résultat une clause non réduite. Dans ce cas, il est préférable de nettoyer la clause de ses littéraux redondants. Cela limite la taille des clauses manipulées, réduit le temps nécessaire aux calculs de θ-subsomption et permet une réelle unicité du moindre généralisé.

Exemple 29. *Considérons les clauses C et D définies comme suit :*

$$\begin{array}{lcl} C & = & fc(n_1, n_2), ns(n_2, n_3), fc(n_4, n_5). \\ D & = & fc(m_1, m_2), ns(m_2, m_3). \end{array}$$

Le calcul de leur moindre généralisé donne la clause suivante :

$$G = fc(X_1, X_2), ns(X_2, X_3), fc(X_4, X_5).$$

avec $\Theta = \{n_1/m_1/X_1, n_2/m_2/X_2, n_3/m_3/X_3, n_4/m_1/X_4, n_5/m_2/X_5\}$. La clause obtenue G n'est pas réduite car le littéral $fc(X_4, X_5)$ est redondant. La suppression de ce littéral donne la clause :

$$H = fc(X_1, X_2), ns(X_2, X_3)$$

qui est équivalente à G puisque $G \succeq_\theta H$ avec $\theta = \{X_1/X_4, X_2/X_5\}$. Après réduction, c'est donc H qui est fournie.

Algorithme 4 Moindre généralisé.

function lgg_atom$(\ell, \ell', \textbf{var } \Theta)$
1: **let** $\ell = R(t_1, \ldots, t_n)$;
2: **let** $\ell' = P(s_1, \ldots, s_m)$;
3: **if** $R \neq P$ **or** $n \neq m$ **then**
4: **return** $\varnothing$;
5: **for** $i \in \{1, \ldots, n\}$ **do**
6: $t_i^\star = \mathsf{lgg\_term}(t_i, s_i, \Theta)$;
7: **end for**
8: **return** $\{R(t_1^\star, \ldots, t_n^\star)\}$;
end function

function lgg(C, D)
9: $G := \varnothing$;
10: $\Theta := \varnothing$;
11: **for** $\ell \in C$ **do**
12: **for** $\ell' \in D$ **do**
13: $G := G \cup \mathsf{lgg\_atom}(\ell, \ell', \Theta)$;
14: **end for**
15: **end for**
16: **return** G ;
end function

function lgg_term$(t, s, \textbf{var } \Theta)$
17: **if** $t = s$ **then**
18: **return** t ;
19: **if** $(t = X$ **or** $s = Y)$
and $\exists t/s/t^\star \in \Theta$ **then**
20: **return** $t^\star$;
21: **if** $t = R(t_1, \ldots, t_n)$
and $s = R(s_1, \ldots, s_n)$ **then**
22: **for** $i \in \{1, \ldots, n\}$ **do**
23: $t_i^\star = \mathsf{lgg\_term}(t_i, s_i, \Theta)$;
24: **end for**
25: **return** $R(t_1^\star, \ldots, t_n^\star)$;
26: $t^\star :=$ **new** variable ;
27: $\Theta := \Theta \cup \{t/s/t^\star\}$;
28: **return** $t^\star$;
end function

Dans le cas des algorithmes de [Plotkin 1970] présentés, la complexité en temps de la subsomption est en $|D|^{\mathcal{O}(|C|)}$ (exponentielle) et celle du calcul de moindre généralisé en $\mathcal{O}(|C| \times |D|)$ (quadratique). Le calcul d'un moindre généralisé réduit a, quant-à-lui, une complexité en temps plus importante. En effet, le calcul d'un moindre généralisé peut retourner une clause non réduite et la réduction d'une clause est un problème NP-complet.

Afin de prendre en compte une théorie du domaine, Plotkin [Plotkin 1971b, Plotkin 1971a] a introduit la notion de moindre généralisé relatif à une théorie sous θ-subsomption de deux clauses. La généralisation relative à une théorie du domaine permet de prendre en considération une base de connaissance au travers du quasi-ordre utilisé qu'est la θ-subsomption relative à une théorie. Son existence dans le cas des clauses et des clauses de Horn a été étudiée par Nibblet. Il a prouvé dans [Niblett 1988] qu'il n'existe pas de moindre généralisé relalif ni pour les clauses de Horn, ni pour les clauses. Un exemple illustrant la non-existence de ces moindres généralisés se trouve dans [Nienhuys-Cheng 1997b] (page 185). Des sous-classes de clauses assurant l'existence d'un moindre généralisé sous θ-subsomption relative ont alors été recherchées. Cette recherche a abouti, en autre, au théorème suivant :

Théorème 13. *Soit T un ensemble fini d'atomes fondés d'un langage $\mathcal{L}$ de clauses ou de clauses de Horn, chaque ensemble non vide de $\mathcal{L}$ a un lgg relatif à T qui appartient à $\mathcal{L}$.*

Ce théorème est la base du système d'inférence GOLEM [Muggleton 1990]. Ce système repose sur le calcul d'un moindre généralisé relatif à un ensemble d'atomes fondés T à partir de deux faits fondés e_1 et e_2. Le moindre généralisé relatif à un ensemble d'atomes fondés $T = (a_1 \wedge \cdots \wedge a_n)$ de deux atomes fondés e_1 et e_2, noté $\mathsf{rlgg}_T(e_1, e_2)$, est calculé de la manière suivante :

$$\mathsf{rlgg}_T(e_1, e_2) = \mathsf{lgg}(e_1 \leftarrow a_1 \wedge \cdots \wedge a_n, e_2 \leftarrow a_1 \wedge \cdots \wedge a_n)$$

où lgg est le moindre généralisé, défini par Plotkin, de deux clauses. Rappelons qu'un ensemble d'atomes fondés peut-être obtenu en considérant un h-easy modèle de Herbrand d'un ensemble de clauses définies (voir définition 37). Cela permet de construire un moindre généralisé relatif à un ensemble de clauses définies.

Voici un exemple de la construction d'un moindre généralisé relatif :

Exemple 30. *Soit une théorie du domaine T contenant les clauses suivantes :*

$$a_1 = oiseau(titi) \leftarrow$$
$$a_2 = oiseau(tweety) \leftarrow$$

et un ensemble d'exemples :

$$e_1 = voler(titi) \leftarrow$$
$$e_2 = voler(tweety) \leftarrow$$

On peut ensuite construire les clauses suivantes :

$$C_1 = voler(titi) \leftarrow oiseau(titi), oiseau(tweety)$$
$$C_2 = voler(tweety) \leftarrow oiseau(tweety), oiseau(titi)$$

On obtient alors le moindre généralisé suivant :

$$\mathsf{rlgg}_T(C_1, C_2) = voler(X) \leftarrow oiseau(X), oiseau(tweety), oiseau(titi)$$

Après réduction, le moindre généralisé relatif à T est :

$$\mathsf{rlgg}_T(C_1, C_2) = voler(X) \leftarrow oiseau(X)$$

On a ainsi $T \wedge C \vdash e_1 \wedge e_2$.

Malgré les restrictions apportées, le moindre généralisé sous θ-subsomption relatif souffre de quelques problèmes. Le premier problème, exposé dans [Buntine 1988], provient de l'existence de moindres généralisés dont la taille croît exponentiellement avec le nombre d'exemples. Le calcul d'un moindre généralisé devient alors intractable dans certains cas. Le second problème, présenté dans [Plotkin 1971b, Plotkin 1971a], provient de l'existence de moindre généralisé de taille infinie dans les cas où la théorie n'est pas limitée à un ensemble d'atomes fondés. Enfin, comme pour le moindre généralisé sous θ-subsomption, le moindre généralisé sous θ-subsomption relative peut construire des clauses non réduites. La réduction de ces clauses se révèle coûteuse du fait de la complexité de la θ-subsomption relative.

Le moindre généralisé relatif ou non à une théorie sous θ-subsomption présente plusieurs avantages comme la preuve de son existence pour tout ensemble de clauses ou bien l'existence d'algorithmes. Cependant, la complexité de ces algorithmes peut rendre difficile leur utilisation, il convient donc de définir un cadre plus restrictif avant de les utiliser. Nous avons précédemment vu une restriction de la θ-subsomption nommée l'OI-subsomption. Nous nous y intéressons donc.

4.2.3 Généralisation sous OI-subsomption

Une définition du moindre généralisé sous OI-subsomption est obtenable en instanciant la définition 67 avec, pour quasi-ordre, l'OI-subsomption. Cette définition diffère légèrement de celles proposées dans [Semeraro 1994, Geibel 1997]. En effet, dans ces dernières, une généralisation d'un ensemble de clauses est appelée moindre généralisé s'il n'existe pas

de généralisation plus spécifique. Ainsi, il peut exister deux moindres généralisés incomparables. Un amalgame est ici fait entre les notions de moindre généralisé et de généralisation minimale précédemment définies. Contrairement à la θ-subsomption, l'unicité du moindre généralisé respectant la θ-subsomption sous Identité d'Objet n'est plus assurée [Semeraro 1994, Haussler 1989]. Ainsi, les moindres généralisés manipulés correspondent en réalité à des généralisations minimales non équivalentes sous Identité d'Objet. L'espace de recherche n'est donc plus un treillis contrairement à la θ-subsomption.

Exemple 31. *Soit deux clauses :*

$$C = h \leftarrow cercle(a), bleu(a), petit(a).$$
$$D = h \leftarrow cercle(b), bleu(b), large(a), cercle(c), vert(c), petit(c).$$

Le moindre généralisé sous θ-subsomption de C et D est :

$$E = h \leftarrow cercle(X), bleu(X), cercle(Y), petit(Y).$$

tandis que les moindres généralisés sous Identité d'Objet sont :

$$F = h \leftarrow cercle(X), bleu(X).$$
$$G = h \leftarrow cercle(X), petit(X).$$

Le calcul des moindres généralisés sous Identité d'Objet se fait en plusieurs étapes. Nous présentons, pour plus de simplicité, la construction d'un moindre généralisé sous OI-subsomption de clauses sans terme complexe ni constante. La première étape de la construction consiste à calculer le moindre généralisé défini par Plotkin. Ce moindre généralisé ne respecte pas l'Identité d'Objet. Cependant, il a été prouvé que l'ensemble des généralisations sous Identité d'Objet d'une conjonction de littéraux, dont les termes sont des variables, est un sous-ensemble de l'ensemble des parties (à un renommage de variables près) de cette conjonction [Esposito 1996]. Il en est de même dans le cas des clauses de Horn [Esposito 2004]. Seules les généralisations dont le mapping Θ est injectif respectent la contrainte d'Identité d'Objet. Parmi elles se trouvent des généralisations minimales. Il est alors nécessaire de supprimer les généralisations équivalentes ou trop générales vis-à-vis de l'Identité d'Objet. Les généralisations restantes sont alors les moindres généralisés sous Identité d'Objet des clauses de départ. Un algorithme détaillé de ce processus se trouve dans [Rodrigues 2013]. Une adaptation de ce dernier aux clauses connectées est présentée dans [Esposito 2004]. Cette restriction aux clauses connectées permet de réduire la taille de l'espace de recherche et le nombre de généralisations possibles.

La taille d'un moindre généralisé sous θ-subsomption n'est pas limitée et peut être exponentielle dans le nombre de clauses généralisées. Ceci n'est cependant pas le cas avec la θ-subsomption sous Identité d'Objet. En effet, une généralisation produite à partir du LGG de deux clauses C et D sous Identité d'Objet a l'avantage de contenir un nombre de littéraux inférieur à $min(\{|C|, |D|\})$. La conséquence de cet avantage est l'augmentation de la taille de l'espace d'hypothèses du fait de l'absence d'unicité. Ce dernier contient dans le pire des cas $2^{min(\{|C|,|D|\})}$ généralisations possibles. On a donc un nombre important d'hypothèses à examiner.

La subsomption et le moindre généralisé sous Identité d'Objet ont donc des avantages appréciables, comme la limitation de la taille des hypothèses et une subsomption plus simple comparée à celle de la θ-subsomption. Ils disposent également d'inconvénients non négligeables, comme la présence de plusieurs généralisations possibles et une non-équivalence vis-à-vis de l'implication contrairement à la θ-subsomption. Il convient donc d'utiliser la subsomption et la généralisation sous Identité d'Objet dans le cadre de problèmes où cette dernière peut se révéler utile et justifiée. Une étude de notre problème devra donc être faite afin de sélectionner ou non cette relation de subsomption.

4.2.4 Généralisation sous AC-Projection

Revenons maintenant sur la notion d'AC-Projection et complétons la en définissant un opérateur de généralisation. La généralisation, dans le cadre des graphes, peut s'effectuer à l'aide d'un produit cartésien entre graphes respectant la notion d'homomorphisme. L'AC-Projection portant sur des digraphes, il convient de préciser ce produit pour les digraphes.

Définition 70 (Produit de digraphes respectant l'homomorphisme)**.** *Le* produit de deux digraphes, G_1 et G_2, respectant l'homomorphisme *est* $G_1 \otimes G_2 = G$ *où* G *est un digraphe tel que :*

- $lab_G = lab_{G_1} \cap lab_{G_2}$,
- $N_G = \{(x_1, x_2) \in N_{G_1} \times N_{G_2} \mid lab(x) = lab(x_1) = lab(x_2)\}$,
- $E_G = \{((x_1, x_2), (x'_1, x'_2)) \mid (x_1, x'_1) \in E_{G_1} \text{ et } (x_2, x'_2) \in E_{G_2}\}$

respectant les propriétés suivantes :

- $G \mapsto G_1$ *et* $G \mapsto G_2$,
- *pour tout digraphe* G'*, si* $G' \mapsto G_1$ *et* $G' \mapsto G_2$*, alors* $G' \mapsto G$.

Plus de détails sur la notion de produit de digraphes respectant l'homomorphisme se trouvent dans [Liquiere 1998]. Ce produit de digraphes est utilisé pour définir le produit entre digraphes respectant l'AC-Projection.

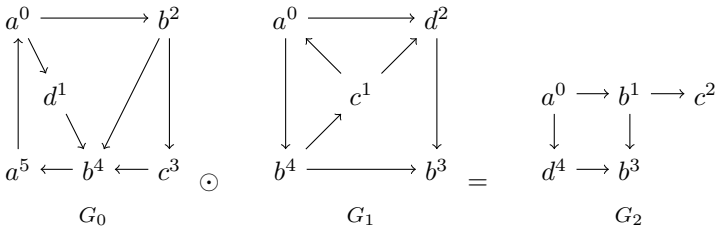
Définition 71 (Produit de digraphes respectant l'AC-Projection)**.** *Le* produit de deux digraphes, G_1 et G_2, respectant l'AC-Projection *est* $G_1 \odot G_2 = \mathcal{R}(G_1 \otimes G_2)$ *tel que :*

- $(G_1 \odot G_2) \rightharpoonup G_1$ *et* $(G_1 \odot G_2) \rightharpoonup G_2$,
- *pour tout digraphe* G'*, si* $G' \rightharpoonup G_1$ *et* $G' \rightharpoonup G_2$*, alors* $G' \rightharpoonup (G_1 \odot G_2)$

où $\mathcal{R}(G_1 \otimes G_2)$ *est le digraphe réduit sous AC-Projection de* $G_1 \otimes G_2$ *(cf. définition 66 pour la notion de digraphe réduit sous AC-Projection).*

Comme pour la θ-subsomption et contrairement à la subsomption sous Identité d'Objet, le moindre généralisé obtenu sous AC-Projection est unique. Nous illustrons cette définition avec l'exemple suivant :

Exemple 32. *Soit trois digraphes :*



On a : $G_0 \not\rightharpoonup G_1$, $G_1 \not\rightharpoonup G_0$, $G_2 \rightharpoonup G_0$ *et* $G_2 \rightharpoonup G_0$. *Le digraphe* G_2 *est obtenu par produit des digraphes* G_0 *et* G_1 *sous AC-Projection, on a donc* $G_2 = G_0 \odot G_1$.

Ces graphes peuvent être facilement représentés par des clauses de Horn à l'aide d'un prédicat binaire edge/2 représentant la notion d'arc dirigé ainsi que des prédicats unaires représentant les labels du graphe, c'est-à-dire $a/1$, $b/1, c/1$ *et* $d/1$. *Les clauses obtenues sont ainsi les suivantes :*

$$
\begin{array}{lll}
concept & \leftarrow & edge(X_0, X_1),\ edge(X_0, X_2),\ edge(X_5, X_0),\ edge(X_1, X_4),\\
 & & edge(X_4, X_5),\ edge(X_4, X_3),\ edge(X_2, X_3),\ edge(X_2, X_4),\\
 & & a(X_0),\ d(X_1),\ b(X_2),\ c(X_3),\ b(X_4),\ a(X_5).\\
concept & \leftarrow & edge(X_0, X_1),\ a(X_0),\ d(X_1),\ edge(X_1, X_2),\ b(X_2),\\
 & & edge(X_0, X_3),\ edge(X_3, X_2),\\
 & & b(X_3),\ edge(X_3, X_4),\ c(X_4),\ edge(X_4, X_0),\ edge(X_4, X_1).
\end{array}
$$

Le moindre généralisé réduit de ces clauses contient 55 arcs et est ainsi plus spécifique que le digraphe réduit obtenu à l'aide de l'AC-Projection. Un moindre généralisé obtenu par AC-Projection est donc moins spécifique que son homologue sous θ-subsomption.

Les algorithmes de sumbsomption, de réduction et de généralisation sous AC-Projection sont polynomiaux. L'AC-Projection définit donc des outils intéressants et utilisables dans le cadre d'un apprentissage. La seule ombre au tableau est que l'AC-Projection ne respecte pas l'implication logique qui est une relation de subsomption dont l'usage se justifie par sa sémantique ainsi que par la définition du problème de classification en PLI. Une des conséquences de ce non-respect est la non-équivalence du problème d'homomorphisme de graphes avec celui de l'AC-Projection. Ainsi, les moindres généralisés sous AC-Projection sont moins spécifiques que ceux existants sous θ-subsomption. Il est alors nécessaire de voir si l'utilisation d'une telle relation est pertinente pour notre problème.

4.3 Restrictions syntaxiques sur les clauses

La relation de subsomption choisie lors d'un apprentissage joue un rôle important dans la réussite de ce dernier. En fonction de celle-ci, le nombre de généralisations possibles d'un ensemble de clauses et leur taille varieront de manière conséquente. Elle influence ainsi directement l'espace de recherche en omettant de celui-ci des clauses incohérentes selon elle. L'espace de recherche est, par exemple, infini pour la θ-subsomption sur des clauses et fini, mais exponentiellement grand, pour l'AC-Projection et la OI-subsomption sur des clauses Datalog. D'autres techniques permettent de restreindre l'espace de recherche. On retrouve notamment les biais syntaxiques imposés au langage de clauses manipulées. Un biais syntaxique définit l'ensemble des hypothèses envisageables pour un problème en spécifiant explicitement leur syntaxe.

Dans cette section, nous détaillons quelques restrictions syntaxiques intéressantes pour une tâche de classification. Ces restrictions ont, pour la plupart, été étudiées avec la θ-subsomption comme relation de subsomption. Le choix de cette relation est motivé par l'équivalence établie entre la θ-subsomption et l'implication logique pour des clauses non récursives. Rappelons que le test de θ-subsomption entre clauses de Horn est un problème NP-complet. Ainsi, une partie des restrictions syntaxiques présentées ont été définies dans le but d'outrepasser ce problème.

4.3.1 Clause déterminée

Une première approche, suggérée dans [Quinlan 1991] puis reprise dans d'autres travaux [Muggleton 1990, Kietz 1994a], est la notion de clauses déterminées. L'idée de cette dernière est assez simple. Nous savons que le problème combinatoire de la θ-subsomption provient de la recherche d'une assignation des littéraux de la première clause vers ceux de la seconde. Plus précisément, lorsque toutes les assignations possibles d'un littéral ont échoué, il est nécessaire de revenir sur le littéral précédent afin de trouver une nouvelle assignation. Ce processus de retour en arrière, également appelé *backtrack* peut, dans le pire des cas, entraîner l'énumération d'un nombre exponentiel d'assignations distinctes des littéraux de

la première clause vers ceux de la seconde. Le principe de déterminisme d'une clause consiste alors à limiter ce nombre d'assignations en imposant pour chaque littéral l'existence d'au plus un appariement possible. Ce dernier peut dépendre des appariements précédents. C'est sur cette idée que repose la θ-subsomption pour clauses déterminées.

Définition 72 (Clause déterminée)**.** *La clause $C = h \leftarrow \ell_1, \cdots, \ell_n$ est* déterminée par rapport *à $D = h' \leftarrow \ell'_1, \cdots, \ell'_m$ si et seulement si pour tout $1 \leq i \leq n$, s'il existe une substitution θ telle que $h\theta = h'$ et $\{\ell_j\}_{j<i}\theta \subseteq D$, alors il existe au plus une substitution θ' compatible avec θ telle que $\{\ell_j\}_{j\leq i}\theta\theta' \subseteq D$. Une clause C est déterminée par rapport à un langage $\mathcal{L}$ si et seulement si elle est déterminée vis-à-vis de toute clause de $\mathcal{L}$.*

Autrement dit, les littéraux des clauses sont ordonnés. On a la garantie qu'au cours de la subsomption, il n'y a pour un littéral de C qu'un *matching* possible dans les littéraux de D du fait des choix faits pour les littéraux précédents.

Algorithme 5 θ-subsomption pour clauses déterminées.

```
function sub_D(C, D, θ)
 1: while C ≠ ∅ do
 2:     ℓ := first(C) ;
 3:     for ℓ' ∈ D do
 4:         if sub_atom(ℓ, ℓ', θ) then
 5:             C := C \ {ℓ} ;
 6:             continue outer loop ;
 7:     end for
 8:     return false ;
 9: end while
10: return true ;
end function
```

Par rapport à l'algorithme général de θ-subsomption entre clauses de Horn, nous sommes donc dispensés de remettre nos choix en question. C'est ce que traduit l'algorithme 5 ($\mathsf{sub_D}$) qui précise le test de θ-subsomption dédié aux clauses déterminées. La classe des clauses déterminées permet ainsi de définir une θ-susomption quadratique dans la taille des clauses manipulées, plus précisément de complexité $\mathcal{O}(|C| \times |D|)$ en temps.

La contre-partie de ce gain en complexité se traduit par un espace de recherche réduit. Ainsi, les concepts non représentables par une clause déterminée ne sont donc pas concernés par cette classe de clauses. L'apprentissage de ces derniers est alors impossible. D'autres classes de clauses plus générales ont donc été étudiées. C'est le cas des k-locales détaillées dans la sous-section suivante.

4.3.2 Clause k-locale

Les *clauses k-locales* ont été introduites par Kietz et Lübbe dans [Kietz 1994a]. L'idée principale induite par ces clauses est la suivante : une clause est k-locale si l'instanciation de chaque variable influence les variables apparaissant dans, au plus, k autres littéraux de son corps. Afin de proposer une subsomption efficace, les clauses k-locales réutilisent la notion de clauses déterminées. Une clause k-locale se décompose alors en deux parties. La première contient des littéraux déterminés et la seconde ceux non déterminés dont l'instanciation des variables influence, au plus, k autres littéraux de cette partie. La notion de k-localité correspond donc à une restriction sur la partie non déterminée d'une clause. Nous définissons maintenant formellement la notion de k-localité pour les clauses de Horn.

Définition 73 (Clause k-locale)**.** *Soit une clause de Horn* $C = h \leftarrow B_{DET}, B_{NONDET}$ *où* h *est l'atome de tête,* B_{DET} *un ensemble de littéraux déterminés et* B_{NONDET} *un ensemble de littéraux non-déterminés.*

Un ensemble de littéraux $LOC_i \subseteq B_{NONDET}$ *est* une locale non déterminée de D *si et seulement si on a* $(vars(LOC_i) \setminus vars(\{h\} \cup B_{DET})) \cap vars(D_{NONDET} \setminus LOC_i) = \emptyset$ *et s'il n'existe pas une partie locale non déterminée* $LOC_{j \neq i}$ *de* D *telle que* $LOC_i \subset LOC_j$*.*

Une locale non déterminée LOC_i *est* une k-locale *de* D *si et seulement s'il existe une constante* k *telle que* $k \geqslant min(|vars(LOC_i) \setminus vars(\{h\} \cup B_{DET})|, |LOC_i|)$*.*

Une clause k-locale *est une une clause dont chaque locale non déterminée est* k*-locale.*

Cette définition permet de réécrire l'ensemble de littéraux non déterminés B_{NONDET} sous la forme d'une union de k-locales. On a ainsi $B_{NONDET} = LOC_1 \cup \cdots \cup LOC_n$. La k-localité est une restriction syntaxique plus faible que celle des clauses déterminées. Son test de θ-subsomption est ainsi plus coûteux que celui des clauses déterminées comme nous l'indique le théorème suivant :

Théorème 14. *Soit deux clauses de Horn* C *et* D *avec* $D = h \leftarrow B_{DET}, B_{NONDET}$ *et* $B_{NONDET} = LOC_1 \cup \cdots \cup LOC_n$ *telles que la clause* $h \leftarrow B_{DET}$ *est déterminée par rapport à* C *et pour tout* $i \in [1..n]$*,* LOC_i *est une* k*-locale de* D*. Le test de* θ*-subsumption de* D *vis-à-vis de* C *est réalisé en* $|B_{DET}| \times |B| \times |C| + |B_{NONDET}|^2 + n \times (k^k \times |C|)$ *étapes d'unification.*

Dans le cas d'une clause ne contenant qu'une k-locale, la complexité du test de subsomption est exponentielle, c'est-à-dire en $\mathcal{O}(|D|^{|D|} \times |C|)$ en temps. Il en est de même avec la réduction d'une clause k-locale. Cependant, si k est une constante, la taille du moindre généralisé de Plotkin est polynomiale [Morik 1999]. La classe des k-locales constitue donc une sous-classe de clauses de Horn capable de contenir une partie non déterminée. Elle bénéficie également d'un test de subsomption ainsi que d'un moindre généralisé fini susceptibles d'être polynomiaux dans le cas où k est une constante. Il convient cependant de limiter k à de faibles valeurs ce qui restreint la nature des clauses représentables par les k-locales. Elles sont ainsi dépendantes du paramètre k qui doit être ajusté au problème étudié, si cela est possible.

4.3.3 Clause ij-déterminée

Nous allons maintenant définir une sous-classe des clauses de Horn utilisée dans le système d'inférence Golem [Muggleton 1990] : les clauses ij-déterminées. Le ij-déterminisme est une restriction syntaxique imposée sur les termes d'une clause et basée sur les notions de profondeur (cf. définition 9) et de degré.

Définition 74 (Degré)**.** *Soit une variable* V *et une clause connectée* $C = h \leftarrow \ell_1, \cdots, \ell_n$*. Le* degré (*level*) d'une variable et d'une clause connectée *se calcule de la manière suivante :*

$$level(V) = \begin{cases} 0 & si \quad V \in vars(h) \\ 1 & sinon,\ i.e. \quad 1 + min(\{level(V') \mid \\ & \qquad V' \text{ et } V \text{ sont des variables d'un même littéral } \}) \end{cases}$$

$$level(C) = max(\{level(V) \mid V \in vars(C)\}.$$

Dans le cas d'une clause non connectée, on considère son degré comme infini. Il est en de même pour le degré des variables non connectées à la tête de la clause. Le degré et la profondeur correspondent respectivement aux paramètres i et j des clauses i, j-déterminées.

Définition 75 (Clause ij-déterminée)**.** *Un fait est une clause* $0, j$*-déterminée. Une clause* $h \leftarrow \ell_1, \cdots, \ell_m, \ell_{m+1}, \cdots, \ell_n$ *est* une clause i, j-déterminée *si et seulement si :*

— $h \leftarrow \ell_1, \cdots, \ell_m$ *est* $(i-1), j$*-déterminée,*
— *les littéraux de la séquence* $\ell_{m+1}, \cdots, \ell_n$ *sont déterminés et de degré inférieur ou égal à* j*.*

Exemple 33. *Soit deux clauses :*

$$\begin{array}{lcll} C & = & grandfather(X,Y) \leftarrow & father(X,U), father(U,V), mother(W,V), \\ & & & mother(W,Y). \\ D & = & grandfather(X,Y) \leftarrow & father(X,U), parent(U,Y). \end{array}$$

La clause C *est une clause* $2,2$*-déterminée et la clause* D *est une clause* $1,2$*-déterminée. La clause* D *est obtenue à partir de la clause* C *par suppression des littéraux non* $1,2$*-déterminés. Cette stratégique d'élagage des littéraux non* i,j*-déterminés est utilisée par GOLEM afin d'obtenir des clauses* i,j*-déterminées.*

Les clauses i,j-déterminées sont également des clauses syntaxiquement génératives (cf. définition 38). Rappelons que les clauses syntaxiquement génératives ont la propriété d'avoir un h-easy modèle fini pour une valeur de h donnée. On peut donc construire, à partir d'un ensemble de clauses génératives, un ensemble fini d'atomes fondés. Il en est donc de même pour les clauses i,j-déterminées. Cet ensemble d'atomes fondés constitue une théorie utilisable dans le calcul d'un moindre généralisé de clauses i,j-déterminées.

Dans le cas de GOLEM, une théorie du domaine T doit être un ensemble de clauses syntaxiquement génératives et i,j-déterminées tandis que les exemples $E = \{e_1, \cdots, e_n\}$ doivent être des atomes fondés. On peut ainsi construire le h-easy modèle de T, $M_h(T)$, qui correspond à un ensemble fini d'atomes fondés de T pour une profondeur donnée h. Cet ensemble d'atomes fondés est alors utilisé comme théorie pour calculer le moindre généralisé relatif à la théorie $M_h(T)$ de l'ensemble d'exemples E, c'est-à-dire : $\mathsf{rlgg}_{M_h(T)}(E)$. Afin de conserver la propriété d'i,j-déterminisme, seuls les littéraux i,j-déterminés de $\mathsf{rlgg}_{M_h(T)}(E)$ sont conservés. Dans le but d'effectuer ces traitements efficacement, Muggleton et Feng proposent d'utiliser un système de modes d'entrée-sortie pour les variables. Ceci permet de limiter le domaine des variables et donc le nombre d'instanciations possibles (pour plus de détails voir [Shapiro 1983]). Il n'est pas rare de voir ce système de modes utilisé, comme dans [Raedt 1997] ou bien récemment dans [Stolle 2005, Ray 2008]. Il est généralement complété par un typage restreignant davantage le domaine d'instanciation des variables. On obtient ainsi ce que Muggleton appelle *un moindre généralisé* i,j*-déterminé*. Ce dernier a la propriété d'être de taille finie et non exponentielle par rapport au nombre d'exemples. Le théorème suivant nous en donne plus de détails.

Théorème 15. *Soit une théorie du domaine* T *constituée de clauses syntaxiquement génératives et un ensemble d'exemples qui sont des atomes fondés* $e_1, \cdots, e_n$*. Si* t *est le nombre de termes apparaissant dans* $lgg(e_1 \cdots, e_n)$ *et* m *le nombre de symboles de prédicats distincts dans* $M_h(T)$*, alors le nombre de littéraux du* $\mathsf{rlgg}_{M_h(T)}(e_1, \cdots, e_n)$ i,j*-déterminé est au plus de* $\mathcal{O}((t \times m \times h)^{i^j})$*.*

Ce théorème nous indique qu'il est nécessaire de conserver des valeurs de i et j faibles au risque de rendre le calcul d'un moindre généralisé i,j-déterminé intractable. Un moindre généralisé i,j-déterminé est construit à l'aide de l'algorithme de moindre généralisation de Plotkin, il peut donc contenir des littéraux redondants que la notion d'i,j-déterminée n'empêche pas. Il faut donc le réduire si nécessaire.

Les clauses i,j-déterminées possèdent donc des propriétés intéressantes comme un moindre généralisé de taille finie et non exponentielle par rapport au nombre d'exemples. Cependant, ce dernier peut être de taille exponentielle en fonction des valeurs de i et j. Ainsi, cela impose

d'avoir de petites valeurs de i et j afin de conserver une complexité raisonnable pour le calcul du moindre généralisé. Cette contrainte rend l'usage de Golem impossible pour certaines tâches comme l'apprentissage de propriétés chimiques de molécules [Muggleton 2010].

4.4 Conclusion

Nous avons vu au travers de ce chapitre, les notions de base nécessaires à la compréhension d'un problème en PLI. Nous avons également rappelé les différentes relations de subsomption que sont l'implication logique, la θ-subsomption, ... ainsi que d'autres relations moins connues. L'accent a notamment été mis sur la difficulté d'un apprentissage utilisant l'implication logique, de par sa complexité et son indécidabilité, mais également l'attrait que cette dernière représente de par sa signification. Afin de palier ce problème, une direction nous semble intéressante. Il s'agit d'utiliser la θ-subsomption comme relation de subsomption. Elle semble être un bon candidat de par son respect de l'implication logique, sa calculabité mais également l'existence d'un moindre généralisé unique pour tout ensemble de clauses. Cependant, celui-ci possède des inconvénients comme une taille pouvant être exponentielle dans le nombre d'exemples généralisés. Certaines restrictions portant sur des sous-classes de clauses de Horn ont ainsi été regardées afin d'identifier des restrictions susceptibles de limiter les désagréments de la θ-subsomption. Nous allons donc pouvoir nous pencher dans le chapitre suivant sur le problème de classification de documents semi-structurés en basant notre approche sur l'usage de la θ-subsomption ainsi que sur son moindre généralisé.

Chapitre 5

Restrictions sur les clauses pour la classification

Sommaire

Dans ce chapitre, nous nous focalisons sur la tâche de classification d'arbres en PLI. Rappelons que nous nous sommes placés dans le cadre de la sémantique définie [Muggleton 1991, Muggleton 1994b] comme indiqué au chapitre 3 et avons choisi, lors du chapitre 4, la θ-subsomption [Plotkin 1970], comme relation de subsomption. Cette dernière bénéficie de l'unicité de son moindre généralisé.

L'étude de ce problème commence par la recherche de représentations logiques d'arbres. Nous présentons ainsi de nouvelles familles de clauses adaptées à la représentation d'arbres et de documents XML. Pour chacune de ces classes est proposée une réécriture plus efficace des opérations de base de l'apprentissage relationnel que sont la θ-subsomption et le calcul de moindre généralisé. Nous comparons principalement ces nouvelles familles à celle des *clauses déterminées* [Muggleton 1990, Quinlan 1991, Kietz 1994a] présentée dans le chapitre 4. Les clauses déterminées disposent d'un test de θ-subsumption polynomial dans la taille des clauses manipulées contrairement à la θ-subsomption de [Plotkin 1970] qui est exponentielle dans la taille des clauses (cf. chapitre 4). Nous justifions ce positionnement par le souhait de bénéficier d'opérations de subsomption et de réduction polynomiales en temps dans la taille de leurs entrées. Ces opérations forment un framework d'apprentissage pour chaque classe de clauses et seront utilisées afin d'infirmer ou confirmer une possible identification à la limite [Gold 1967] de ces classes. Enfin, nous réalisons des expériences dans le but d'observer leurs comportements vis-à-vis de jeux de données réelles.

5.1 Langages de clauses pour les arbres

Dans cette section, nous présentons différents langages de clauses permettant la représentation d'arbres ordonnés, d'arité bornée et étiquetés sur un alphabet donné. Ces langages vont être utilisés comme langage d'hypothèses lors d'un apprentissage. Leur finalité est ainsi la représentation de documents XML. Ils sont inspirés de la définition d'arbre et des différents codages sans perte d'arbres présentés au chapitre 3. La tâche de classification portant sur des documents XML et afin de respecter la sémantique définie, un exemple correspond, ici, à une clause d'un langage d'exemples et décrit un document XML. Par souci de simplicité, nous omettons l'atome de tête de ces clauses et représentons une clause par son corps. Nous justifions cela en considérant que la tête d'une clause, dans notre problème de classification, n'a pour but que d'indiquer la classe de la clause considérée. Ainsi, les codages introduits permettent de représenter un document XML indépendamment de sa classe. La tête de ces clauses sera réintroduite lors de l'étude de l'apprentissage.

5.1.1 Langage $\mathcal{L}^{T_\Sigma}_{child}$

Notre première représentation clausale d'un arbre tire profit de la définition 42 d'arbre du chapitre 3. Rappelons qu'un arbre t ordonné, d'arité non bornée et étiqueté sur un alphabet Σ est défini par un tuple $(N_t, root_t, child_t, ns_t, lab^t_\Sigma)$. Dans ce tuple, N_t est l'ensemble des nœuds de t, $root_t$ la racine de t, $child_t$ l'ensemble des couples de nœuds de la relation père-fils de t, ns_t l'ensemble des couples de nœuds de la relation de fraternité de t et lab^t_Σ la fonction d'étiquetage des nœuds de t. Cette représentation utilise les relations $child_t$, ns_t ainsi que de la fonction lab^t_Σ afin de transformer chaque arbre t d'un ensemble d'arbres T_Σ

en une unique clause C_t. L'ensemble des arbres de T_Σ, transformés de la manière suivante, forme alors l'ensemble des clauses $\mathcal{L}^{T_\Sigma}_{child}$:

Définition 76 (Langage $\mathcal{L}^{T_\Sigma}_{child}$)**.** *Le langage $\mathcal{L}^{T_\Sigma}_{child}$ est l'ensemble des clauses sans constante ni symbole de fonction basées sur l'utilisation des symboles de prédicat child/2, ns/2 et a/1 avec $a \in \Sigma$ tel qu'une clause $C_t = \ell_1, \cdots, \ell_k$ appartient à $\mathcal{L}^{T_\Sigma}_{fcns}$ si et seulement s'il existe un arbre $t \in T_\Sigma$ tel que :*

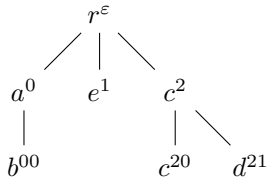
1. *un atome $child(X_n, X_m) \in C_t$ ssi il existe un arc $(n, m) \in child_t$;*
2. *un atome $ns(X_n, X_m) \in C_t$ ssi il existe un arc $(n, m) \in ns_t$;*
3. *un atome $a(X_n) \in C_t$ ssi il existe un nœud $n \in N_t$ avec $lab^t_\Sigma(n) = a$*
4. *pour tout littéral $\ell_{1<i} \in C_t$, il existe au moins un littéral $\ell_{1\leq j<i} \in C_t$ tel qu'au moins une variable de ℓ_i apparait également dans ℓ_j.*

Suivant ce codage, chaque arbre de T_Σ est représenté par une unique clause connectée. Les atomes binaires présents dans le corps d'une clause sont construits à partir des ensembles *child* et *ns* tandis que ceux unaires proviennent de la fonction d'étiquetage lab_Σ. La clause C_t obtenue à partir d'un arbre t contient ainsi $|ns_t| + |child_t| + |N_t|$ littéraux. Les ensembles de symboles de prédicat, de symboles de fonction et de constantes sont finis.

Exemple 34. *Soit un arbre $t \in T_\Sigma$ où $t = (N_t, \varepsilon, child_t, ns_t, lab^t_\Sigma)$ tel que :*

$$
\begin{array}{lll}
\Sigma = \{r, a, b, c, d, e\} & N_t & = \{\varepsilon, 0, 1, 2, 00, 20, 21\} \\
child_t & & = \{(\varepsilon, 0),\ (\varepsilon, 1),\ (\varepsilon, 2),\ (0, 00),\ (2, 20), (2, 21)\} \\
ns_t & & = \{(0, 1),\ (1, 2),\ (20, 21)\} \\
lab^t_\Sigma(\varepsilon) = r, & lab^t_\Sigma(0) = a, & lab^t_\Sigma(1) = e, \quad lab^t_\Sigma(2) = c, \quad lab^t_\Sigma(00) = b, \\
lab^t_\Sigma(20) = c, & lab^t_\Sigma(21) = d &
\end{array}
$$

présenté ci-dessous et représenté par la clause $C_t \in \mathcal{L}^{T_\Sigma}_{child}$ suivante :



$t \in T_\Sigma$

$$
\begin{array}{lll}
C_t & = & child(X_\varepsilon, X_0),\ ns(X_0, X_1), \\
 & & child(X_\varepsilon, X_1),\ ns(X_1, X_2), \\
 & & child(X_\varepsilon, X_2),\ child(X_0, X_{00}), \\
 & & child(X_2, X_{20}),\ ns(X_{20}, X_{21}), \\
 & & child(X_2, X_{21}), \\
 & & r(X_\varepsilon),\ a(X_0),\ e(X_1), \\
 & & c(X_2),\ b(X_{00}),\ c(X_{20}),\ d(X_{21}).
\end{array}
$$

5.1.2 Langage $\mathcal{L}^{T_\Sigma}_{fcns}$

Nous présentons une seconde représentation clausale d'arbre inspirée du codage *first-child next-sibling* [Rabin 1969, Comon 1997] présenté au chapitre 3 sous-section 3.1.4. Ce codage sans perte a pour intérêt de transformer un arbre d'arité non bornée en un arbre binaire. Cette transformation est rendue possible en ne considérant plus l'ensemble de tous les arcs présents dans $child_t$ d'un arbre t à transformer mais seulement un sous-ensemble de celui-ci, nommé fc_t pour *first-child*. Ce sous-ensemble correspond à l'ensemble des arcs liant un nœud à son premier fils. Donnons maintenant la définition du langage $\mathcal{L}^{T_\Sigma}_{fcns}$ permettant la construction des clauses issues de l'ensemble des arbres T_Σ suivant le codage *first-child next-sibling*.

Définition 77 (Langage $\mathcal{L}^{T_\Sigma}_{fcns}$)**.** *Le langage $\mathcal{L}^{T_\Sigma}_{fcns}$ est l'ensemble des clauses sans constante ni symbole de fonction basées sur l'utilisation des symboles de prédicat $fc/2$, $ns/2$ et $a/1$ avec $a \in \Sigma$ tel qu'une clause $C_t = \ell_1, \cdots, \ell_k$ appartient à $\mathcal{L}^{T_\Sigma}_{fcns}$ si et seulement s'il existe un arbre $t \in T_\Sigma$ tel que :*

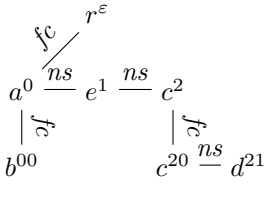
1. *un atome $fc(X_n, X_m) \in C_t$ ssi il existe un arc $(n, m) \in fc_t$;*
2. *un atome $ns(X_n, X_m) \in C_t$ ssi il existe un arc $(n, m) \in ns_t$;*
3. *un atome $a(X_n) \in C_t$ ssi il existe un nœud $n \in N_t$ avec $lab^t_\Sigma(n) = a$*
4. *pour tout littéral $\ell_{1<i} \in C_t$, il existe au moins un littéral $\ell_{1\leq j<i} \in C_t$ tel qu'au moins une variable de ℓ_i apparait également dans ℓ_j.*

À l'instar du précédent codage, chaque arbre de T_Σ est représenté par une unique clause connectée de $\mathcal{L}^{T_\Sigma}_{fcns}$. Une clause C_t obtenue à partir d'un arbre t selon ce codage contient $|fc_t|+|ns_t|+|N_t|$ littéraux. L'ensemble *fc* est un sous-ensemble de *child*. Ainsi, pour un arbre $t \in T_\Sigma$, la clause $C_t \in \mathcal{L}^{T_\Sigma}_{fcns}$ a une taille inférieure ou égale à son homologue $C'_t \in \mathcal{L}^{T_\Sigma}_{child}$. Enfin, comme pour le langage $\mathcal{L}^{T_\Sigma}_{child}$, les ensembles de symboles de prédicat, de fonctions et de constantes sont finis.

Exemple 35. *Nous reprenons l'arbre t de l'exemple 34. Le codage* first-child next-sibling *permet de définir, en plus de l'ensemble ns_t, l'ensemble fc_t suivant :*

$$fc_t \quad = \quad \{(\varepsilon, 0),\ (0, 00),\ (2, 20)\}$$

On obtient ainsi l'arbre ci-dessous à partir duquel est construit la clause $C_t \in \mathcal{L}^{T_\Sigma}_{fcns}$:



Arcs fc_t et ns_t de l'arbre $t \in T_\Sigma$

$$\begin{aligned} C_t \quad = \quad & fc(X_\varepsilon, X_0),\ fc(X_0, X_{00}), \\ & ns(X_0, X_1),\ ns(X_1, X_2), \\ & fc(X_2, X_{20}),\ ns(X_{20}, X_{21}) \\ & r(X_\varepsilon),\ a(X_0),\ e(X_1), \\ & c(X_2),\ b(X_{00}),\ c(X_{20}),\ d(X_{21}). \end{aligned}$$

5.1.3 Le langage $\mathcal{L}^{T_\Sigma}_{nested}$

Le codage présenté ici se base sur les *nested words* dont la définition se trouve au chapitre 3 sous-section 3.1.4. Rappelons qu'un *nested word* est une représentation sous forme de mot d'une structure arborescente. Un arbre $t \in T_\Sigma$ se présente ainsi sous la forme d'un mot $w_t \in W^{T_\Sigma}_{nested}$ qui est un sous-ensemble de $\hat{\Sigma}^\star$ où $\hat{\Sigma} = \{open, close\} \times \Sigma$. Un mot $w_t \in W^{T_\Sigma}_{nested}$ est ensuite codé sous la forme d'une clause $C_t \in \mathcal{L}^{T_\Sigma}_{nested}$. La $i^{\text{ième}}$ lettre de ce mot, $w_i \in \hat{\Sigma}$, est représentée dans la clause C_t par un atome unaire $w_i(X_i)$ où la variable X_i identifie la position de cette lettre dans le mot. Deux variables X_i et X_{i+1} identifiant respectivement deux lettres voisines, w_i et w_{i+1}, d'un mot $w_t \in W^{T_\Sigma}_{nested}$, sont liées par un atome binaire $next(X_i, X_{i+1})$. Donnons maintenant une définition plus formelle de la construction de l'ensemble des clauses $\mathcal{L}^{T_\Sigma}_{nested}$ à partir de T_Σ.

Définition 78. *Le langage $\mathcal{L}^{T_\Sigma}_{nested}$ est l'ensemble des clauses sans constante ni symbole de fonction basées sur l'utilisation des symboles de prédicat $next/2$ et $a/1$ avec $a \in \hat{\Sigma}$ tel qu'une clause $C_t = \ell_1, \cdots, \ell_k$ appartient à $\mathcal{L}^{T_\Sigma}_{nested}$ si et seulement s'il existe le nested word $w_t = w_0 w_1 \cdots w_{n-1} w_n \in W^{T_\Sigma}_{nested}$ de $t \in T_\Sigma$ tel que :*

1. *un atome* $w_i(X_i) \in C_t$ *ssi* $0 \leqslant i \leqslant n$,
2. *un atome* $next(X_i, X_{i+1}) \in C_t$ *ssi* $0 \leqslant i < n$,
3. *pour tout littéral* $\ell_{1<i} \in C_t$, *il existe au moins un littéral* $\ell_{1\leq j<i} \in C_t$ *tel qu'au moins une variable de* ℓ_i *apparait également dans* ℓ_j.

Chaque arbre $t \in T_\Sigma$ est représenté par une unique clause connectée $C_t \in \mathcal{L}^{T_\Sigma}_{nested}$ et réciproquement. Une clause C_t contient ainsi $4 \times |N_t| - 1$ littéraux. Les ensembles de symboles de prédicat et de fonction nécessaires à la construction de l'ensemble $\mathcal{L}^{T_\Sigma}_{nested}$ sont finis comme ceux de $\mathcal{L}^{T_\Sigma}_{child}$ et $\mathcal{L}^{T_\Sigma}_{fcns}$.

Exemple 36. *Reprenons l'arbre t de l'exemple 34. Grâce aux* nested words*, nous pouvons le coder sous la forme d'un mot de* $\hat{\Sigma}^\star$ *où :*

$$\begin{aligned}\hat{\Sigma} \quad = \quad & \{(open, a), (close, a), (open, b), (close, b), (open, c), (close, c),\\ & (open, d), (close, d),\ (open, e), (close, e),\ (open, r), (close, r)\}\end{aligned}$$

Le nested word *associé à l'arbre t est le suivant :*

$$\begin{aligned}w_t \quad = \quad & \textit{(open,r)(open,a)(open,b)(close,b)(close,a)(open,e)(close,e)}\\ & \textit{(open,c)(open,c)(close,c)(open,d)(close,d)(close,c)(close,r)}\end{aligned}$$

Afin de rendre plus lisible la clause construite, nous utiliserons le symbole a_{op} *à la place du couple* $(open, a)$ *et* a_{cl} *à la place de* $(close, a)$*. Ce mot est ainsi représenté par la clause de* $\mathcal{L}^{T_\Sigma}_{nested}$ *suivante :*

$$\begin{aligned}C_t \quad = \quad & next(X_1, X_2), r_{\mathsf{op}}(X_1),\\ & a_{\mathsf{op}}(X_2),\ next(X_2, X_3),\ b_{\mathsf{op}}(X_3),\ next(X_3, X_4),\ b_{\mathsf{cl}}(X_4),\ next(X_4, X_5),\ a_{\mathsf{cl}},\\ & next(X_5, X_6),\ e_{\mathsf{op}}(X_6),\ next(X_6, X_7),\ e_{\mathsf{cl}}(X_7),\\ & next(X_7, X_8),\ c_{\mathsf{op}}(X_8),\ next(X_8, X_9),\ c_{\mathsf{op}}(X_9),\ next(X_9, X_{10}),\ c_{\mathsf{cl}}(X_{10}),\ next(X_{10}, X_{11}),\\ & d_{\mathsf{op}}(X_11),\ next(X_{11}, X_{12}),\ d_{\mathsf{cl}}(X_12),\ next(X_{12}, X_{13}),\ c_{\mathsf{cl}}(X_{13}), next(X_{13}, X_{14}),\ r_{\mathsf{cl}}(X_{14}).\end{aligned}$$

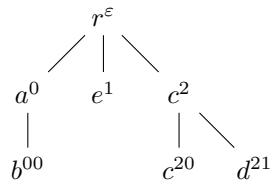
5.1.4 Langage $\mathcal{L}^{T_\Sigma}_{pc}$

Terminons avec un dernier codage sans perte reposant sur l'identification unique de chaque nœud d'un arbre. Ce principe d'identification unique a déjà été utilisé avec succès pour le traitement de requêtes sur des bases de données [Tropashko 2005, Faroult 2006]. Dans ce codage, chaque nœud d'un arbre est identifié par un mot $w_0 w_1 \cdots w_n \in \mathbb{N}^\star$. La lettre w_i de ce mot est le nombre de frères de gauche du $i^{\text{ième}}$ nœud du chemin de la racine de cet arbre à ce nœud. Pour plus de lisibilité, nous utilisons le symbole "." comme opérateur de concaténation et associons ainsi le symbole `0` à la racine de l'arbre, puis `0.0` à son premier fils, `0.1` au deuxième, puis `0.0.0` au premier petit-fils de la racine, ... Une clause $C_t \in \mathcal{L}^{T_\Sigma}_{pc}$ représentant un arbre $t \in T_\Sigma$ contient un atome unaire pour chaque nœud de cet arbre. Un atome a pour symbole de prédicat l'identifiant de son nœud et pour argument l'étiquette de son nœud sous la forme d'une constante. De cette manière, chaque atome utilise un symbole de prédicat qui lui est propre. Nous définissions de manière plus formelle le langage $\mathcal{L}^{T_\Sigma}_{pc}$ représentant ces clauses.

Définition 79 (Langage $\mathcal{L}^{T_\Sigma}_{pc}$)**.** *Le langage* $\mathcal{L}^{T_\Sigma}_{pc}$ *est l'ensemble des clauses fondées basées sur l'utilisation de prédicats unaires de l'ensemble* $\mathcal{P} = \{p/1 \mid p \in \mathbb{N}^\star\}$ *ainsi que sur les constantes* $a \in \Sigma$ *tel qu'une clause* C_t *appartient à* $\mathcal{L}^{T_\Sigma}_{pc}$ *si et seulement s'il existe un arbre* $t \in T_\Sigma$ *tel qu'un atome* $p(a) \in C_t$ *si et seulement s'il existe un nœud* $n \in N_t$ *avec* $p = 0.n$ *et* $lab^t_\Sigma(n) = a$.

Chaque arbre $t \in T_\Sigma$ est représenté par une unique clause non connectée $C_t \in \mathcal{L}_{pc}^{T_\Sigma}$ et réciproquement. Une clause C_t contient $|N_t|$ littéraux et a ainsi une taille inférieure à celle de ses homologues des ensembles $\mathcal{L}_{child}^{T_\Sigma}$, $\mathcal{L}_{fcns}^{T_\Sigma}$ et $\mathcal{L}_{nested}^{T_\Sigma}$. Cependant, contrairement à ceux-ci, l'ensemble des prédicats nécessaire pour coder les clauses est infini. Enfin, remarquons que les littéraux d'une clause ne sont pas ordonnés. Une clause de ce langage est alors un ensemble de littéraux contrairement aux langages ultérieurement définis où les clauses sont des séquences.

Exemple 37. *Reprenons l'arbre t de l'exemple 34. Nous n'utiliserons cette fois-ci que l'ensemble des nœuds N_t ainsi que la fonction d'étiquetage lab_Σ^t afin de construire la clause suivante appartenant à $\mathcal{L}_{pc}^{T_\Sigma}$:*



$t \in T_\Sigma$

$$C_t = 0(r),\ 0.0(a),\ 0.1(e),\ 0.2(c),\ 0.0.0(c),\ 0.2.0(c),\ 0.2.1(d).$$

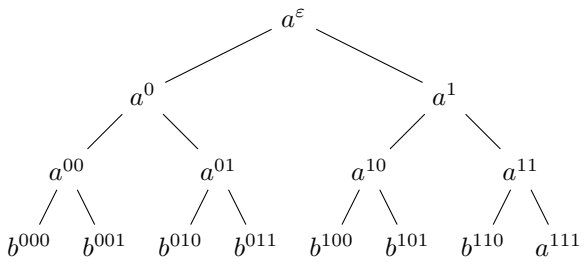
5.1.5 Pré-sélection des codages pour notre problème

Nous réalisons maintenant une étude préliminaire des codages proposés. Rappelons que nous avons fait le choix d'utiliser comme relation de subsomption la θ-subsomption [Plotkin 1970] et comme opérateur de généralisation le moindre généralisé [Plotkin 1971b, Plotkin 1971a]. Ces opérations vont être utilisées intensivement lors d'un apprentissage afin, entre autres, de tester la couverture d'une hypothèse, l'équivalence de deux hypothèses, réduire une hypothèse et assurer la généralisation d'exemples dans le but de produire des hypothèses. Elles forment le noyau du framework d'apprentissage que nous souhaitons proposer. Le but de cette étude est alors de confirmer ou d'infirmer l'intérêt de chaque codage pour notre tâche de classification de documents XML. Ainsi, nous leur cherchons des prédispositions facilitant l'usage de la θ-subsomption, en diminuant sa complexité par exemple, ou celui du moindre-généralisé, en bénéficiant d'une absence d'explosion de sa taille en fonction du nombre d'exemples. Nous présentons également les inconvénients de ces codages s'ils sont rédhibitoires pour un apprentissage efficace.

Codage $\mathcal{L}_{child}^{T_\Sigma}$

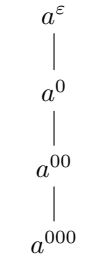
Commençons par le langage $\mathcal{L}_{child}^{T_\Sigma}$. Le côté naturel et propre à la définition d'arbre rend ce codage intéressant pour notre problème. Cependant, celui-ci nous semble peu ou pas attrayant du point de vue de la θ-subsomption. En effet, sans avoir considéré les conséquences d'une généralisation, nous pouvons voir avec l'exemple suivant qu'un test de θ-subsomption entre deux clauses de $\mathcal{L}_{child}^{T_\Sigma}$ risque de se révéler couteux.

Exemple 38. *Soit deux arbres $t, t' \in T_\Sigma$ ainsi que les clauses $C_t, C_{t'} \in \mathcal{L}_{child}^{T_\Sigma}$:*



$t \in T_\Sigma$

$$
\begin{aligned}
C_t \quad = \quad & \mathit{child}(X_\varepsilon, X_0),\ \mathit{child}(X_\varepsilon, X_1),\\
& \mathit{child}(X_0, X_{00}),\ \mathit{child}(X_0, X_{01}),\\
& \mathit{child}(X_{00}, X_{000}),\ \mathit{child}(X_{00}, X_{001}),\\
& \mathit{child}(X_{01}, X_{010}),\ \mathit{child}(X_{00}, X_{011}),\\
& \mathit{child}(X_1, X_{10}),\ \mathit{child}(X_1, X_{11}),\\
& \mathit{child}(X_{10}, X_{100}),\ \mathit{child}(X_{10}, X_{101}),\\
& \mathit{child}(X_{11}, X_{110}),\ \mathit{child}(X_{11}, X_{111}),\\
& \mathit{ns}(X_0, X_1),\ \mathit{ns}(X_{00}, X_{01}),\ \mathit{ns}(X_{100}, X_{101}),\\
& \mathit{ns}(X_{110}, X_{111}),\\
& \mathit{ns}(X_{000}, X_{001}),\ \mathit{ns}(X_{010}, X_{010}),\ \mathit{ns}(X_{10}, X_{11})\\
& a(X_\varepsilon),\ a(X_0),\ a(X_1),\ a(X_{00}),\ a(X_{01}),\\
& a(X_{10}),\ a(X_{11}),\ b(X_{000}),\ b(X_{001}),\\
& b(X_{010}),\ b(X_{011}),\ b(X_{100}),\ b(X_{101}),\\
& b(X_{110}),\ a(X_{110}).
\end{aligned}
$$



$t' \in T_\Sigma$

$$
\begin{aligned}
C_{t'} \quad = \quad & \mathit{child}(Y_\varepsilon, Y_0),\ \mathit{child}(Y_0, Y_{00}),\\
& \mathit{child}(Y_{00}, Y_{000}),\\
& a(Y_\varepsilon),\ a(Y_0),\ a(Y_{00}),\ a(Y_{000}).
\end{aligned}
$$

Le test visant à savoir si la clause $C_{t'}$ θ-subsume la clause C_t entraîne l'énumération d'un nombre exponentiel de substitutions avant de retourner une réponse positive. Il tente, entre autres, les substitutions suivantes :

$$
\begin{array}{ll}
\{Y_\varepsilon/X_\varepsilon, Y_0/X_0, Y_{00}/X_{00}, Y_{000}/X_{000}\}, & \{Y_\varepsilon/X_\varepsilon, Y_0/X_0, Y_{00}/X_{00}, Y_{000}/X_{001}\}\\
\{Y_\varepsilon/X_\varepsilon, Y_0/X_0, Y_{00}/X_{01}, Y_{000}/X_{010}\}, & \{Y_\varepsilon/X_\varepsilon, Y_0/X_0, Y_{00}/X_{01}, Y_{000}/X_{011}\}\\
\{Y_\varepsilon/X_\varepsilon, Y_0/X_1, Y_{00}/X_{10}, Y_{000}/X_{100}\}, & \{Y_\varepsilon/X_\varepsilon, Y_0/X_1, Y_{00}/X_{10}, Y_{000}/X_{101}\}\\
\{Y_\varepsilon/X_\varepsilon, Y_0/X_1, Y_{00}/X_{11}, Y_{000}/X_{110}\} &
\end{array}
$$

avant de trouver la substitution $\{Y_\varepsilon/X_\varepsilon, Y_0/X_1, Y_{00}/X_{11}, Y_{000}/X_{111}\}$ clôturant le test. Cette énumération est causée par l'absence, dans la clause $C_{t'}$, de littéraux dont le prédicat est ns. L'ordre induit par les littéraux de C_t usant de ce même prédicat est alors ignoré et l'arbre t considéré comme non ordonné, i.e. sans ordre entre les fils d'un même nœud.

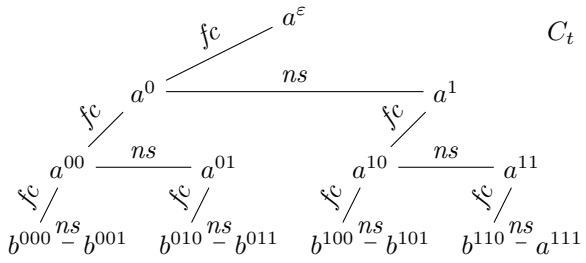
L'énumération des substitutions durant un test de θ-subsomption justifie la complexité de cette opération pour les clauses de Horn. Elle se traduit notamment par la présence de retours en arrière mettant en avant l'absence de déterminisme de ce codage. Ne disposant pas naturellement de contraintes limitant le nombre d'appariements des littéraux, il semble difficile de proposer une θ-subsomption polynomiale pour ce codage. Nous ne le retenons donc pas pour la suite de notre étude.

Codage $\mathcal{L}^{T_\Sigma}_{fcns}$

Le codage *first-child next-sibling* permet la transformation d'un arbre d'arité non bornée en un arbre d'arité binaire. Ceci est rendu possible en omettant une partie des arêtes présentes dans *child*. Plus précisément, seules les arêtes liant un nœud à son premier fils sont conservées afin de former l'ensemble d'arcs *fc*. Cette construction a pour conséquence

de limiter le nombre d'appariements possibles des littéraux usant du prédicat *fc*. Illustrons nos propos avec cet exemple.

Exemple 39. *Reprenons les arbres $t, t' \in T_\Sigma$ de l'exemple 38 et construisons les clauses $C_t, C_{t'} \in \mathcal{L}^{T_\Sigma}_{fcns}$:*



$$
\begin{aligned}
C_t \quad = \quad & fc(X_\varepsilon, X_0),\ fc(X_0, X_{00}),\ fc(X_{00}, X_{000}),\\
& ns(X_{000}, X_{001}),\ ns(X_{00}, X_{01}),\ fc(X_{01}, X_{010}),\\
& ns(X_{010}, X_{011}),\ ns(X_0, X_1),\ fc(X_1, X_{10}),\\
& fc(X_{10}, X_{100}),\ ns(X_{100}, X_{101}),\ ns(X_{10}, X_{11}),\\
& fc(X_{11}, X_{110}),\ ns(X_{110}, X_{111}),\\
& a(X_\varepsilon),\ a(X_0),\ a(X_1),\ a(X_{00}),\ a(X_{01}),\\
& a(X_{10}),\ a(X_{11}),\ b(X_{000}),\ b(X_{001}),\\
& b(X_{010}),\ b(X_{011}),\ b(X_{100}),\ b(X_{101}),\\
& b(X_{110}),\ a(X_{110}).
\end{aligned}
$$

$t \in T_\Sigma$

a^ε — fc — a^0 — fc — a^{00} — fc — a^{000}

$t' \in T_\Sigma$

$$
\begin{aligned}
C_{t'} \quad = \quad & fc(Y_\varepsilon, Y_0),\ fc(Y_0, Y_{00}),\\
& fc(Y_{00}, Y_{000}),\\
& a(Y_\varepsilon),\ a(Y_0),\ a(Y_{00}),\ a(Y_{000}).
\end{aligned}
$$

Le test de θ-subsomption de la clause $C_{t'}$ vers la clause C_t commence par l'appariement du littéral $fc(Y_\varepsilon, Y_0)$ avec un littéral de la clause C_t. Une fois celui-ci apparié, le littéral suivant n'a, au plus, qu'un appariement possible et ainsi de suite pour les autres littéraux de la clause $C_{t'}$. L'échec de l'appariement d'un littéral ne remet donc en cause que l'appariement du premier littéral contrairement à l'exemple 38 où l'assignation de chaque littéral est remise en question.

Cet exemple montre que, dans le cas d'une θ-subsomption entre clauses de $\mathcal{L}^{T_\Sigma}_{fcns}$, l'appariement du premier littéral limite à, au plus, un appariement possible les littéraux suivants. Cette limitation nous semble intéressante pour la définition d'une θ-subsomption efficace sur ce langage. Elle est due à l'ordre imposé entre les littéraux par le point 3 de la définition 77, l'obligation qu'ont les clauses de $\mathcal{L}^{T_\Sigma}_{fcns}$ d'être connectées et à l'existence, pour un nœud donné, d'au plus une arête allant vers et une autre partant de ce nœud dans les ensembles *fc* et *ns*. Nous appelons ce phénomène *quasi-déterminisme* en référence à la notion de déterminisme où l'unique appariement possible du premier littéral fixe celui des autres littéraux. Comme vu au chapitre 4, le moindre généralisé de deux clauses est susceptible d'être déconnecté. Il est donc nécessaire de vérifier si l'avantage observé pour les clauses connectées de $\mathcal{L}^{T_\Sigma}_{fcns}$ existe également pour le moindre généralisé de clauses de $\mathcal{L}^{T_\Sigma}_{fcns}$. Nous nous intéressons donc à ce codage et allons formaliser son utilisation dans les sections suivantes afin de proposer une θ-subsomption et une réduction polynomiales.

Codage $\mathcal{L}^{T_\Sigma}_{nested}$

Le langage $\mathcal{L}^{T_\Sigma}_{nested}$ se rapproche quant-à-lui du codage $\mathcal{L}^{T_\Sigma}_{fcns}$ où le prédicat *next* serait substitué par le prédicat *fc* et le prédicat *ns* serait omis. En effet, pour une lettre donnée

d'un mot, il existe au plus un voisin de gauche et un voisin de droite. Ce même constat a été fait dans le cas du langage $\mathcal{L}_{fcns}^{T_\Sigma}$ où, pour chaque nœud d'un arbre, il existe au plus une arête partant et une autre allant vers ce nœud dans les ensembles fc et ns. De cette manière, un mot de $\hat{\Sigma}^\star$ peut être vu sous la forme d'un arbre ordonné, étiqueté sur l'alphabet $\hat{\Sigma}$ et d'arité bornée à 1, c'est-à-dire dont l'ensemble ns d'arcs est vide. Nous obtenons ainsi $\mathcal{L}_{nested}^{T_\Sigma} \subset \mathcal{L}_{fcns}^{T_\Sigma}$. L'étude théorique de ce codage est alors réalisée lors de celle du langage $\mathcal{L}_{fcns}^{T_{\hat{\Sigma}}}$ et donc de celle de $\mathcal{L}_{fcns}^{T_\Sigma}$ où l'alphabet Σ est égal à $\hat{\Sigma}$. Une contrainte existant sur l'alphabet, le langage $\mathcal{L}_{nested}^{T_\Sigma}$ est néanmoins un cas particulier du langage $\mathcal{L}_{fcns}^{T_{\hat{\Sigma}}}$. De ce fait, nous l'utiliserons principalement lors de nos expériences afin d'étudier l'influence de l'aplatissement d'un arbre.

Codage $\mathcal{L}_{pc}^{T_\Sigma}$

Le codage $\mathcal{L}_{pc}^{T_\Sigma}$ appartient à la famille des clauses déterminées [Quinlan 1991, Kietz 1994a]. Cette affirmation s'explique par le fait que chaque nœud est représenté par un unique symbole de prédicat. Ainsi, lors d'un test de subsomption, un atome peut tout au plus être apparié à un autre atome. À l'instar des clauses i, j-déterminées de Muggleton et Feng [Muggleton 1990], il nous semble donc possible d'en tirer profit pour les opérations de subsomption, de généralisation et de réduction polynomiales. Nous poursuivrons donc l'étude de ce dernier.

Cette pré-sélection a mis en avant les codages $\mathcal{L}_{pc}^{T_\Sigma}$, $\mathcal{L}_{nested}^{T_\Sigma}$ et $\mathcal{L}_{fcns}^{T_\Sigma}$ au détriment de $\mathcal{L}_{child}^{T_\Sigma}$. Ces codages présentent des propriétés se rapprochant du déterminisme. Nous procédons donc à leur étude théorique afin de définir des opérations de θ-subsomption, de réduction et de moindre généralisation efficaces.

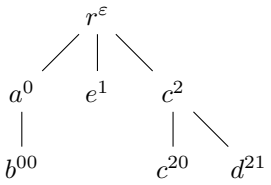
5.2 Apprentissage du langage $\mathcal{L}_{pc}^{T_\Sigma}$

L'étude théorique d'un codage est dirigée par la tâche d'apprentissage visée qu'est la classification de documents XML. Elle consiste en la proposition et l'étude d'opérations de θ-subsomption, de moindre généralisation et de réduction adaptées à nos codages. Ces opérateurs doivent fonctionner à la fois sur les exemples ainsi que sur les généralisations de ces exemples afin d'être utilisés lors d'un apprentissage en ILP. Ils devront être de complexité polynomiale dans la taille de leur entrée afin de permettre un apprentissage efficace. Cette étude débute en vérifiant que le langage de clauses $\mathcal{L}_{pc}^{T_\Sigma}$ est clos par moindre généralisation. Si ce n'est pas le cas, nous définissons un langage clos par moindre généralisé incluant $\mathcal{L}_{pc}^{T_\Sigma}$. Nous proposons ensuite les opérations nécessaires à un framework d'apprentissage. Puis, nous étudions la taille des moindres généralisés produits par nos codages afin d'affirmer ou non l'existence d'une explosion de leur taille dans le nombre d'exemples. Enfin, nous tentons de généraliser, si cela est possible, ce langage afin d'obtenir des algorithmes pour des familles de clauses plus vastes.

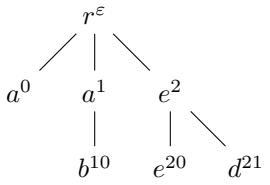
5.2.1 Clôture de $\mathcal{L}_{pc}^{T_\Sigma}$

Commençons cette étude en invalidant l'hypothèse de clôture par moindre généralisation du langage $\mathcal{L}_{pc}^{T_\Sigma}$ à l'aide de l'exemple suivant.

Exemple 40. *Soit deux clauses* $C_t, C_{t'} \in \mathcal{L}_{pc}^{T_\Sigma}$ *:*



$t \in T_\Sigma$

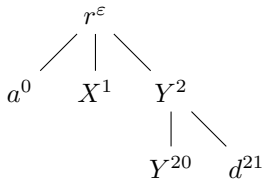


$t' \in T_\Sigma$

$$C_t = 0(r),\ 0.0(a),\ 0.1(e),\ 0.2(c),\ 0.0.0(c),\ 0.2.0(c),\ 0.2.1(d).$$

$$C_{t'} = 0(r),\ 0.0(a),\ 0.1(a),\ 0.2(e),\ 0.1.0(b),\ 0.2.0(e),\ 0.2.1(d).$$

Les clauses C_t et $C_{t'}$ ont pour moindre généralisé la clause $C_{t''}$ suivante :

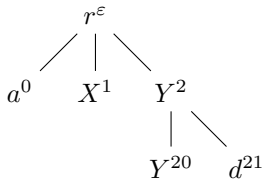


t''

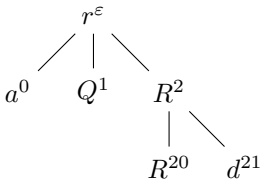
$$C_{t''} = 0(r),\ 0.0(a),\ 0.1(X),\ 0.2(Y),\ 0.2.0(Y),\ 0.2.1(d).$$

Dans l'exemple 40, la clause moindre généralisée obtenue à partir de deux clauses $\mathcal{L}_{pc}^{T_\Sigma}$ n'est pas une clause de $\mathcal{L}_{pc}^{T_\Sigma}$. En effet, les nœuds de l'arbre t'' de l'exemple 40 ne sont pas tous étiquetés par une lettre de Σ. Ainsi, il n'appartient pas à T_Σ. L'ensemble $\mathcal{L}_{pc}^{T_\Sigma}$ n'est donc pas clos par moindre généralisation sous θ-subsomption. Les nœuds de t'' non étiquetés par un symbole de Σ le sont par la variable de l'atome correspondant. Ce principe est illustré dans l'arbre t'' où l'on peut voir que les nœuds 1, 2 et 20 sont étiquetés par des variables. Constatons qu'une même variable peut étiqueter plusieurs nœuds. Ces nœuds sont alors étiquetés par une même lettre inconnue, celle-ci appartenant à Σ.

L'arbre t'' obtenu par moindre généralisation appartient à l'ensemble des arbres $T_{\Sigma\cup V}$ où V est un ensemble infini de variables comme défini dans la définition d'un langage (voir définition 7 du chapitre 2). La présence de variables dans les arbres soulève, comme cela a été le cas pour les clauses, la question d'équivalence. Ainsi, comme dans le cas des clauses, nous considérons que $T_{\Sigma\cup V}$ contient l'ensemble de tous les arbres construits sur l'alphabet $\Sigma \cup V$ modulo un renommage de variables. Ainsi, les arbres suivants :



t''



t'''

sont équivalents puisqu'il est possible de remplacer les variables X et Y de l'arbre t'' par les variables Q et R afin d'obtenir l'arbre t''' et réciproquement. Nous notons $T_{\Sigma\cup V}$ l'ensemble de ces arbres (modulo un renommage de variables) et $\mathcal{L}_{pc}^{T_{\Sigma\cup V}}$ l'ensemble des clauses représentant ces arbres suivant la construction de la définition 80.

Définition 80 (Langage $\mathcal{L}_{pc}^{T_{\Sigma\cup V}}$)**.** *Le langage $\mathcal{L}_{pc}^{T_{\Sigma\cup V}}$ est l'ensemble des clauses basées sur l'utilisation des symboles de prédicat unaires de l'ensemble $\mathcal{P} = \{p/1 \mid p \in \mathbb{N}^\star\}$, des variables*

de V et des constantes $a \in \Sigma$ tel qu'une clause C_t appartient à $\mathcal{L}_{pc}^{T_{\Sigma\cup V}}$ si et seulement s'il existe un arbre $t \in T_{\Sigma\cup V}$ tel qu'un atome $p(s) \in C_t$ si et seulement s'il existe un nœud $n \in N_t$ avec $p = 0.n$ et $s = lab^t_{\Sigma\cup V}(n)$.

L'ensemble $\mathcal{L}_{pc}^{T_{\Sigma\cup V}}$ contient tous les arbres construits sur l'alphabet $\Sigma \cup V$ et inclut donc l'ensemble des clauses $\mathcal{L}_{pc}^{T_\Sigma}$. Nous vérifions maintenant que ce langage est clos par moindre généralisation, puis proposons un algorithme de calcul de moindre généralisé lui étant propre.

Théorème 16. *Le langage $\mathcal{L}_{pc}^{T_{\Sigma\cup V}}$ est clos par moindre généralisation.*

Preuve. Nous prouvons ici que $\mathcal{L}_{pc}^{T_{\Sigma\cup V}}$ est clos par moindre généralisation. Nous savons que les termes présents dans les clauses de $\mathcal{L}_{pc}^{T_{\Sigma\cup V}}$ sont des constantes de Σ ou des variables. Il en est donc de même des termes d'un moindre généralisé de ces clauses.

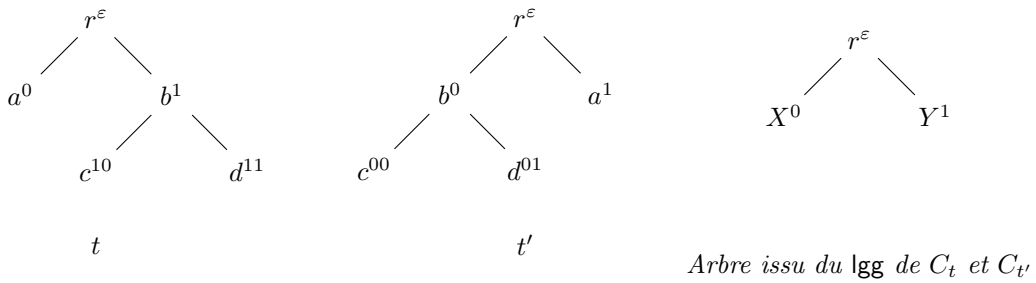
Nous savons également que chaque clause de $\mathcal{L}_{pc}^{T_{\Sigma\cup V}}$ a pour chaque symbole de prédicat au plus un atome dans une clause. Un atome de la première clause n'est donc généralisable qu'avec au plus un littéral de la seconde clause. Ainsi, seuls les atomes dont le symbole de prédicat est présent dans les clauses à généraliser sont conservés dans le moindre généralisé.

Enfin, par la définition de ce langage, nous savons que si un atome $wi(t_i)$ avec $0 < i$ où $i \in \mathbb{N}$ et $w \in \mathbb{N}^\star$ est présent dans une clause de $\mathcal{L}_{pc}^{T_{\Sigma\cup V}}$, alors il existe, dans cette même clause, tous les atomes représentant les ancêtres de ce nœud, les frères de gauche de ce nœud, ainsi que les frères de gauche de ses ancêtres. Ainsi, si un atome est présent dans le moindre généralisé, les atomes représentant ses ancêtres, ses frères de gauche et les frères de gauche de ses ancêtres sont également conservés dans ce moindre généralisé. De cette manière, la structure arborescente imposée par la construction de l'ensemble des nœuds est préservée dans le moindre généralisé.

Le moindre généralisé de deux clauses de $\mathcal{L}_{pc}^{T_{\Sigma\cup V}}$ représente donc bien un arbre dont les nœuds sont étiquetés par des lettres de Σ et des variables. Le langage $\mathcal{L}_{pc}^{T_{\Sigma\cup V}}$ est donc bien clos par moindre généralisation. □

Il faut souligner que cette preuve met en avant l'obligation pour un moindre généralisé de clauses $\mathcal{L}_{pc}^{T_{\Sigma\cup V}}$ de représenter un arbre. Ainsi, seuls les motifs d'arbres communs aux clauses à généraliser et reliés à la racine sont conservés. C'est ce qu'illustre l'exemple suivant :

Exemple 41. *Soit deux arbres t et t' appartenant à $T_{\Sigma\cup V}$:*



Dans l'exemple 41, le sous-arbre $b(c(), d())$ de t n'est pas conservé dans le moindre généralisé malgré sa présence dans l'arbre t'. Cette information commune à ces exemples a donc été perdue au cours de la généralisation. Il convient de s'interroger sur l'impact de cette perte d'information lors d'un apprentissage sur des données réelles.

5.2.2 Moindre généralisation et Réduction

Nous proposons maintenant un algorithme efficace pour le calcul du moindre généralisé de clauses de ce langage. Rappelons qu'un moindre généralisé de clauses se traduit par une généralisation deux à deux d'atomes ayant un même symbole de prédicat (voir chapitre 4, sous-section 4.2.2). Or, chaque clause de $\mathcal{L}_{pc}^{T_\Sigma \cup V}$ a pour chaque symbole de prédicat au plus un atome dans une clause. Un littéral de la première clause ne peut donc être généralisé avec au plus un littéral de la seconde clause. Il est alors possible de tirer profit de ce constat. C'est ce que fait l'algorithme 6 correspondant au calcul de moindre généralisé destiné à ces clauses. La complexité de cet algorithme est en $\mathcal{O}(\min(|C|, |D|))$ étapes d'unification, c'est-à-dire linéaire dans la taille des clauses C et D à généraliser. La profondeur des termes d'une clause de $\mathcal{L}_{pc}^{T_\Sigma \cup V}$ et l'arité de ses prédicats sont bornées par une constante. Ainsi, le nombre d'étapes d'unification correspond au nombre minimum de littéraux que contiennent les clauses C ou D.

Algorithme 6 Moindre généralisé de deux clauses SOP.

function $\mathsf{lgg}_{\mathsf{SOP}}(C, D)$
1: $G \leftarrow \emptyset$;
2: $\Theta \leftarrow \emptyset$;
3: **for** $R(t) \in C,\ R(s) \in D$ **do**
4: $\quad G \leftarrow G \cup \mathsf{lgg\_atom}(R(t), R(s), \Theta)$;
5: **end for**
6: **return** G ;
end function

Rappelons que la fonction lgg_atom est présentée dans le chapitre 4 à la section 4.2.2. Elle permet de généraliser deux atomes en respectant et en complétant, si nécessaire, l'ensemble Θ de triplets terme/terme/variable construit lors du calcul de ce moindre généralisé. Ainsi, dans cet algorithme les termes autorisés peuvent être des constantes, des variables mais également des termes complexes.

Un algorithme efficace de moindre généralisation pour les clauses de $\mathcal{L}_{pc}^{T_\Sigma \cup V}$ présenté, nous nous interrogeons sur la nécessité de la réduction de la clause obtenue ainsi que sur l'évolution de sa taille en fonction du nombre d'exemples. Ces questions se révèlent être importantes car nous savons que le moindre généralisé de clauses peut contenir des littéraux redondants et souffrir d'une taille exponentielle dans le nombre d'exemples [Buntine 1988]. Lors de la preuve du théorème 16 a été mis en avant l'impossibilité, pour le processus de généralisation, de construire un moindre généralisé contenant plusieurs atomes avec un même symbole de prédicat. La taille d'un moindre généralisé de plusieurs clauses reste donc constante ou diminue mais elle ne peut en aucun cas augmenter. Ce langage de clauses n'a donc pas, contrairement au cas général des clauses, le problème de moindre généralisé exponentiel dans le nombre de clauses. Pour cette même raison, il faut noter que la clause obtenue est réduite. En effet, l'existence d'un littéral redondant impliquerait l'existence d'un autre littéral dans la même clause avec le même symbole de prédicat, ce qui est ici impossible de par la définition de ces clauses.

Les clauses $\mathcal{L}_{pc}^{T_\Sigma \cup V}$ bénéficient donc d'un calcul de moindre généralisé linéaire dans la taille des clauses, d'une absence d'explosion de la taille du moindre généralisé et d'une non-obligation de réduction du moindre généralisé. Ces opérations fortement utilisées lors d'un apprentissage ne pénalisent ainsi pas un apprentissage utilisant ce langage de clauses.

5.2.3 Une θ-subsomption polynomiale

Bénéficiant d'un calcul de moindre généralisé plus efficace que dans la version originale de [Plotkin 1970] dont la complexité est quadratique dans la taille des clauses, nous proposons une formulation d'une θ-subsomption propre au langage $\mathcal{L}_{pc}^{T_{\Sigma\cup V}}$. Celle-ci tire parti des particularités de ce langage qui sont :

1. chaque nœud est représenté par un unique atome,
2. chaque atome a un symbole de prédicat unaire décrivant le chemin partant de la racine à ce nœud et lui étant donc propre,
3. l'unique argument d'un atome est l'étiquette du nœud à partir duquel il a été construit,
4. enfin les termes de cette clause ne peuvent être que des constantes ou des variables.

Cette subsomption est présentée à l'algorithme 7. Contrairement à la θ-subsomption pour des clauses de Horn qui est, dans le pire cas, exponentielle dans la taille des clauses, la complexité en temps de notre algorithme est en $\mathcal{O}(|C|)$ où C est la clause devant subsumer. Rappelons que la profondeur des termes ainsi que l'arité des symboles de prédicat des clauses de $\mathcal{L}_{pc}^{T_{\Sigma\cup V}}$ sont bornées. Ainsi, la taille d'une clause C est son nombre de littéraux. La complexité de cette θ-subsomption dépend donc du nombre de littéraux de la clause subsumante.

Cette complexité est possible grâce à la présence d'un index permettant d'obtenir, en temps constant, le littéral portant un symbole de prédicat donné (e.g. une table de hachage). Il a pour intérêt de tirer profit de l'existence pour un symbole de prédicat donné d'au plus un atome l'utilisant dans une clause.

Algorithme 7 θ-subsomption pour clauses SOP.

function $\mathsf{sub}_{\mathsf{SOP}}(C, D)$
1: $\theta \leftarrow \emptyset$;
2: **for** $R(t) \in C$ **do**
3: **if** $\nexists R(s) \in D$ **such that**
4: $\mathsf{sub\_atom}(R(t), R(s), \theta)$;
5: **then**
6: **return** *false* ;
7: **end for**
8: **return** *true* ;
end function

La θ-subsomption de deux clauses, C et D, de $\mathcal{L}_{pc}^{T_{\Sigma\cup V}}$ consiste alors à s'assurer que, pour chaque atome $p(t_1, \cdots, t_n)$ de C, il existe dans D un atome ayant le même symbole de prédicat p/n. Dès lors, bénéficier d'une table de hachage, ayant pour clés les symboles de prédicat des atomes de D, permet de tester en temps constant la présence d'un littéral ayant un symbole de prédicat donné. Une fois l'existence d'un symbole de prédicat prouvée, on tente d'unifier les atomes provenant des deux clauses et disposant de ce même symbole. Si ce processus est validé pour chaque atome, alors la clause C θ-subsume la clause D. Nous disposons donc pour le langage $\mathcal{L}_{pc}^{T_{\Sigma\cup V}}$ d'un test de subsomption linéaire dans la taille des clauses.

Nous avons volontairement présenté des algorithmes de généralisation et de subsomption utilisant les fonctions génériques $\mathsf{lgg\_atom}$ et $\mathsf{sub\_atom}$ définies pour la moindre généralisation et pour la θ-subsomption de Plotkin [Plotkin 1970]. L'utilisation de ces fonctions nous permet de définir une famille de clauses plus générales contenant le langage $\mathcal{L}_{pc}^{T_{\Sigma\cup V}}$. Nous la définissons dans la section suivante.

5.2.4 La famille $\mathcal{F}_{sop}$

La famille des clauses SOP, pour *Single Occurence Predicate*, constitue une sous-famille des clauses déterminées. On la définit de la manière suivante :

Définition 81 (Single Occurrence Predicates)**.** *Une clause SOP est une clause de Horn dans laquelle chaque symbole de prédicat apparaît au plus une fois. Un langage de clauses $\mathcal{L}$ est SOP si chaque clause de ce langage est SOP. La famille des clauses SOP, notée $\mathcal{F}_{sop}$, est l'ensemble des langages $\mathcal{L}_{sop}$.*

Une clause SOP est déterminée par rapport à n'importe quelle autre clause SOP et cela indépendamment de l'ordre des littéraux. En effet, lorsque l'on compare deux clauses SOP, un littéral de la première clause a, au plus, une correspondance dans la seconde : le littéral utilisant le même symbole de prédicat. Cette propriété nous a permis de proposer une θ-subsomption ainsi qu'un moindre généralisé efficaces et dédiés aux clauses $\mathcal{L}_{pc}^{T_{\Sigma\cup V}}$. Cette même propriété nous autorise à réutiliser ces algorithmes pour n'importe quel langage de $\mathcal{F}_{sop}$. En effet, la limitation pour un symbole de prédicat d'au plus un atome dans chaque clause de chaque langage de $\mathcal{F}_{sop}$ est vérifiée. Cet usage restreint des symboles de prédicat induit la non-nécessité de réduction d'une clause obtenue par moindre généralisation. La famille de clauses $\mathcal{F}_{sop}$ bénéficie donc d'opérations de θ-subsomption et de moindre généralisation efficaces et utilisables lors d'un apprentissage. Cette famille de clauses permet la définition d'autres codages d'arbres dont certains présentés dans la section suivante.

5.2.5 Exemples de langages présents dans $\mathcal{F}_{sop}$

Nous présentons ici deux langages appartenant à la famille $\mathcal{F}_{sop}$. Ils ont pour intérêt de montrer que la famille $\mathcal{F}_{sop}$ n'a pas été établie pour la seule définition du langage $\mathcal{L}_{pc}^{T_{\Sigma\cup V}}$.

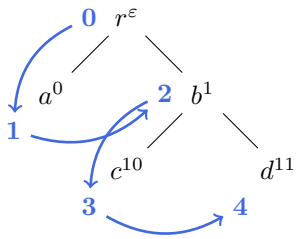
Codage d'un arbre par son parcours en profondeur

Présentons en premier lieu un codage inspiré du parcours en profondeur préfixe d'un arbre. Nous utilisons ce codage afin de numéroter les nœuds d'un arbre de sorte que chaque nœud soit associé à un numéro différent. Le parcours en profondeur préfixe d'un arbre commence par la racine, puis continue par l'exploration du sous-arbre dont la racine est le fils du nœud courant, est non visité et se trouve le plus à gauche. Lorsque tous les fils d'un nœud sont visités, on revient au nœud parent et continuons l'exploration des nœuds et ainsi de suite.

Définition 82 (Numérotation des nœuds par parcours en profondeur préfixe)**.** *Soit un arbre $t = (N_t, root_t, child_t, ns_t, lab_\Sigma^t)$, on note depth-first$(n)$ l'entier spécifiant l'ordre d'exploration du nœud $n \in N_t$ lors du parcours en profondeur préfixe de t :*

$$\textit{depth-first}(n) = \begin{cases} 1 + \sum_{j=0}^{i-1} |\textit{subtree}_t(mj)| + \textit{depth-first}(m) & \textit{si } n = mi \\ & \textit{avec } m \in N_t \textit{ et } i \in \mathbb{N} \\ 0 & \textit{sinon, i.e. } n = \varepsilon \end{cases}$$

Exemple 42. *Soit l'arbre suivant, la numérotation des nœuds induite par le parcours en profondeur préfixe est représentée par les flèches ainsi que les nombres en* **bleu** *:*



On a ainsi : depth-first$(\varepsilon) = 0$, *depth-first*$(0) = 1$, *depth-first*$(1) = 2$, *depth-first*$(10) = 3$ *et depth-first*$(11) = 4$. *La clause représentant cet arbre et appartenant à* $\mathcal{L}_{df}^{T_\Sigma}$ *est la suivante :*

$$C \quad = \quad 0(r),\ 1(a),\ 2(b),\ 3(c),\ 4(d).$$

Chaque nœud d'un arbre est identifié de manière unique par un numéro lors de ce parcours. Nous utilisons cette numérotation pour définir le langage $\mathcal{L}_{df}^{T_\Sigma} \in \mathcal{F}_{sop}$. Chaque clause $C_t \in \mathcal{L}_{df}^{T_\Sigma}$ représente le parcours d'un arbre $t \in T_\Sigma$. Un atome de C_t code un nœud $n \in N_t$. Il a pour prédicat l'entier *depth-first*(n) et pour argument une constante correspondant à l'étiquette de ce nœud. À l'instar du langage $\mathcal{L}_{pc}^{T_\Sigma}$, celui-ci n'est pas clos par moindre généralisation puisqu'il ne prend pas en compte les variables susceptibles d'apparaître dans le moindre généralisé de clauses de $\mathcal{L}_{pc}^{T_\Sigma}$. Dans ce but est défini le langage $\mathcal{L}_{df}^{\Sigma}$.

Définition 83 (Langages $\mathcal{L}_{df}^{\Sigma}$ et $\mathcal{L}_{df}^{T_\Sigma}$)**.** *Le langage* $\mathcal{L}_{df}^{\Sigma}$ *est l'ensemble des clauses SOP basées sur l'utilisation des symboles de prédicat unaires de l'ensemble* $\mathcal{P} = \{p/1 \mid p \in \mathbb{N}\}$ *et des constantes* $a \in \Sigma$ *tel qu'une clause* $C \in \mathcal{L}_{df}^{\Sigma}$ *si et seulement s'il existe pour chaque entier* k *avec* $0 \leqslant k \leqslant |C|$ *un atome de la forme* $k(s)$ *où* s *est une variable ou une constante de* Σ.

Le langage $\mathcal{L}_{df}^{T_\Sigma}$ *est un sous-ensemble de* $\mathcal{L}_{df}^{\Sigma}$ *tel qu'une clause* $C_t \in \mathcal{L}_{df}^{T_\Sigma}$ *si et seulement s'il existe un arbre* $t \in T_\Sigma$ *tel que chaque atome* $p(s) \in C_t$ *si et seulement s'il existe un nœud* $n \in N_t$ *avec* $p =$ *depth-first*(n) *et* $s = lab_\Sigma^t(n)$.

Le langage $\mathcal{L}_{df}^{T_\Sigma}$ est un sous-ensemble $\mathcal{L}_{df}^{\Sigma}$ contenant toutes les clauses fondées de $\mathcal{L}_{df}^{\Sigma}$. Une clause $C_t \in \mathcal{L}_{df}^{T_\Sigma}$ contient $|N_t|$ littéraux. Elle est déconnectée, ce qui n'est pas forcément le cas d'une clause de $\mathcal{L}_{df}^{\Sigma}$. Le langage $\mathcal{L}_{df}^{T_\Sigma}$ ne permet pas de coder un arbre sans perte. Il est impossible de retrouver à partir d'une clause de ce langage l'arbre dont elle est issue. En effet, l'ordre du parcours des nœuds n'indique pas si deux nœuds sont liés par une relation de fraternité ou de paternité. Ce codage, ainsi que les codages précédemment définis, n'utilisent pas de termes complexes. Nous allons donc illustrer une manière de profiter de la présence de termes complexes dans une clause.

Codage comptant les balises

Le codage présenté permet de compter le nombre de nœuds d'un arbre associés à chaque symbole de Σ. Afin de représenter les nombres entiers sous forme de termes, nous nous inspirons des axiomes de Peano et définissons la fonction *peano* prenant en entrée un nombre entier et renvoyant sa transcription sous forme de terme comme suit :

$$peano(n) = \begin{cases} 0 & \text{si } n = 0, \\ s(peano(n-1)) & \text{sinon.} \end{cases}$$

Les termes construits n'utilisent que le symbole de fonction unaire s et une constante 0. La constante 0 représente l'entier 0 et le symbole s l'incrémentation de 1 du nombre courant

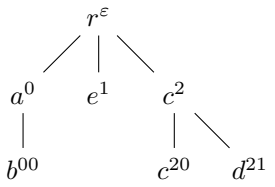
représenté par son argument. Le terme $s^n(0)$ correspond ainsi à l'entier n tandis que le terme $s^n(X)$ représente un entier supérieur ou égal à n. Nous définissons maintenant le langage $\mathcal{L}^{T_\Sigma}_{peano}$ basé sur la construction de ces termes. Une clause $C_t \in \mathcal{L}^{T_\Sigma}_{peano}$ représente le nombre de nœuds associés à chaque étiquette utilisée dans un arbre $t \in T_\Sigma$. Chaque atome de C_t est de la forme $a(n)$ où $a \in \Sigma$ et n est le nombre, sous forme de terme de peano, de nœuds étiquetés par a dans t. Comme les langages définis ultérieurement, ce langage n'est pas clos par moindre généralisation puisqu'il ne prend pas en compte les variables susceptibles d'apparaître dans le moindre généralisé de clauses de $\mathcal{L}^{T_\Sigma}_{peano}$. Le langage $\mathcal{L}^{\Sigma}_{peano}$ est introduit pour y remédier.

Définition 84 (Langages $\mathcal{L}^{\Sigma}_{peano}$ et $\mathcal{L}^{T_\Sigma}_{peano}$)**.** *Le langage $\mathcal{L}^{\Sigma}_{peano}$ est l'ensemble des clauses SOP basées sur l'utilisation de symboles de prédicat $a/1$ avec $a \in \Sigma$, du symbole de fonction $s/1$ et et de la constante 0.*

Le langage $\mathcal{L}^{T_\Sigma}_{peano}$ est un sous-ensemble de clauses de $\mathcal{L}^{\Sigma}_{peano}$ tel qu'une clause $C_t \in \mathcal{L}^{T_\Sigma}_{peano}$ si et seulement s'il existe un arbre $t \in T_\Sigma$ tel qu'un atome $a(s) \in C_t$ si et seulement s'il existe un label $a \in \Sigma$ pour lequel $s = peano(|\{n \in N_t \mid lab_\Sigma(n) = a\}|)$.

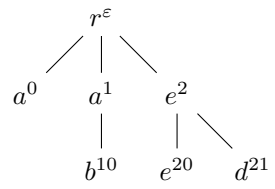
Les clauses de $\mathcal{L}^{T_\Sigma}_{peano}$ est l'ensemble de toutes les clauses fondées de $\mathcal{L}^{\Sigma}_{peano}$. Une clause de $\mathcal{L}^{T_\Sigma}_{peano}$ est déconnectée, ce qui n'est pas toujours le cas des clauses de $\mathcal{L}^{\Sigma}_{peano}$. Elles contiennent néanmoins toutes, au plus, $|\Sigma|$ littéraux.

Exemple 43. *Reprenons les arbres t et t' de l'exemple 40, nous obtenons les clauses suivantes appartenant à $\mathcal{L}^{T_\Sigma}_{peano}$:*



$$C_t = r(s(0)),\ a(s(0)),\ b(s(0))\ c(s(s(0)),\ d(s(0)), e(s(0))$$

$t \in T_\Sigma$



$$C_{t'} = r(s(0)),\ a(s(s(0))),\ b(s(0)),\ d(s(0)),\ e(s(s(0))$$

$t' \in T_\Sigma$

Le moindre généralisé des clauses C_t et $C_{t'}$ appartenant à $\mathcal{L}^{\Sigma}_{peano}$ est le suivant :

$$\mathsf{lgg}(C_t, C_{t'}) = r(s(0)),\ a(s(X)),\ b(s(0)),\ d(s(0)),\ e(s(X))$$

La présence de la variable X dans ce moindre généralisé indique qu'il existe au moins une lettre a et une lettre e. Leur nombre doit être le même dans l'arbre.

Le langage $\mathcal{L}^{\Sigma}_{peano}$ est clos par moindre généralisation contrairement à $\mathcal{L}^{T_\Sigma}_{peano}$. La complexité du test de θ-subsomption ainsi que celle du moindre généralisé sont toujours linéaires dans la taille des clauses manipulées. Cependant, comme les termes complexes sont autorisés dans ce langage, il convient de prendre en compte leurs profondeurs. Ce codage sert à compter le nombre de fois qu'apparait un label dans un arbre. La profondeur d'un terme

est donc limitée par le nombre maximum de nœuds que peut contenir un arbre. Enfin, la fonction lgg_atom assurant la généralisation et reposant sur un mécanisme de remplacement des termes par des variables, la profondeur des termes ne pourra que diminuer ou rester constante au fur et à mesure des généralisations.

Ce codage n'est pas sans perte puisqu'il est impossible de retrouver l'arbre original à partir du nombre de balises présentes dans un arbre. Cependant, dans le cas où deux langages SOP ne partagent aucun symbole de prédicat, il est possible de les combiner afin d'obtenir un nouveau langage SOP. Les clauses de ce nouveau langage sont alors obtenues en concaténant les littéraux des clauses des autres langages. Ainsi, il est possible, par exemple, de combiner les langages $\mathcal{L}^{\Sigma}_{peano}$ et $\mathcal{L}^{T_{\Sigma\cup V}}_{pc}$ afin d'obtenir un codage sans perte. Chaque clause représentant un arbre contiendra alors les littéraux des clauses correspondent à cet arbre et appartiennent à chacun de ces langages. L'enrichissement des codages précédents ainsi que celui de tous codages SOP sont donc facilement réalisables. Nous pouvons, par exemple, obtenir les combinaisons de langages suivantes : *sop*+*peano*, *sop*+*df*, *peano*+*df* et *sop*+*peano*+*df* où *pc* fait référence au codage $\mathcal{L}^{T_\Sigma}_{pc}$, *peano* à $\mathcal{L}^{T_\Sigma}_{peano}$ et *df* à $\mathcal{L}^{T_\Sigma}_{df}$. Ceci permet de constituer de nouveaux codages d'arbres sans perte pouvant disposer d'informations supplémentaires.

5.2.6 Conclusion

La famille des clauses $\mathcal{F}_{sop}$ est donc intéressante pour le traitement des arbres et par conséquent celui des documents XML. Son intérêt réside en :

- la présence d'algorithmes de θ-subsomption et de moindre généralisé linéaires dans la taille des clauses manipulées,
- l'absence de réduction des moindres généralisés produits lors d'un apprentissage, opération généralement obligatoire et coûteuse pour la plupart des langages de clauses,
- l'absence d'une explosion de la taille d'un moindre généralisé dans le nombre de clauses,
- et la possibilité de combiner plusieurs langages permettant la création de codages d'arbres sans perte et contenant des informations tierces.

Tout ceci permet de constituer un framework d'apprentissage efficace, i.e. des opérations de subsomption et de généralisation pouvant être linéaires en temps dans la taille de l'entrée.

Cependant, le langage $\mathcal{L}^{T_\Sigma}_{pc}$ ainsi que les autres codages sans perte issus de la famille $\mathcal{F}_{sop}$ ont généralement l'inconvénient d'être dépendants de la structure des documents étudiés. En effet, la plupart des langages *sop* proposés se base sur une identification unique des nœuds ou des arcs commune à tous les arbres. Cela simplifie l'apprentissage, mais rend la généralisation dépendante de cette identification. Nous nous intéressons, dans la section suivante, au codage d'arbres $\mathcal{L}^{T_\Sigma}_{fcns}$. Celui-ci est indépendant d'une quelconque identification.

5.3 Apprentissage du langage $\mathcal{L}^{T_\Sigma}_{fcns}$

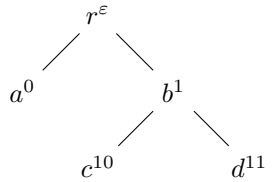
Penchons nous sur l'étude théorique du langage $\mathcal{L}^{T_\Sigma}_{fcns}$. Nous avons vu dans la section 5.1 que ce langage permet la représentation d'arbres. Ainsi, nous désirons l'utiliser pour représenter nos documents XML lors d'une classification. Comme pour le langage $\mathcal{L}^{T_\Sigma}_{pc}$, commençons notre étude théorique par la question de clôture du langage $\mathcal{L}^{T_\Sigma}_{fcns}$. Une fois cette question réglée, éventuellement par la définition d'un langage incluant $\mathcal{L}^{T_\Sigma}_{fcns}$ si la clôture de ce langage par moindre généralisé n'est pas vérifiée, nous proposons un test de θ-subsomption, une réduction ainsi qu'un calcul de moindre généralisé lui étant dédié. Enfin, nous généralisons ce langage et présentons une famille de clauses plus large bénéficiant des opérations

précédemment définies.

5.3.1 Clôture de $\mathcal{L}_{fcns}^{T_\Sigma}$

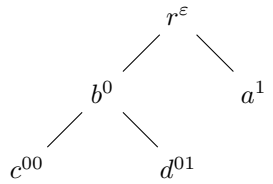
L'exemple ci-dessous invalide l'hypothèse de clôture par moindre généralisation du langage $\mathcal{L}_{fcns}^{T_\Sigma}$.

Exemple 44. *Soit deux arbres t et t' appartenant à T_Σ :*



t

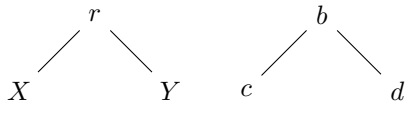
$$C_t = fc(X_\varepsilon, X_0),\ ns(X_0, X_1),\ fc(X_1, X_{10}),\ ns(X_{10}, X_{11}),\ r(X_\varepsilon),\ a(X_0),\ b(X_1),\ c(X_{10}),\ d(X_{11}).$$



t'

$$C_{t'} = fc(X_\varepsilon, X_0),\ fc(X_0, X_{00}),\ ns(X_{00}, X_{01}),\ ns(X_0, X_1),\ r(X_\varepsilon),\ b(X_0),\ a(X_1),\ c(X_{00}),\ d(X_{01}).$$

Le moindre généralisé des C_t et $C_{t'}$ est le suivant :



lgg *de t et t'*

$$\mathsf{lgg}(C_t, C_{t'}) = fc(Y_0, Y_1),\ ns(Y_1, Y_2),\ r(Y_0),\ fc(Z_0, Z_1),\ ns(Z_1, Z_2),\ b(Z_0),\ c(Z_1),\ d(Z_2).$$

Un moindre généralisé de deux clauses de $\mathcal{L}_{fcns}^{T_\Sigma}$ n'est pas obligatoirement une clause de $\mathcal{L}_{fcns}^{T_\Sigma}$ ou une clause connectée. L'absence de constante et de symbole de fonction dans les clauses de $\mathcal{L}_{fcns}^{T_\Sigma}$ permet d'affirmer qu'il ne peut avoir comme termes que des variables. Il est donc, comme les clauses de $\mathcal{L}_{fcns}^{T_\Sigma}$, entièrement variabilisé. Nous devons donc définir un langage de clauses plus général capturant le langage $\mathcal{L}_{fcns}^{T_\Sigma}$ ainsi que ses moindres généralisées.

Désirant bénéficier d'une opération de θ-subsomption efficace, nous proposons de changer la structure d'une clause en la partitionnant en plusieurs ensembles de littéraux. Cette approche fut utilisée par Kietz et Lübbe [Kietz 1994a] afin de définir une θ-subsomption pour les clauses k-locales. Elle est généralement plus efficace que celle de Plotkin (cf. chapitre 4) mais demeure, dans le pire des cas, de complexité exponentielle en temps dans la taille des clauses. Notre approche diffère de la leur et s'appuie sur les connections entre littéraux. Nous avons vu dans la sous-section 5.1.5 la possibilité de réduire le nombre d'appariements des littéraux lors d'une θ-subsomption grâce au *quasi-déterminisme* présent dans les clauses connectées du langage $\mathcal{L}_{fcns}^{T_\Sigma}$. Ainsi, le partionnement que nous proposons a pour enjeu de conserver cette propriété afin de proposer une réécriture plus efficace de la θ-subsomption pour nos clauses.

Avant d'expliciter ce découpage, nous présentons un premier type de clause appelé *multi-clause*. Une multi-clause correspond à un ensemble d'ensembles d'atomes. Chaque ensemble d'atomes est connecté et ne partage pas de variable commune avec les autres ensembles. Elle permet ainsi une représentation intelligente des clauses déconnectées. Nous les définissons plus formellement de la manière suivante :

Définition 85 (Multi-clause)**.** *Une* multi-clause *est un ensemble de clauses connectées par les variables $\mathscr{C} = \{C_1, C_2, \ldots, C_k\}$ tel que les clauses prises deux à deux ne partagent aucune variable. Une multi-clause est interprétée comme l'union de ses composantes, sa taille est la somme des tailles de ses composantes.*

Cette définition n'impose aucune contrainte sur la forme des termes manipulés. Ainsi, une multi-clause peut contenir des termes complexes et des constantes. Notons également qu'une clause connectée peut-être réécrite sous la forme d'une multi-clause ne contenant qu'une composante. Les clauses de $\mathcal{L}_{fcns}^{\Sigma}$ peuvent ainsi être vues comme des multi-clauses entièrement variabilisées. Nous pouvons maintenant définir le langage $\mathcal{L}_{fcns}^{\Sigma}$ basé sur l'utilisation de multi-clauses.

Définition 86 (Langage $\mathcal{L}_{fcns}^{\Sigma}$)**.** *Le langage $\mathcal{L}_{fcns}^{\Sigma}$ contient l'ensemble des multi-clauses sans constante, ni symbole de fonction basées sur l'utilisation des symboles de prédicat suivants :*

- *$fc(X, Y)$ dénote que Y est le premier fils de X,*
- *$ns(X, Y)$ dénote que X et Y sont frères et que Y suit X,*
- *$a(X)$ indique que X porte le label a avec $a \in \Sigma$.*

L'apparition d'une variable au sein d'une clause est sujette aux règles ci-dessous :

1. *elle ne peut être utilisée qu'au plus une fois comme premier argument de fc et au plus une fois comme premier argument de ns ;*
2. *elle ne peut apparaître qu'au plus une fois comme second argument de fc ou ns ;*
3. *elle ne peut être utilisée qu'une seule fois comme argument d'un label $a \in \Sigma$;*
4. *l'existence d'un atome avec un symbole de prédicat ns dans une clause de $\mathcal{L}_{fcns}^{\Sigma}$ implique l'existence de deux atomes $fc(X_1, X_2)$ et $ns(X_2, X_3)$ présents (modulo un renommage de variable près) dans une même composante ;*
5. *pour tout littéral $l_{1<i}$ d'une composante $C_k = (l_1, ..., l_n) \in \mathscr{C}$, il existe au moins un littéral $l_{1 \leqslant j<i}$ appartenant à C_k tel qu'au moins une variable de l_i apparait également dans l_j.*

Nous avons vu au cours de la sous-section 5.1.5 que le langage $\mathcal{L}_{nested}^{T_\Sigma}$ peut, modulo un renommage du symbole de prédicat *next* en *fc*, constituer un sous-ensemble du langage $\mathcal{L}_{fcns}^{T_{\hat{\Sigma}}}$. Le langage $\mathcal{L}_{fcns}^{T_{\hat{\Sigma}}}$ inclus dans $\mathcal{L}_{fcns}^{\hat{\Sigma}}$, nous pouvons donc affirmer que $\mathcal{L}_{nested}^{T_\Sigma} \subseteq \mathcal{L}_{fcns}^{\hat{\Sigma}}$. Vérifions maintenant le théorème suivant portant sur la clôture de ce langage par moindre généralisation.

Théorème 17. *Le langage $\mathcal{L}_{fcns}^{\Sigma}$ est clos par moindre généralisation.*

Preuve. Soit deux clauses C et D appartenant à $\mathcal{L}_{fcns}^{\Sigma}$. Nous voulons montrer que le moindre généralisé de deux clauses de $\mathcal{L}_{fcns}^{\Sigma}$ est une clause de $\mathcal{L}_{fcns}^{\Sigma}$.

Le moindre généralisé issu de C et D est obtenu en généralisant chaque atome $\ell_C = p(X_1, \cdots, X_n)$ de la clause C avec chaque atome $\ell_D = p(Y_1, \cdots, Y_n)$ de la clause D ayant les mêmes symboles de prédicats.

Nous savons par les conditions 1 et 2 de la définition 86 qu'une variable X ne peut apparaître, au plus, qu'une fois comme premier argument d'un atome dont le prédicat est *fc* ou *ns* et une fois comme second argument d'un atome avec pour prédicat *fc* ou *ns*. La condition 3 impose également qu'une variable ne soit utilisée, au plus, qu'une fois comme argument d'un label $a \in \Sigma$. Ainsi, une variable Z d'un triplet $X/Y/Z$ de l'ensemble Θ des triplets terme/terme/variable construit lors de la moindre généralisation ne peut apparaître, au plus, qu'une fois à une position donnée dans un atome dont le prédicat est *fc*, *ns* ou un label $a \in \Sigma$. Le moindre généralisé satisfait donc les conditions 1, 2 et 3 de la définition 86.

Nous nous intéressons maintenant à la condition 4. Le langage $\mathcal{L}^{\Sigma}_{fcns}$ impose que chaque clause possédant un atome dont le prédicat est *ns* contienne également deux atomes : $fc(X, Y)$ et $ns(Y, Z)$. Nous savons que l'existence du prédicat *ns* est liée à celle de *fc* dans le codage *fcns*. Ainsi, le moindre généralisé de clauses possédant le prédicat *ns* contient systématiquement les atomes $fc(X, Y)$ et $ns(Y, Z)$. Si l'une des clauses utilisées pour construire un moindre généralisé ne contient pas de symbole *ns*, alors aucun atome n'utilisera ce symbole dans ce moindre généralisé. Le point 4 est donc bien vérifié.

Enfin, la condition 5 est implicite à la définition de multi-clause. En effet, le moindre généralisé obtenu peut être découpé sous la forme d'une multi-clause dont les composantes sont connectées. Ainsi, la condition 5 est automatiquement vérifiée.

Le moindre généralisé de clauses de $\mathcal{L}^{\Sigma}_{fcns}$ appartient bien à $\mathcal{L}^{\Sigma}_{fcns}$. On en conclut donc que le langage $\mathcal{L}^{\Sigma}_{fcns}$ est clos par moindre généralisation. □

Disposant d'un langage clos par moindre généralisation, nous établissons formellement par la suite la propriété de quasi-déterminisme repérée lors de la pré-sélection de ce codage et proposons ensuite des opérations de subsomption et de généralisation adaptées à celui-ci.

5.3.2 Quasi-déterminisme

Lors de la sélection du langage $\mathcal{L}^{T_\Sigma}_{fcns}$, nous avons vu que l'appariement au cours d'un test de θ-subsomption du premier littéral d'une de ses clauses limite à, au plus, un appariement possible les littéraux restants. Cet appariement détermine ainsi celui des littéraux restants et est donc le seul à nécessiter une énumération. Ce degré de liberté autorisé pour le premier littéral et le déterminisme des littéraux suivants traduisent la notion de *quasi-déterminisme*. Donnons en maintenant une définition formelle.

Définition 87 (Quasi-déterminisme)**.** *Une clause $C = (l_1, \cdots, l_n)$ est quasi-déterminée (QD) par rapport à une clause D si et seulement si pour toute substitution θ telle que $\{l_1\}\theta \subseteq D$, la clause $(l_2, \cdots, l_n)\theta$ est déterminée par rapport à D. Une clause C est QD par rapport à un langage $\mathcal{L}$ si et seulement si elle est QD vis-à-vis de toute clause de $\mathcal{L}$. Un langage $\mathcal{L}$ est QD si chaque clause de $\mathcal{L}$ est QD par rapport à chaque clause $\mathcal{L}$. On note $\mathcal{F}_{\mathsf{QD}}$ la famille des langages de clauses QD.*

Comme les clauses déterminées, une clause quasi-déterminée est une séquence de littéraux. Cependant, la notion de quasi-déterminisme relaxe les contraintes présentes dans la définition de clause déterminée. Ainsi, contrairement aux clauses déterminées, le premier littéral d'une clause quasi-déterminée peut faire l'objet de choix multiples. Une fois ce choix fait, le reste de la clause est déterminé. La famille des langages quasi-determinés englobe donc celle des langages déterminés.

Lors de la sélection des codages, le langage $\mathcal{L}^{T_\Sigma}_{fcns}$ a été mis en avant par la présence du quasi-déterminisme dans ces clauses. Cependant, les clauses de ce langage sont connectées, propriété qui n'est pas toujours garantie pour les clauses du langage $\mathcal{L}^{\Sigma}_{fcns}$. En effet, le langage $\mathcal{L}^{\Sigma}_{fcns}$ contenant des multi-clauses dont les composantes ne partagent pas de variable,

le quasi-déterminisme n'est donc plus assuré pour l'ensemble d'une clause mais demeure vérifié pour chaque composante connectée. Ainsi, nous prouvons avec le lemme suivant que chaque composante d'une clause de $\mathcal{L}^{\Sigma}_{fcns}$ est quasi-déterminée par rapport à n'importe qu'elle autre composante d'une clause de $\mathcal{L}^{\Sigma}_{fcns}$.

Lemme 5. *Chaque composante d'une clause de $\mathcal{L}^{\Sigma}_{fcns}$ est réduite et est quasi-déterminée par rapport à n'importe quelle compostante d'une clause de $\mathcal{L}^{\Sigma}_{fcns}$.*

Preuve. Soit C_k et $D_{k'}$ deux composantes de clauses $\mathcal{L}^{\Sigma}_{fcns}$. Ces composantes sont connectées telles que pour tout littéral $l_{1<i} \in C_k$, il existe au moins un littéral $l_{0\leqslant j<i} \in C_k$ contenant l'une des variables de l_i. Ces composantes appartiennent à des multi-clauses et sont donc connectées et indépendantes.

De ce fait, s'il existe une substitution θ_1 t.q. $l_1\theta_1 \subseteq D_{k'}$ alors il existe au moins un argument de $l_2 \in C_k$ présent dans l_1 dont la valeur est fixée par θ_1. Or, grâce aux conditions 1, 2 et 3 de la définition 86, l'association d'un argument t_i a un symbole de prédicat p/n, pouvant être $fc/2$, $ns/2$ ou un label $a/1$, implique l'existence d'au plus un atome $p(t_1, \cdots, t_i, \cdots, t_n)$ dans une clause $\mathcal{L}^{\Sigma}_{fcns}$. Il en est ainsi de même pour $D_{k'}$. On a donc, au plus, une substitution θ_2 telle que $l_2\theta_1\theta_2 \subset D_{k'}$.

Par induction, il existe alors, au plus, une substitution θ_i telle que $l_i\theta_1 \cdots \theta_i \subseteq D_{k'}$. La composante C_k est donc QD par rapport à n'importe quelle composante $D_{k'}$ d'une clause de $\mathcal{L}^{\Sigma}_{fcns}$.

La composante C_k est QD par rapport à n'importe quelle composante de clauses $\mathcal{L}^{\Sigma}_{fcns}$ et ses littéraux sont connectés puisque que C_k appartient à une multi-clause. Il ne peut alors exister que des littéraux redondants connectés. Or, grâce aux conditions 1, 2 et 3, on sait que, pour deux atomes $p(X_1, \cdots, X_n)$, $p(Y_1, \cdots, Y_n) \in C_k$, s'il existe $i \in [1..n]$ tel que $X_i = Y_i$ alors $p(X_1, \cdots, X_n) = p(Y_1, \cdots, Y_n)$, il n'existe donc pas de littéral $p(Y_1, \cdots, Y_n)$ connecté à un littéral $p(X_1, \cdots, X_n)$ partageant au moins une variable et avec au moins une variable différente. Il n'y a donc pas de littéral redondant dans C_k. La composante C_k est donc une composante QD réduite par rapport à n'importe quelle composante de clause de $\mathcal{L}^{\Sigma}_{fcns}$. □

Cette observation est utile à l'élaboration d'une θ-subsomption destinée aux clauses de $\mathcal{L}^{\Sigma}_{fcns}$. La notion de multi-clause permet de regrouper ensemble les atomes connectés afin de former plusieurs groupes indépendants les uns des autres. Une substitution θ appliquée à une multi-clause peut en effet suivre le même partitionnement. La substitution θ est alors l'union de plusieurs substitutions $\theta_1, \cdots, \theta_n$ indépendantes les unes des autres et s'appliquant chacune à des groupes d'atomes de la multi-clause. Suivant ce raisonnement, une θ-subsomption pour une clause de $\mathcal{L}^{\Sigma}_{fcns}$ se résume à plusieurs tests de θ-subsomption entre des composantes quasi-déterminées. Nous proposons ainsi l'algorithme 8 permettant de tester si une clause quasi-déterminée C θ-subsume une clause quasi-déterminée D.

La subsomption d'une clause QD se déroule en deux étapes. Elle commence par la construction d'une substitution permettant au premier littéral de la clause C de subsumer l'un des littéraux de D. L'algorithme cherche ensuite à étendre cette substitution au reste de la clause à l'aide de l'algorithme standard de subsomption de clauses déterminées (L'algorithme $\mathsf{sub_D}$ se trouve dans le chapitre 4, section 4.3.1). Si cette seconde étape échoue, un nouvel appariement est recherché pour le premier littéral afin d'obtenir une nouvelle première substitution. L'algorithme a donc une complexité en temps de $\mathcal{O}(|C| \times |D|)$ et est quadratique dans la taille des clauses considérées. Remarquons que la complexité d'un test de θ-subsomption entre clauses QD est supérieure à celle entre clauses SOP, la première est quadratique et la seconde linéaire dans la taille des clauses. Cela était prévisible

Algorithme 8 θ-subsumption pour clauses QD.

function $\mathsf{sub}_{\mathsf{QD}}(C, D)$
1: **if** $C = \varnothing$ **then**
2: **return** *true* ;
3: $\ell \leftarrow \mathsf{first}(C)$;
4: **for** $\ell' \in D$ **do**
5: $\theta \leftarrow \varnothing$;
6: **if** $\mathsf{sub\_atom}(\ell, \ell', \theta)$ **and**
7: $\mathsf{sub}_{\mathsf{D}}(C \setminus \{\ell\}, D, \theta)$
8: **then**
9: **return** *true* ;
10: **return** *false* ;
end function

puisque les clauses SOP constituent une sous famille des clauses déterminées et les clauses quasi-déterminées incluent la famille des clauses déterminées.

L'opération de θ-subsomption pour les clauses QD est intéressante de par sa complexité. Cependant, il est possible pour d'autres problèmes d'avoir besoin d'un degré de liberté plus élevé. Afin de le permettre, il est possible de généraliser la définition des clauses quasi-déterminées en autorisant les k premiers littéraux d'une clause à disposer de plusieurs substitutions. Ceci a pour effet d'augmenter la complexité de la θ-subsomption. Ainsi, si la contrainte de déterminisme est supprimée pour les k premiers littéraux d'un clause quasi-déterminée, la complexité du test de subsomption est alors en $\mathcal{O}(|C| \times |D|^k)$.

5.3.3 Multi-Quasi-Déterminisme

Nous disposons grâce à la section précédente d'une θ-subsomption efficace pour les clauses QD ainsi que pour les composantes d'une clause du langage $\mathcal{L}^{\Sigma}_{fcns}$. Cependant, le langage $\mathcal{L}^{\Sigma}_{fcns}$ permet la représentation de multi-clauses à plusieurs composantes. Toute clause de $\mathcal{L}^{\Sigma}_{fcns}$ n'est donc pas quasi-déterminée par rapport à toute autre clause de $\mathcal{L}^{\Sigma}_{fcns}$. Or, chaque composante d'une clause de $\mathcal{L}^{\Sigma}_{fcns}$ est QD par rapport à n'importe quelle autre composante d'une clause de ce même langage. Nous proposons donc d'étendre le quasi-déterminisme à la notion de multi-clause. La définition de *clause multi-quasi-déterminée* allie ainsi la décomposition sous forme de composantes d'une multi-clause au quasi-déterminisme des clauses QD.

Définition 88 (Multi-Quasi-Déterminisme)**.** *Une clause $\mathscr{C} = \{C_1, C_2, \ldots, C_k\}$ est multi-quasi-déterminée (*multi-quasi-determinate *en anglais), notée MQD, par rapport à une clause D si et seulement si chaque composante C_i de $\mathscr{C}$ est une clause QD maximale par rapport à D, c'est-à-dire que $C_i \cup \{\ell\}$ n'est pas QD quelle que soit $\ell \in C_{j \neq i}$. Un langage $\mathcal{L}$ est MQD si chaque clause de ce langage est MQD vis-à-vis de toute clause de ce langage. On note $\mathcal{F}_{\mathsf{MQD}}$ la famille des langages de clauses MQD.*

Le lemme suivant est une conséquence directe de l'application du lemme 5 et de la définition 88.

Lemme 6. *Le langage $\mathcal{L}^{\Sigma}_{fcns}$ est MQD.*

Preuve. Nous savons par le lemme 5 que chaque composante d'une clause de $\mathcal{L}^{\Sigma}_{fcns}$ est une clause QD réduite par rapport à toute composante d'une clause $\mathcal{L}^{\Sigma}_{fcns}$. Selon la définition 88,

une multi-clause de $\mathcal{L}_{fcns}^{\Sigma}$ constituée de plusieurs composantes QD indépendantes et réduites est ainsi MQD par rapport à n'importe quelle clause de $\mathcal{L}_{fcns}^{\Sigma}$. Le langage $\mathcal{L}_{fcns}^{\Sigma}$ est donc MQD. □

On peut illustrer cette définition en reprenant l'exemple précédent.

Exemple 45. *Considérons les deux clauses suivantes du langage $\mathcal{L}_{fcns}^{\Sigma}$:*

$$\begin{array}{lcl} C & = & fc(N_1, N_2), ns(N_2, N_5), fc(N_2, N_3), fc(N_3, N_4), td(N_4). \\ D & = & fc(M_1, M_2), ns(M_2, M_3), td(M_2). \end{array}$$

Le moindre généralisé de C et D est la clause :

$$fc(X_1, X_2), ns(X_2, X_7), fc(X_5, X_6), td(X_6), fc(X_3, X_4).$$

avec $\Theta = \{N_1/M_1/X_1, N_2/M_2/X_2, N_2/M_1/X_3, N_3/M_2/X_4, N_3/M_1/X_5, N_4/M_2/X_6, N_5/M_3/X_7\}$ et a pour forme réduite :

$$fc(X_1, X_2), ns(X_2, X_7), fc(X_5, X_6), td(X_6).$$

Les composantes de la multi-clause MQD construite à partir de la clause E sont :

$$\begin{array}{lcl} E_1 & = & fc(X_1, X_2), ns(X_2, X_7). \\ E_2 & = & fc(X_5, X_6), td(X_6). \\ E_3 & = & fc(X_3, X_4). \end{array}$$

telle que $E = \{E_1, E_2, E_3\}$. On remarque également que E_1, E_2 et E_3 sont QD *vis-à-vis des C et D. De ce fait, E est bien MQD par rapport à C et à D.*

Nous avons vu précédemment que la subsomption entre deux clauses peut être, lorsque ces clauses sont quasi-déterminées, testée de manière quadratique. Les clauses MQD sont une extension des QD, nous nous attendons à ce qu'elles bénéficient également d'un test de subsomption efficace basé sur l'utilisation de la θ-subsomption entre clauses QD. Nous proposons ainsi le test de θ-subsumption entre deux clauses MQD $\mathscr{C}$ et $\mathscr{D}$ précisé par l'algorithme 9 ($\mathsf{sub}_{\mathsf{MQD}}$).

Algorithme 9 θ-subsumption pour clauses MQD.

```
function sub_MQD(𝒞, 𝒟)
1: for C ∈ 𝒞 do
2:    for D ∈ 𝒟 do
3:       if sub_QD(C, D) then
4:          continue first loop;
5:    end for
6:    return false;
7: end for
8: return true;
end function
```

La θ-subsomption entre clauses MQD s'appuie principalement sur le fait que les composantes d'une MQD ne partagent pas de variable. Elles sont donc indépendantes du point de vue de la θ-subsomption. La méthode consiste alors à utiliser la subsomption entre QD définie par l'algorithme $\mathsf{sub}_{\mathsf{QD}}$ pour trouver des correspondances indépendantes entre les composantes de $\mathscr{C}$ et $\mathscr{D}$. La complexité de la θ-subsomption dépendra donc du nombre de composantes de $\mathscr{C}$ ainsi que de la complexité de l'algorithme $\mathsf{sub}_{\mathsf{QD}}$. La taille d'une clause

MQD $\mathscr{C} = \{C_1, \ldots, C_n\}$ est la somme des tailles de ses composants, i.e. $|\mathscr{C}| = |C_1| + \ldots + |C_n|$ et la subsomption entre QD est quadratique dans la taille des clauses manipulées, la complexité de $\mathsf{sub}_{\mathsf{MQD}}$ est donc en $\mathcal{O}(\sum_{C \in \mathscr{C}} \sum_{D \in \mathscr{D}} |C| \times |D|)$ et, par suite, en $\mathcal{O}(|\mathscr{C}| \times |\mathscr{D}|)$ (quadratique).

Avant de continuer l'étude des clauses quasi-déterminées et multi-quasi-déterminées, résumons les résultats obtenus pour la θ-subsomption de ces clauses à l'aide du théorème suivant :

Théorème 18. *Le test de θ-subsomption entre deux clauses quasi-déterminées C et de D est effectué en temps $\mathcal{O}(|C| \times |D|)$ à l'aide de l'algorithme 8. Le test de θ-subsomption entre deux clauses multi-quasi-déterminées $\mathscr{C}$ et $\mathscr{D}$ est effectué en temps $\mathcal{O}(|\mathscr{C}| \times |\mathscr{D}|)$ à l'aide de l'algorithme 9.*

La définition de clause multi-quasi-déterminée utilise la notion de clause QD maximale. Cette notion induit la répartition des littéraux dans chacune des composantes de sorte à préserver le quasi-déterminisme des composantes. Nous nous intéressons donc dans la section suivante au partionnement des littéraux d'une clause de $\mathcal{L}^{\Sigma}_{fcns}$.

5.3.4 Décomposition d'une clause de $\mathcal{L}^{\Sigma}_{fcns}$ en multi-clause

Afin de décomposer une clause de $\mathcal{L}^{\Sigma}_{fcns}$ en une multi-clause, nous proposons l'algorithme 10 (split). Cet algorithme découpe un ensemble de littéraux, pouvant être une clause, en composantes connectées et indépendantes les unes des autres. L'ensemble de ces composantes forme alors la multi-clause correspondant à l'ensemble de littéraux donné en entrée de l'algorithme.

Algorithme 10 Fonction split de partitionnement d'un ensemble de littéraux en une multi-clause.

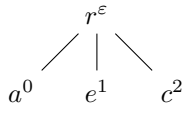
function split(G)
1: $\mathscr{C} \leftarrow \varnothing$;
2: **while** $G \neq \varnothing$ **do**
3: **choose** $\ell \in G$;
4: $G \leftarrow G \setminus \{\ell\}$;
5: $C \leftarrow \{\ell\}$;
6: $F \leftarrow \{\ell\}$;
7: **while** $F \neq \varnothing$ **do**
8: **choose** $R(t_1, \ldots, t_n) \in F$;
9: $F \leftarrow F \setminus \{R(t_1, \ldots, t_n)\}$;
10: **for** $P(s_1, \ldots, s_m) \in G$ **do**
11: **if** $\exists i, j\ t_i = s_j$ **and** $t_i,\ s_i$ are variables **then**
12: $G \leftarrow G \setminus \{P(s_1, \ldots, s_m)\}$;
13: $C \leftarrow C \cup \{P(s_1, \ldots, s_m)\}$;
14: $F \leftarrow F \cup \{P(s_1, \ldots, s_m)\}$;
15: **end for**
16: **end while**
17: $\mathscr{C} \leftarrow \mathscr{C} \cup \{C\}$;
18: **end while**
19: **return** $\mathscr{C}$;
end function

L'algorithme découpe une clause G en une multi-clause $\mathscr{C}$. Le découpage est effectué en regroupant les atomes connectés ensemble. On choisit alors un atome qui est ajouté à la séquence d'atomes C présent à la ligne 5. On recherche ensuite dans G les littéraux partageant des variables avec C et les ajoutons au fur et à mesure à C en fin de séquence. On construit ainsi une séquence d'atomes connectés. Afin d'utiliser la fonction split pour les clauses de $\mathcal{L}_{fcns}^{\Sigma}$, il est nécessaire pour les séquences d'une clause de $\mathcal{L}_{fcns}^{\Sigma}$ de respecter la dernière propriété des clauses de $\mathcal{L}_{fcns}^{\Sigma}$ (voir définition 86). Cette propriété impose que tout atome $l_{0<i} \in C$ partage au moins une variable avec un atome $l_{i<j} \in C$. Ce point respecté par les séquences ainsi produites, l'algorithme split peut donc être utilisé dans le but de découper le moindre généralisé de clauses de $\mathcal{L}_{fcns}^{T_\Sigma}$ et $\mathcal{L}_{fcns}^{\Sigma}$.

Bien qu'elle utilise trois boucles, cette fonction a une complexité en $\mathcal{O}(|G|^2)$ (quadratique dans la taille de l'entrée). Ceci se justifie par le fait que la deuxième boucle parcourt F qui sert de frontière aux éléments de l'ensemble G. Ainsi, tout élément ajouté à F est ensuite supprimé de G, chaque élément est alors ajouté à F exactement une seule fois.

Le langage $\mathcal{L}_{fcns}^{\Sigma}$ bénéficie grâce à l'algorithme split d'une méthode permettant d'obtenir à partir d'un ensemble de littéraux respectant les contraintes du langage $\mathcal{L}_{fcns}^{\Sigma}$ un ensemble de composantes formant une clause multi-quasi-déterminée. Malheureusement, cet algorithme de décomposition n'est pas valable pour tout langage MQD. En effet, les clauses MQD n'imposent aucune contrainte sur les termes présents. Ainsi, l'utilisation de l'algorithme split sur des clauses contenant des atomes fondés revient à créer une composante pour chaque atome. Une telle décomposition sur des langages se révélant être MQD peut causer la perte du multi-quasi-déterminisme. Illustrons ce point avec l'exemple suivant :

Exemple 46. *Soit le langage $\mathcal{L}$ correspondant au langage $\mathcal{L}_{fcns}^{T_\Sigma}$ où chaque variable d'une clause a été remplacée par une constante propre à la clause et à la variable. Soit un arbre $t \in T_\Sigma$, la clause de $\mathcal{L}$ qui le représente est la suivante :*



$t \in T_\Sigma$

$$C_t = \{\ fc(c_\varepsilon, c_0),\ ns(c_1, c_2),\ ns(c_2, c_3),\ r(c_\varepsilon),\ a(c_0),\ e(c_1),\ c(c_1)\}.$$

Les constantes d'une clause de $\mathcal{L}$ ne sont utilisées que dans cette clause, ainsi le moindre généralisé de clauses de $\mathcal{L}$ est entièrement variabilisé. Soit une clause $D = \{fc(X, Y), ns(Y, Z)\}$. Il est facile de voir que D est QD par rapport à C. L'application sur les clauses C et D de l'algorithme split *nous donne les multi-clauses suivantes :*

$$\begin{aligned} \mathscr{C} &= \{\{fc(c_\varepsilon, c_0)\},\ \{ns(c_1, c_2)\},\ \{ns(c_2, c_3)\}, \{r(c_\varepsilon)\},\ \{a(c_0)\},\ \{e(c_1)\},\ \{c(c_1)\}\} \\ \mathscr{D} &= \{\{fc(X, Y), ns(Y, Z)\}\} \end{aligned}$$

La multi-clause $\mathscr{D}$ ne θ-subsume pas la multi-clause $\mathscr{C}$. La décomposition effectuée par l'algorithme split *n'est donc pas adaptée à ce langage.*

Comme nous le montre cet exemple, il convient de définir l'algorithme de décomposition adapté à chaque langage MQD. Seuls les langages de clauses présentés précédemment nous intéresse et comme la définition d'un algorithme générique adapté à tous langages risque de ne pas être optimale, nous laissons ce travail aux personnes intéressés par les clauses multi-quasi-déterminées afin de leur permettre de définir un algorithme propre au langage qu'ils définiront. Nous continuons notre étude et proposons une réécriture du moindre généralisé pour les clauses $\mathcal{L}_{fcns}^{\Sigma}$.

5.3.5 Généralisation des clauses de $\mathcal{L}^{\Sigma}_{fcns}$

Le calcul du moindre généralisé de deux clauses de $\mathcal{L}^{\Sigma}_{fcns}$ est similaire à celui de deux multi-clauses entièrement variabilisées. Ainsi, nous présentons un algorithme calculant le moindre généralisé de deux multi-clauses entièrement variabilisées. Intuitivement, le calcul de lgg entre deux multi-clauses entièrement variabilisées peut se réaliser en généralisant deux à deux les composantes de ces clauses. Cette approche est permise par l'indépendance des composantes au sein d'une même clause vis-à-vis de leurs variables. Nous proposons ainsi de réaliser le calcul du moindre généralisé de deux multi-clauses à l'aide de l'algorithme 11. Cet algorithme construit le moindre généralisé de chaque composante de la première multi-clause avec chaque composante de la seconde. Le moindre généralisé de deux composantes n'étant pas forcément connecté et donc de ce fait une composante, il est nécessaire d'appliquer l'algorithme split permettant le découpage d'une clause en une multi-clause. La multi-clause, qui est le moindre généralisé de deux multi-clauses, est alors obtenue en faisant l'union de tous les ensembles de composantes retournés par l'algorithme split.

Algorithme 11 lgg de deux multi-clauses variabilisées.

function $\mathsf{lgg}_{\mathsf{split}}(\mathscr{C}, \mathscr{D})$
1: $\mathscr{G} \leftarrow \varnothing$;
2: **for** $C \in \mathscr{C}$ **do**
3: **for** $D \in \mathscr{D}$ **do**
4: **let** $\mathsf{lgg}_{comp} \leftarrow \mathsf{lgg}(C, D)$;
5: $\mathscr{G} \leftarrow \mathscr{G} \cup \mathsf{split}(\mathsf{lgg}_{comp})$;
6: **end for**
7: **end for**
8: **return** $\mathscr{G}$;
end function

Cet algorithme convient également pour le calcul de lgg de clauses de $\mathcal{L}^{\Sigma}_{fcns}$. En effet, l'algorithme split découpe le moindre généralisé de deux composantes en respectant la contrainte 5 de la définition 86. De plus, le moindre moindre généralisé de deux clauses de $\mathcal{L}^{\Sigma}_{fcns}$ respecte les contraintes 1, 2, 3 et 4 de la définition 86. Ainsi, la multi-clause obtenue par l'algorithme 11 à partir de deux clauses de $\mathcal{L}^{\Sigma}_{fcns}$ est bien une clause de $\mathcal{L}^{\Sigma}_{fcns}$.

La complexité en temps du calcul de moindre généralisé de deux clauses $\mathscr{C}$ et $\mathscr{D}$ est $\mathcal{O}(|\mathscr{C}| \times |\mathscr{D}|)$. Elle est donc quadratique comme l'algorithme de moindre généralisation de Plotkin. Cependant, l'algorithme 11 découpe le moindre généralisé en multi-clause, à l'aide de l'algorithme split, dont la complexité en temps est $\mathcal{O}(|\mathscr{C}|^2 \times |\mathscr{D}|^2)$. Ainsi, la complexité en temps de l'algorithme 11 est $\mathcal{O}(|\mathscr{C}|^2 \times |\mathscr{D}|^2)$.

Le moindre généralisé de deux clauses nécessite généralement une réduction. Intéressons nous maintenant à cette question.

5.3.6 La réduction polynomiale

Grâce à la définition du langage $\mathcal{L}^{\Sigma}_{fcns}$ et au lemme 5, nous savons qu'une clause de $\mathcal{L}^{\Sigma}_{fcns}$ ne possède que des composantes réduites. Ce lemme ne permet cependant pas d'affirmer que la clause obtenue après moindre généralisation est forcément réduite. L'existence de littéraux redondants dans un moindre généralisé de clauses de $\mathcal{L}^{\Sigma}_{fcns}$ est d'ailleurs observable à l'exemple 45. La réduction du moindre généralisé obtenu après utilisation de l'algorithme $\mathsf{lgg}_{\mathsf{split}}$ est donc nécessaire.

Exemple 47. *Après application de la fonction* split*, la clause calculée dans l'exemple 45 devient :*

$$[fc(X_1, X_2), ns(X_2, X_7)], [fc(X_5, X_6), td(X_6)], [fc(X_3, X_4)]$$

La troisième clause QD, redondante, peut être supprimée.

Nous proposons de traiter la redondance dans les clauses MQD à l'aide de l'algorithme 12 ($\mathsf{reduce}_{\mathsf{MQD}}$). Ce dernier a l'avantage de manipuler les composantes QD de la clause à réduire. Ceci a pour effet de diminuer le nombre de tests de θ-subsomption nécessaires par rapport à l'algorithme reduce de [Plotkin 1970] qui, lui, opère littéral par littéral.

La réduction d'une clause MQD s'apparente à un test de θ-subsomption entre clauses MQD. En effet, lors d'un test de θ-subsomption entre clauses multi-quasi-déteminées $\mathscr{C} = \{C_1, \cdots, C_n\}$ et $\mathscr{D} = \{D_1, \cdots, D_m\}$, il est nécessaire de tester si chaque composante C_i de la première clause θ-subsume, à l'aide de l'algorithme $\mathsf{sub}_{\mathsf{QD}}$, au moins une composante D_j de $\mathscr{D}$. Dans le cas de la θ-subsumption, si une composante θ-subsume une autre, on passe à la vérification de la composante C_{i+1}. Dans le cas de la réduction, si cela s'avère exact, la composante C_i est alors redondante et doit donc être supprimée afin de passer à la composante C_{i+1}.

La complexité au pire de $\mathsf{reduce}_{\mathsf{MQD}}$ est ainsi en $\mathcal{O}(|\mathscr{G}|^2)$ (quadratique), et est proche de celle de l'algorithme $\mathsf{sub}_{\mathsf{MQD}}$. Cette complexité est bien meilleure que celle de Plotkin, qui rappelons-le, est en $|C|^{\mathcal{O}(|C|)}$ où C est la clause à réduire.

Algorithme 12 Réduction d'une clause MQD.

function $\mathsf{reduce}_{\mathsf{MQD}}(\mathscr{G})$
1: $\mathscr{R} \leftarrow \mathscr{G}$;
2: **for** $C_i \in \mathscr{G}$ **do**
3: **for** $C_j \in \mathscr{G}$ **do**
4: **if** $i \neq j$ and $\mathsf{sub}_{\mathsf{QD}}(C_i, C_j)$ **then**
5: $\mathscr{R} \leftarrow \mathscr{R} \setminus \{C_i\}$;
6: **break** ;
7: **end for**
8: **end for**
9: **return** $\mathscr{R}$;
end function

Nous pouvons maintenant résumer les différents résultats obtenus pour les clauses du langage $\mathcal{L}_{fcns}^{\Sigma}$ ainsi que pour les multi-clauses concernant leur décomposition, leur moindre généralisation et leur réduction à l'aide du théorème suivant :

Théorème 19. *L'algorithme 10 calcule* split *en temps $O(|G|^2)$ pour une clause G sans constante ni terme complexe. L'algorithme 11 calcule le moindre généralisé* $\mathsf{lgg}_{\mathsf{split}}$ *en temps $O(|\mathscr{C}|^2 \times |\mathscr{D}|^2)$ de deux clauses $\mathscr{C}, \mathscr{D} \in \mathcal{L}_{fcns}^{\Sigma}$. L'algorithme 12 réduit une clause multi-quasi-déterminée $\mathscr{G}$ en temps $O(|\mathscr{G}|^2)$.*

Nous venons de proposer, au travers de l'étude du langage $\mathcal{L}_{fcns}^{\Sigma}$, plusieurs familles de clauses multi-clauses, les quasi-déterminées et les multi-quasi-déterminées. Pour les mutli-clauses et les clauses de $\mathcal{L}_{fcns}^{\Sigma}$, un algorithme de décomposition (split) ainsi qu'un calcul de moindre généralisé adaptés ($\mathsf{lgg}_{\mathsf{split}}$) ont été proposés. En complément, ont été présentés des algorithmes de θ-subsomption utilisables pour les clauses quasi-déterminées, multi-quasi-déterminées ainsi que les clauses de $\mathcal{L}_{fcns}^{\Sigma}$. Nous disposons ainsi d'une solution permettant de réaliser un apprentissage pour des clauses de $\mathcal{L}_{fcns}^{\Sigma}$.

5.3.7 La famille $\mathcal{F}_{\mathsf{FDP}}$

Nous continuons la démarche de généralisation de nos langages d'arbres afin de permettre d'autres applications et proposons ainsi une sous-famille de $\mathcal{F}_{\mathsf{MQD}}$: la famille FDP. La famille de langages FDP, pour *Functional Depencies Predicate*, a pour but de proposer une famille de langages de clauses réutilisant les algorithmes polynomiaux proposés pour le langage $\mathcal{L}^{\Sigma}_{fcns}$. Elle se base sur le constat que, dans une clause du langage $\mathcal{L}^{\Sigma}_{fcns}$, il n'existe, au plus, qu'un atome usant d'un certain symbole de prédicat et d'un argument à une position donnée. Cette observation garantit le quasi-déterminisme dans une composante d'une multi-clause et le multi-quasi-déterminisme dans une multi-clause. Nous définissons cette famille de manière formelle comme suit :

Définition 89. *Un langage appartient à la famille des langages* FDP *s'il contient un ensemble de multi-clauses sans constante ni symbole de fonction basées sur des symboles de prédicat d'un ensemble $\mathcal{P}$ dont l'utilisation dans une clause* FDP *$\mathscr{C} = \{C_1, \cdots, C_k, \cdots, C_m\}$ est restreinte de la manière suivante :*

1. *soit deux atomes $p(X_1, \cdots, X_n), p(Y_1, \cdots, Y_n) \in C_k$, s'il existe un entier $i \in [1..n]$ tel que $X_i = Y_i$, alors pour tout entier $j \in [1..n]$ on a $X_j = Y_j$.*
2. *pour tout littéral $l_{1<i}$ d'une clause $C_k = (l_1, \cdots, l_n) \in \mathscr{C}$, il existe au moins un littéral $l_{1 \leqslant j<i}$ appartenant à C_k t.q. au moins une variable de l_i apparait également dans l_j.*

On note $\mathcal{F}_{\mathsf{FDP}}$ la famille de langages FDP et $\mathcal{L}^{\mathcal{P}}_{\mathsf{FDP}}$ le langage FDP construit à partir de l'ensemble $\mathcal{P}$ de symboles de prédicat.

Autrement dit, on exige pour chaque triplet constitué d'un symbole de prédicat p/n, d'un terme t et d'une position $i \in [1..n]$ l'existence d'au plus un atome $p(t_1, \cdots, t_i, \cdots, t_n)$ dans une clause FDP avec $t_i = t$. Ce triplet associé à une clause FDP fixe chaque argument de cet atome et ceci de manière unique. L'autre contrainte des FDP ordonne les littéraux d'une clause telle que chaque atome, à l'exception du premier, ait au moins une variable partagée avec un littéral précédent. Cette caractéristique est avantageuse pour la θ-subsomption. En effet, lors d'un test de θ-subsomption, la substitution construite à partir de l'appariement du premier littéral permet, une fois appliquée aux littéraux partageant une variable commune à ce littéral, de limiter à, au plus, une substitution possible ces littéraux. Par induction, les substitutions des littéraux suivants sont, si elles existent, également uniques. On retrouve ainsi le comportement des clauses quasi-déterminées. Cet ordre évite ainsi qu'un atome non connecté avec les littéraux précédents n'apparaisse, cassant le quasi-déterminisme de la clause. Cette intuition est valide si la propagation atteint effectivement tous les littéraux. C'est ce que traduit le lemme suivant qui établit un lien formel avec la notion de quasi-déterminisme :

Lemme 7. *Chaque composante d'une clause d'un langage $\mathcal{L}^{\mathcal{P}}_{\mathsf{FDP}}$ est une clause* QD *réduite par rapport à n'importe quelle composante d'une clause de $\mathcal{L}^{\mathcal{P}}_{\mathsf{FDP}}$.*

Preuve. Soit C_k et $D_{k'}$ deux composantes issues respectivement de deux clauses d'un langage $\mathcal{L}^{\mathcal{P}}_{\mathsf{FDP}}$. Ces composantes appartiennent à des multi-clauses, elles sont donc connectées et indépendantes. Il existe ainsi, pour tout litéral $l_{1<i} \in C_k$, au moins un littéral $l_{0 \leqslant j<i} \in C_k$ contenant l'une des variables de $l_{j<i}$.

De ce fait, s'il existe une substitution θ_1 t.q. $l_1\theta_1 \subseteq D_{k'}$ alors il existe au moins un argument de $l_2 \in C_k$ présent dans l_1 dont la valeur est fixée par θ_1. Or, on sait que pour un symbole de prédicat p/n associé à un argument t_i, il existe, au plus, un atome $p(t_1, \cdots, t_i, \cdots, t_n)$ dans une clause de $\mathcal{L}^{\mathcal{P}}_{\mathsf{FDP}}$. Il en est de même pour $D_{k'}$. On a donc

au plus une substitution θ_2 t.q. $l_2\theta_1\theta_2 \subset D_{k'}$. Par induction, il existe alors, au plus, une substitution θ_i t.q. $l_i\theta_1 \cdots \theta_i \subseteq D_{k'}$. La composante C_k est donc QD par rapport à n'importe quelle composante $D_{k'}$ d'une clause de $\mathcal{L}^{\mathcal{P}}_{\mathsf{FDP}}$.

Nous prouvons maintenant que cette composante est réduite. La composante C_k est QD par rapport à n'importe quelle composante de clauses de $\mathcal{L}^{\mathcal{P}}_{\mathsf{FDP}}$. Elle appartient à une multi-clause et a ainsi ses littéraux connectés. S'agissant d'une composante FDP, nous savons que, pour deux atomes $p(X_1, \cdots, X_n)$, $p(Y_1, \cdots, Y_n) \in C_k$, s'il existe $i \in [1..n]$ tel que $X_i = Y_i$, alors $p(X_1, \cdots, X_n) = p(Y_1, \cdots, Y_n)$. Il n'existe donc pas de littéral $p(Y_1, \cdots, Y_n)$ connecté à un littéral $p(X_1, \cdots, X_n)$ partageant une variable et dont une autre diffère. Il n'existe alors pas de littéral redondant dans C_k. La composante C_k est donc une composante QD réduite par rapport à n'importe quelle composante de clause de $\mathcal{L}^{\mathcal{P}}_{\mathsf{FDP}}$. □

Lemme 8. *Tout langage $\mathcal{L}^{\mathcal{P}}_{\mathsf{FDP}}$ est MQD.*

Preuve. Grâce au lemme 7, nous savons que toute composante d'une clause de $\mathcal{L}^{\mathcal{P}}_{\mathsf{FDP}}$ est une QD réduite par rapport à n'importe quelle composante de clause de $\mathcal{L}^{\mathcal{P}}_{\mathsf{FDP}}$. Selon la définition 89, une multi-clause $\mathcal{L}^{\mathcal{P}}_{\mathsf{FDP}}$ constituée de plusieurs composantes QD indépendantes et réduites est MQD par rapport à $\mathcal{L}^{\mathcal{P}}_{\mathsf{FDP}}$. Le langage $\mathcal{L}^{\mathcal{P}}_{\mathsf{FDP}}$ est donc MQD. □

La famille $\mathcal{F}_{\mathsf{FDP}}$ contient donc un ensemble de langages MQD. Chaque clause d'un de ces langages est une mutli-clause variabilisée. De ce fait, nous pouvons bénéficier des algorithmes de subsomption $\mathsf{sub}_{\mathsf{QD}}$ et $\mathsf{sub}_{\mathsf{MQD}}$, de l'algorithme de réduction $\mathsf{reduce}_{\mathsf{MQD}}$ et de l'algorithme de partitionnement split. Nous vérifions maintenant que l'algorithme de moindre généralisation $\mathsf{lgg}_{\mathsf{split}}$ est utilisable pour un langage de $\mathcal{F}_{\mathsf{FDP}}$.

Lemme 9. *Tout langage $\mathcal{L}^{\mathcal{P}}_{\mathsf{FDP}}$ de $\mathcal{F}_{\mathsf{FDP}}$ est clos par moindre généralisé en utilisant l'algorithme $\mathsf{lgg}_{\mathsf{split}}$.*

Preuve. Soit deux clauses $\mathscr{C} = \{C_1, \cdots, C_m\}, \mathscr{D} = \{D_1, \cdots, D_n\} \in \mathcal{L}^{\mathcal{P}}_{\mathsf{FDP}}$. Les composantes de ces clauses sont indépendantes du point de vue des variables, on a alors $\mathsf{lgg}_{\mathsf{split}}(\mathscr{C}, \mathscr{D}) = \cup_{i=1}^{n} \cup_{j=1}^{m} \mathsf{split}(\mathsf{lgg}(C_i, D_j))$. La fonction split décompose le moindre généralisé de deux composantes en composantes indépendantes. Chaque composante de $\mathsf{split}(\mathsf{lgg}(C_i, D_j))$ est donc indépendante des autres composantes du moindre généralisé et les littéraux appartenant à une composante sont donc connectés. On a donc bien une multi-clause. La condition 2 de la définition 89 est alors vérifiée.

Vérifions la condition 1. La généralisation des composantes est indépendante pour des clauses du langage $\mathcal{L}^{\mathcal{P}}_{\mathsf{FDP}}$. Ainsi, si le moindre généralisé de deux clauses de $\mathcal{L}^{\mathcal{P}}_{\mathsf{FDP}}$ n'est plus une clause de $\mathcal{L}^{\mathcal{P}}_{\mathsf{FDP}}$, i.e. ne vérifie plus la condition 1, cela signifie alors qu'il existe un littéral redondant dans l'une des composantes de ce moindre généralisé. Ce littéral est connecté à la composante. Il doit donc exister deux littéraux, $p(X_1, \cdots, X_n)$ et $p(Y_1, \cdots, Y_n)$, tels que $X_i = Y_i$. Cela suppose que l'une des composantes généralisées possède deux atomes $p(X_1, \cdots, X_n)$ et $p(Y_1, \cdots, Y_n)$ avec $X_i = Y_i$. Or, ce type de clauses n'est pas une clause de $\mathcal{L}^{\mathcal{P}}_{\mathsf{FDP}}$. Il y a alors contradiction avec l'hypothèse de départ. Il n'existe ainsi pas de littéral redondant dans les composantes de $\mathsf{split}(\mathsf{lgg}(C_i, C'_j))$. Il en est de même pour chaque composante d'un moindre généralisé de deux clauses de $\mathcal{L}^{\mathcal{P}}_{\mathsf{FDP}}$. Le langage $\mathcal{L}^{\mathcal{P}}_{\mathsf{FDP}}$ est clos par moindre généralisé en utilisant l'algorithme $\mathsf{lgg}_{\mathsf{split}}$.

□

Montrons maintenant l'appartenance du langage $\mathcal{L}^{\Sigma}_{fcns}$ à la famille $\mathcal{F}_{\mathsf{FDP}}$.

Lemme 10. $\mathcal{L}^{\Sigma}_{fcns} \in \mathcal{F}_{\mathsf{FDP}}$.

Preuve. Afin de prouver que $\mathcal{L}^{\Sigma}_{fcns} \in \mathcal{F}_{\mathsf{FDP}}$, nous démontrons que les deux conditions présentes dans la définition 89 sont vraies pour ce langage.

Nous commençons par le premier point de la définition 89. en reprenant les points 1, 2 et 3 de la définition 86 afin de montrer que, pour un symbole de prédicat et un argument, il n'existe qu'un seul atome. Le point 1 permet d'assurer que, pour un symbole de prédicat fc ou ns et une variable X, il existe au plus un atome $fc(X, Y_1)$ et un atome $ns(X, Y_2)$. Cependant, cela n'interdit pas l'existence de deux atomes $fc(X, Y), fc(Z, Y)$ ou bien encore $ns(X, Y), ns(Z, Y)$. C'est ce que fait le point 2 de la définition 86 en imposant le fait qu'une variable ne peut être apparaître, au plus, qu'une fois dans un atome comme second argument de fc ou de ns. Enfin, le point 3 de la définition 86 impose qu'une variable ne puisse être utilisée, au plus, qu'une fois pour un label $a \in \Sigma$. Le langage $\mathcal{L}^{\Sigma}_{fcns}$ respecte donc bien le premier point de la définition 89. Le deuxième point de la définition 89 étant similaire à celui de définition 86, on a donc bien $\mathcal{L}^{\Sigma}_{fcns} \in \mathcal{F}_{\mathsf{FDP}}$.

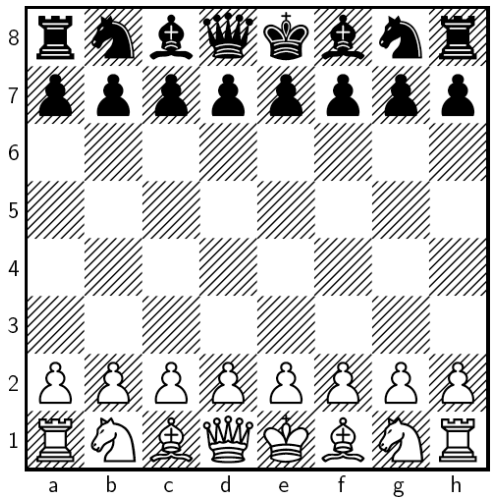
□

Le langage $\mathcal{L}^{\Sigma}_{fcns}$ appartient à la famille FDP, il est ainsi facile de proposer de nouveaux langages FDP basés sur celui-ci. Nous proposons les modifications suivantes :

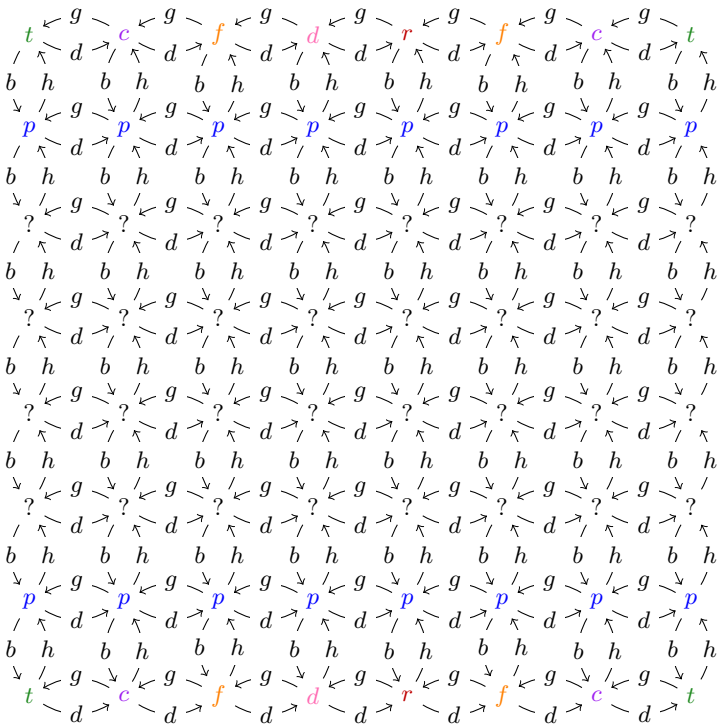
— l'ajout d'un label *root* pour désigner la variable représentant la racine de l'arbre,
— l'ajout un label *leaf* signifiant qu'une variable est une feuille de l'arbre,
— la suppression des labels des nœuds présents dans le codage *fcns*.

Ces modifications constituent les codages suivants : *fcns*+*root*, *fcns*+*leaf*, *fcns*+*root*+*leaf*, *fcns-label*, *fcns-label*+*root*, *fcns-label*+*leaf*, *fcns-label*+*root*+*leaf* où le symbole + désigne l'ajout de labels et - leur suppression.

La famille $\mathcal{F}_{\mathsf{FDP}}$ ne se résume pas uniquement à la représentation d'arbres, nous donnons ci-dessous un autre exemple d'utilisation possible de celle-ci. Soit le langage $\mathcal{L}_{chess} \in \mathcal{F}^{\mathcal{P}}_{\mathsf{FDP}}$ avec $\mathcal{P} = \{h/2, b/2, g/2, d/2\}$ avec h pour haut, b pour bas, g pour gauche et d pour droite ainsi qu'un alphabet $\Sigma = \{r,\ d,\ f,\ c,\ t,\ p\}$ où r est pour roi, d pour dame, f pour fou, c pour cavalier, t pour tour et p pour pion. Ce langage permet la représentation d'une instance d'une partie d'échec. Illustrons cela avec l'échiquier ci-dessous correspondant à un début de partie.



La représentation graphique de la clause de cet échiquier est la suivante :



Dans cette représentation clausale du plateau d'échec, le symbole de ponctuation ? représente une variable. Chaque case du plateau est associée à une unique variable pouvant être étiquetée par un label de Σ. Une case peut posséder un voisin à sa droite et/ou à sa gauche, au dessus et/ou en dessous de lui, c'est ce que représentent les prédicats h, b, g et d. Cette représentation nous montre que la famille de clauses $\mathcal{F}_{\mathsf{FDP}}$ ne se limite pas à la représentation de structures arborescentes. En effet, la figure présentée est assimilable à un graphe aux nœuds et arêtes étiquetés tels que, pour chaque nœud et une étiquette d'arête donnée, il existe au plus une arête entrante et une sortante utilisant cette étiquette pour ce nœud. Ce graphe peut ainsi posséder des cycles. La famille $\mathcal{F}_{\mathsf{FDP}}$ permet donc la représentation de quadrillages, de certains processus fonctionnels ainsi que d'autres objets respectant la contrainte de fonctionnalité décrite ci-dessus.

Comme pour les langages SOP, il est possible de combiner deux langages FDP afin d'obtenir un nouveau langage FDP. Cette affirmation est vraie lorsque les langages FDP combinés ne partagent aucun symbole de prédicat en commun. Des lors, les clauses du langage résultant de la combinaison de plusieurs langages FDP sont obtenues en concaténant les littéraux des clauses des autres langages. L'indépendance de ces langages du fait de l'absence de symbole de prédicat commun permet de conserver leur propriété FDP. L'enrichissement des codages précédents ainsi que tout codage FDP est donc faisable sans difficulté puisqu'aucun nouvel algorithme ne doit être défini.

Nous revenons au langage $\mathcal{L}^{\Sigma}_{fcns}$ et étudions la question d'explosion de la taille du moindre généralisé dans le nombre de clauses pour ce langage.

5.3.8 Explosion du moindre généralisé

Dans le cas des clauses de Horn, le moindre généralisé de clauses peut être de taille exponentielle dans le nombre de clauses généralisées. Cette explosion n'a cependant pas lieu pour tous les langages de clauses comme nous l'avons montré pour le langage SOP (cf.

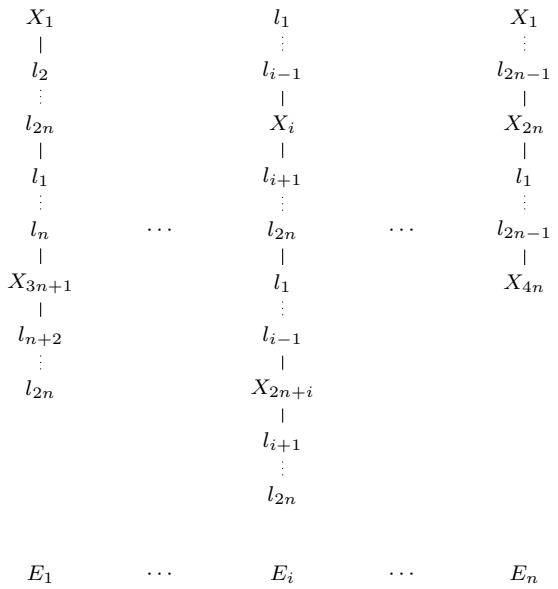
section 5.2.2). Le langage $\mathcal{L}^{\Sigma}_{fcns}$ bénéficiant déjà d'une θ-subsomption ainsi que d'une réduction polynomiale, l'absence d'explosion de la taille du moindre généralisé dans le nombre de clauses serait un avantage supplémentaire lors d'un apprentissage. Ainsi, nous étudions la taille du moindre généralisé obtenu à partir de clauses de $\mathcal{L}^{\Sigma}_{fcns}$.

Explosion du moindre généralisé pour les arbres d'arité borné à 1

Afin d'analyser le comportement d'un moindre généralisé de clause de $\mathcal{L}^{\Sigma}_{fcns}$, nous nous intéressons en premier lieu aux cas des arbres dont l'arité est bornée à 1. Ce type d'arbre, modulo un renommage du symbole de prédicat *fc* par *next*, permet la représentation des mots comme vu en section 5.1.5 avec les nested words du langage $\mathcal{L}^{T_{\Sigma}}_{nested}$. Nous proposons la construction suivante basée sur un ensemble de clauses utilisant le symbole de prédicat *fc* et les prédicats unaires issus d'un alphabet ordonné $\Sigma = \{l_1, \cdots, l_{2n}\}$ avec $|\Sigma| = 2 \times n$:

$$
\begin{array}{rcl}
E & = & \{\ \{\ fc(X_1, X_2),\ \cdots,\ fc(X_{n-1}, X_n), \\
 & & l_1(X_1),\ \cdots,\ l_{i-1}(X_{i-1}),\ l_{i+1}(X_{i+1}),\ \cdots, l_n(X_n), \\
 & & fc(X_n, X_{n+1}),\ \cdots,\ fc(X_{2n-1}, X_{2n}), \\
 & & l_{n+1}(X_{n+1}),\ \cdots,\ l_{2n}(X_{2n}) \\
 & & fc(X_{2n}, X_{2n+1}),\ \cdots,\ fc(X_{3n-1}, X_{3n}), \\
 & & l_1(X_{2n+1}),\ \cdots,\ l_n(X_{3n}), \\
 & & fc(X_{3n}, X_{3n+1}),\ \cdots,\ fc(X_{4n-1}, X_{4n}), \\
 & & l_{n+1}(X_{3n+1}),\ \cdots,\ l_{n+i-1}(X_{3n-1}),\ l_{n+i+1}(X_{3n+1}),\ \cdots,\ l_{2n}(X_{4n})\ \} \\
 & & \mid i : 1 \leqslant i \leqslant n\ \}
\end{array}
$$

Cet ensemble de clauses représente les arbres suivants :



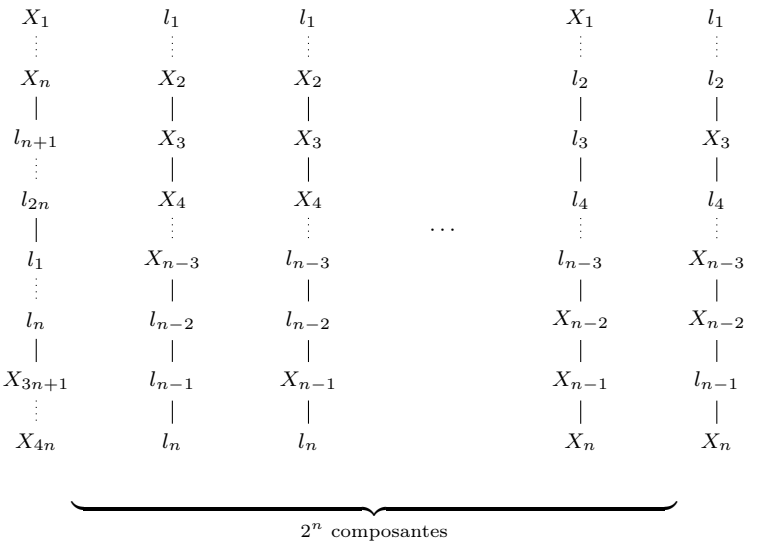
Dans cette représentation arborescente, les pointillés correspondent à des nœuds étiquetés par une lettre de Σ comme indiqué par la construction. Ainsi, l'ensemble E contient n clauses telles que la taille de la $i^{ème}$ clause (avec $1 \leqslant i \leqslant n$) est :

$$|E_i| = 4 \times (2 \times n - 1) + 1$$

et la taille de l'ensemble E, c'est-à-dire de la somme des tailles des n clauses de E, est :

$$||E|| = 4 \times n \times (2 \times n - 1) + 1 = 8 \times n^2 - 4 \times n + 1$$

Après moindre généralisation de cet ensemble, la clause obtenue est la suivante :



La taille du moindre généralisé obtenue est :

$$|\mathsf{lgg}(E)| = 2^n \times (2n-1) + 1 + 2^n \times n$$

La facteur 2^n dans la taille du moindre généralisé permet d'affirmer qu'il a une taille exponentielle dans le nombre de clauses de l'ensemble E. Les arbres utilisés d'arité bornée à 1 sont assimilables à des mots. Nous pouvons donc en déduire que la structure simplifiée des mots ne permet pas théoriquement d'éviter l'explosion combinatoire de la taille du moindre généralisé. Résumons ce résultat à l'aide du théorème suivant :

Théorème 20. *Soit le langage $\mathcal{L}_{fc}^{\Sigma}$, il existe un ensemble de clauses E dont la taille du moindre généralisé $\mathsf{lgg}(E)$ est exponentielle dans le nombre de clauses de cet ensemble.*

Cette construction a nécessité l'usage de labels. Dans le cas où ces derniers sont omis, l'explosion du moindre généralisé de clauses de $\mathcal{L}_{fcns}^{\Sigma}$ représentant des arbres d'arité borné à 1 sans label n'est plus possible. Chaque généralisation est, en effet, réduite à une multi-clause d'une seule composante correspondant au plus grand nombre de littéraux connectés communs à chaque exemple. Ceci s'explique par le mécanisme de réduction des clauses. Un exemple illustre la construction donnée précédemment et se trouve en annexe section A.2. Étendons le résultat portant sur les arbres d'arité bornée à 1 aux nested words avant de passer ensuite au cas général des clauses de $\mathcal{L}_{fcns}^{\Sigma}$ sans label.

Explosion du moindre généralisé pour les nested words

La construction précédente n'est pas utilisable en l'état pour représenter des nested words puisque la contrainte d'imbrication de ces derniers n'est pas respectée. Ainsi, nous l'adaptons et proposons la construction suivante. Soit l'ensemble de clauses E de $\mathcal{L}_{nested}^{T_\Sigma}$ avec $\hat{\Sigma} = \{<l_0>, </l_0>, <l_1>, </l_1>, \cdots, <l_{2n}>, <l_{2n}>, <l'_1>, </l'_1>, \cdots, <l'_{2n}>, <l'_{2n}>\}$:

$$
\begin{aligned}
E \quad = \quad & \{\ \{\ fc(R_0, X_1),\ <l_0>\ (R_0),\\
& fc(X_1, X_2),\ \cdots,\ fc(X_{2n-1}, X_{2n}),\\
& <l_1>\ (X_1),\ </\ l_1>\ (X_2),\ \cdots,\ <l_{i-1}>\ (X_{2i-3}),\ </\ l_{i-1}>\ (X_{2i-2}),\\
& <l'_i>\ (X_{2i-1}),\ </\ l'_i>\ (X_{2i}),\\
& <l_{i+1}>\ (X_{2i+1}),\ </\ l_{i+1}>\ (X_{2i+2}),\ \cdots,\ <l_n>\ (X_{2n-1}),\ </\ l_n>\ (X_{2n}),\\
& fc(X_{2n}, X_{2n+1}),\ \cdots,\ fc(X_{4n-1}, X_{4n}),\\
& <l_{n+1}>\ (X_{2n+1}),\ </\ l_{n+1}>\ (X_{2n+2}),\ \cdots,\ <l_{2n}>\ (X_{4n-1}),\ </\ l_{2n}>\ (X_{4n})\\
& fc(X_{4n}, X_{4n+1}),\ \cdots,\ fc(X_{6n-1}, X_{6n}),\\
& <l_1>\ (X_{4n+1}),\ </\ l_1>\ (X_{4n+2}),\ \cdots,\ <l_n>\ (X_{6n-1}),\ </\ l_n>\ (X_{6n}),\\
& fc(X_{6n}, X_{6n+1}),\ \cdots,\ fc(X_{8n-1}, X_{8n}),\\
& <l_{n+1}>\ (X_{6n+1}),\ </\ l_{n+1}>\ (X_{6n+2}),\ \cdots,\ <l_{n+i-1}>\ (X_{6n+2i-3}),\ </\ l_{n+i-2}>\ (X_{6n+2i-2}),\\
& <l'_{n+i}>\ (X_{6n+2i-1}),\ </\ l'_{n+i}>\ (X_{6n+2i}),\\
& <l_{n+i+1}>\ (X_{6n+2i+1}),\ </\ l_{n+i+1}>\ (X_{6n+2i+2}),\ \cdots,\ <l_{2n}>\ (X_{8n-1}),\ </\ l_{2n}>\ (X_{8n}),\\
& fc(X_{8n}, R_1),\ </\ l_0>\ (R_1),\ \}\\
& \mid i : 1 \leqslant i \leqslant n\ \}
\end{aligned}
$$

correspondant à l'ensemble de mots suivant :

$$
\begin{aligned}
\{\ & <l_0><l'_1></\ l'_1><l_2></\ l_2>\cdots<l_{2n}></\ l_{2n}><l_1></\ l_1>\cdots\\
& \qquad <l_n></\ l_n><l'_{n+1}></\ l'_{n+1}><l_{n+2}></\ l_{n+2}>\cdots<l_{2n}></\ l_{2n}></\ l_0>,\\
& \qquad \cdots,\\
& <l_0><l_1></\ l_1>\cdots<l_{i-1}></\ l_{i-1}><l'_i></\ l'_i><l_{i+1}></\ l_{i+1}>\cdots<l_{2n}></\ l_{2n}><l_1></\ l_1>\cdots\\
& \qquad <l_{n+i-1}></\ l_{n+i-1}><l'_{n+i}></\ l'_{n+i}><l_{n+i+1}></\ l_{n+i+1}>\cdots<l_{2n}></\ l_{2n}></\ l_0>,\\
& \qquad \cdots,\\
& <l_0><l_1></\ l_1>\cdots<l_{n-1}></\ l_{n-1}><l'_n></\ l'_n><l_{n+1}></\ l_{n+1}>\cdots\\
& \qquad <l_{2n}></\ l_{2n}><l_1></\ l_1>\cdots<l_{2n-1}></\ l_{2n-1}><l'_{2n}></\ l'_{2n}></\ l_0>\ \}
\end{aligned}
$$

Le moindre généralisé de cet ensemble est très grand. Ainsi, nous ne donnons que sa taille qui est en $\mathcal{O}(2^{n/2})$. Il est donc possible à partir de clauses de $\mathcal{L}^{T_\Sigma}_{nested}$ de construire un moindre généralisé dont la taille est exponentielle dans le nombre d'exemples.

Explosion du moindre généralisé pour les arbres de $\mathcal{L}^{T_\Sigma}_{fcns}$

Nous avons vu précédemment qu'il est possible d'obtenir un moindre généralisé de taille exponentielle à partir d'un nombre fini de clauses de $\mathcal{L}^{T_\Sigma}_{fcns}$. La construction utilisée repose sur une variation de l'étiquette d'un nœud dans chaque clause. Cela permet d'instaurer un motif commun à toutes les clauses ainsi qu'un autre commun à $n-1$ clauses. Nous proposons ci-dessous une construction sans label mais permettant également une explosion de la taille du moindre généralisé dans le nombre de clauses. Soit l'ensemble E suivant :

$$
\begin{aligned}
& E = \{\ \{\ fc(X_1, X_2), \cdots, fc(X_i, X_{i+1}), \cdots, fc(X_{2n-1}, X_{2n}),\\
& ns(X_1, X_{1,1}),\\
& ns(X_2, X_{2,1}), ns(X_{2,1}, X_{2,2}),\\
& \cdots,\\
& ns(X_{i-1}, X_{i-1,1}), \cdots, ns(X_{i-1,i-2}, X_{i-1,i-1}),\\
& ns(X_{i+1}, X_{i+1,1}), \cdots, ns(X_{i+1,i}, X_{i+1,i+1}),\\
& \cdots,\\
& ns(X_{2n}, X_{2n,1}), \cdots, ns(X_{2n,2n-1}, X_{2n,2n}),\\
& fc(X_{2n+1}, X_{2n+2}), \cdots, fc(X_{2n+i}, X_{2n+i+1}), \cdots, fc(X_{4n-1}, X_{4n}),\\
& ns(X_{2n+1}, X_{2n+1,1}), fc(X_{2n+1,1}, X_{2n+1,2}),\\
& ns(X_{2n+2}, X_{2n+2,1}), ns(X_{2n+2,1}, X_{2n+2,2}),\\
& \cdots,\\
& ns(X_{2n+i-1}, X_{2n+i-1,1}), \cdots, ns(X_{2n+i-1,i-2}, X_{2n+i-1,i-1}),
\end{aligned}
$$

$ns(X_{2n+i+1}, X_{2n+i+1,1}), \cdots, ns(X_{2n+i+1,i}, X_{2n+i+1,i+1}),$
$\cdots,$
$ns(X_{4n}, X_{4n,1}), \cdots, ns(X_{4n,4n-1}, X_{4n,4n}),\ \} \mid i : 1 \leqslant i \leqslant n\ \}$

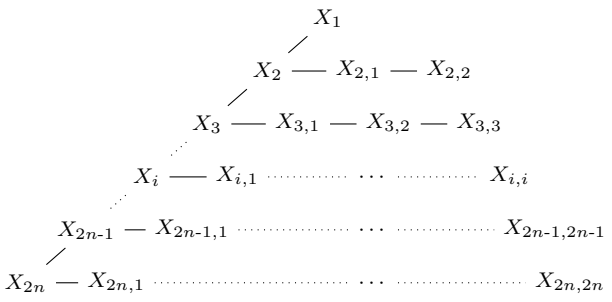
L'ensemble E contient n clauses tel que la taille de la $i^{ème}$ clause (avec $1 \leqslant i \leqslant n$) est :

$$|E_i| = 2 \times (2 \times n - 1) + \sum_{k=1}^{k=i-1} k + \sum_{k=i+1}^{k=2n} k + \sum_{k=1}^{k=n+i-1} k + \sum_{k=n+i+1}^{k=2n} k$$

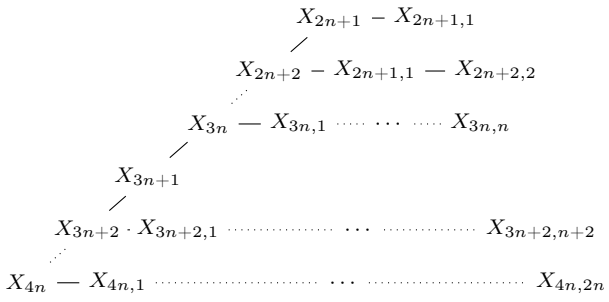
et la taille de l'ensemble E, c'est-à-dire de la somme des tailles des n clauses de E, est :

$$||E|| = \sum_{i=1}^{i=n}(2 \times (2 \times n - 1) + \sum_{k=1}^{k=i-1} k + \sum_{k=i+1}^{k=2n} k + \sum_{k=1}^{k=n+i-1} k + \sum_{k=n+i+1}^{k=2n} k$$

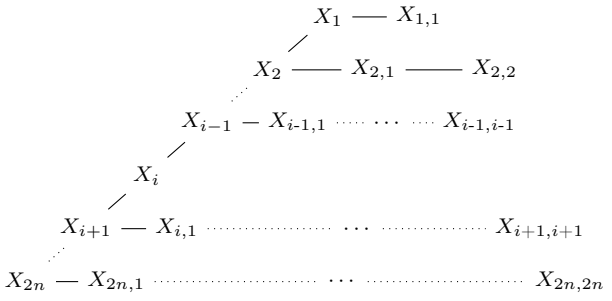
Les sous-arbres suivants représentent certaines clauses présentes dans l'ensemble E. Seuls les liens *fc* et *ns* y sont représentés respectivement par des traits verticaux et des traits horizontaux.



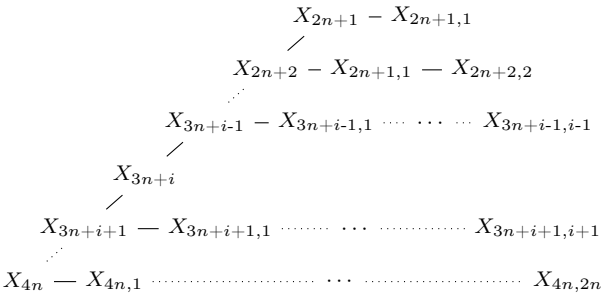
1ière composante de la 1ière clause de E



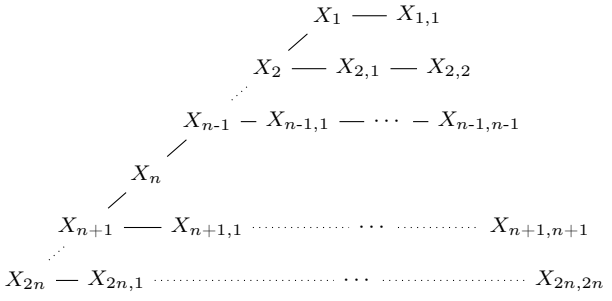
2nde composante de la 1ière clause de E



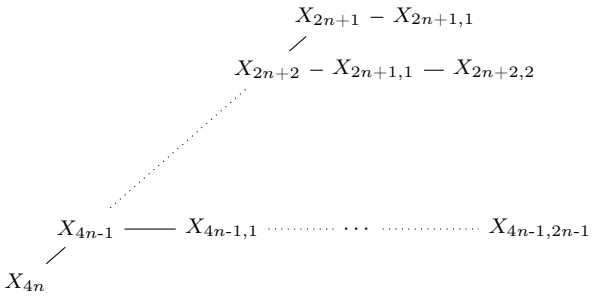
1ière composante de la $i^{ième}$ clause de E



2nde composante de la $i^{ième}$ clause de E



1ière composante de la $n^{ième}$ clause de E



2nde composante de la $n^{ième}$ clause de E

Chaque clause de cet ensemble contient deux composantes déconnectées et appartient à $\mathcal{L}^{\Sigma}_{fcns}$. On peut, sans perte de généralité, connecter ces composantes à l'aide d'un atome $fc(X_{2n}, X_{2n+1})$ et associer chaque variable à symbole de prédicat unaire quelconque afin d'obtenir une clause de $\mathcal{L}^{T_\Sigma}_{fcns}$. Après généralisation de cet ensemble, la taille du moindre généralisé obtenu est :

$$|\mathsf{lgg}(E)| = 2^n \times (2n-1) + 2^{n-1} \times \sum_{k=1}^{k=2n} k$$

La présence d'un facteur 2^n permet d'affirmer que la taille du moindre généralisé issu de cet ensemble est exponentielle dans le nombre d'exemples n. Ainsi, le calcul du moindre généralisé de cet ensemble est coûteux. Un exemple de cette construction ainsi que du moindre généralisé obtenu se trouve en section A.4 de l'annexe A. Nous résumons les différents résultats de cette section à l'aide du théorème suivant :

Théorème 21. *Il existe, pour chaque langage $\mathcal{L}^{\Sigma}_{fcns}$, $\mathcal{L}^{T_\Sigma}_{fcns}$ ou $\mathcal{L}^{T_\Sigma}_{nested}$, un ensemble de clauses E dont la taille du moindre généralisé $\mathsf{lgg}(E)$ est exponentielle dans le nombre de clauses de cet ensemble.*

5.3.9 Conclusion

Nous avons défini dans cette section les langages de clauses $\mathcal{L}^{\Sigma}_{fcns}$, $\mathcal{L}^{T_\Sigma}_{fcns}$ et $\mathcal{L}^{T_\Sigma}_{nested}$ permettant la représentation des arbres de l'ensemble T_Σ. Pour ces langages, nous avons proposé des algorithmes polynomiaux de θ-subsomption, de réduction ainsi que de moindre généralisation. L'utilisation de ces algorithmes a ensuite été étendue à d'autres familles de clauses comme la famille $\mathcal{F}_{\mathsf{FDP}}$ et la famille $\mathcal{F}_{\mathsf{MQD}}$. Enfin, pour les langages $\mathcal{L}^{\Sigma}_{fcns}$, $\mathcal{L}^{T_\Sigma}_{fcns}$ et $\mathcal{L}^{T_\Sigma}_{nested}$, nous avons montré l'existence d'un moindre généralisé de taille exponentielle.

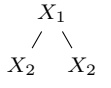
Le langage de clauses $\mathcal{L}^{T_{\Sigma\cup V}}_{pc}$, présenté dans la section précédente, propose également une représentation de l'ensemble des arbres T_Σ. Ce langage de clauses bénéficie d'une θ-subsomption, d'une réduction et d'une moindre généralisation linéaires dans la taille de leurs entrées. Nous désirons le comparer avec le langage $\mathcal{L}^{\Sigma}_{fcns}$ afin de voir si les généralisations obtenues sont équivalentes ou complémentaires.

5.4 Combinaisons des langages SOP et FDP

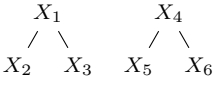
Dans cette section, nous mettons en évidence que les ensembles de clauses obtenues par moindre généralisation des clauses de $\mathcal{L}^{T_\Sigma}_{fcns}$ et de celles du langage $\mathcal{L}^{T_\Sigma}_{pc}$ diffèrent. Suite à cela, nous proposons une manière de les combiner afin de bénéficier des propriétés de chacun et étendons cela aux familles SOP et FDP.

5.4.1 Comparaison de $\mathcal{L}^{T_\Sigma}_{fcns}$ et de $\mathcal{L}^{T_\Sigma}_{pc}$

Chacun de ces langages, que ce soit $\mathcal{L}^{T_\Sigma}_{fcns}$ ou $\mathcal{L}^{T_\Sigma}_{pc}$, permet la représentation de n'importe quel arbre appartenant à T_Σ. Cependant, les arbres et motifs d'arbres obtenus suite à la moindre généralisation des langages $\mathcal{L}^{T_\Sigma}_{fcns}$ et $\mathcal{L}^{T_\Sigma}_{pc}$ sont différents :



$C \in \mathcal{L}^{T_{\Sigma\cup V}}_{pc}$



$D \in \mathcal{L}^{\Sigma}_{fcns}$

La clause C représente un arbre non étiqueté avec deux nœuds dont les labels sont identiques. Cette clause n'a pas d'équivalence dans $\mathcal{L}^{\Sigma}_{fcns}$ car ce langage ne permet pas de spécifier la contrainte d'égalité des labels dans un arbre. La clause D correspond, quant-à-elle, à deux sous-arbres non étiquetés. De manière similaire, cette clause n'a pas d'équivalence dans $\mathcal{L}^{T_{\Sigma\cup V}}_{pc}$ car chaque clause de ce langage ne peut représenter qu'un arbre enraciné. Il existe ainsi des motifs d'arbres propres à chacun de ces langages. Afin de ne pas choisir entre l'un de ces langages, nous proposons de les combiner.

5.4.2 Couplage SOP et FDP

Afin de combiner ces deux familles, nous construisons ces langages à la manière des clauses k-locales de [Kietz 1994a]. Rappelons qu'une clause k-locale est une clause dont les littéraux sont regroupés en fonction de propriétés que sont le déterminisme, la k-localité et le non-déterminisme. Nous optons pour une stratégie similaire et traitons indépendamment les parties SOP et FDP. Pour ce faire, nous supposons que ces parties ne partagent pas de variable, de constante, de prédicat et de symbole de fonction. Ainsi, une clause C obtenue par combinaison de deux langages $\mathcal{L}_1 \in \mathcal{F}_{sop}$ et $\mathcal{L}_2 \in \mathcal{F}_{\mathsf{FDP}}$ sans variable, constante, prédicat et symbole de fonction communs est de la forme $C = (C_{sop}, C_{\mathsf{FDP}})$ où $C_{sop} \in \mathcal{L}_1$ et $C_{\mathsf{FDP}} \in \mathcal{L}_2$.

Nous regardons maintenant le test de θ-subomption pour une telle clause. Les familles de clauses SOP et FDP bénéficient chacunes d'un test de θ-subsomption efficace que sont $\mathsf{sub}_{\mathsf{SOP}}$ (cf. section 5.2) et $\mathsf{sub}_{\mathsf{MQD}}$ (cf. section 5.3). Nous les utilisons afin de définir une subsomption efficace pour des clauses issues de ce couplage comme proposé dans le lemme suivant :

Lemme 11. *Soit deux clauses $C, D \in \mathcal{L}_1 \times \mathcal{L}_2$ avec $C = (C_{sop}, C_{\mathsf{FDP}})$ et $D = (D_{sop}, D_{\mathsf{FDP}})$ où $\mathcal{L}_1 \in \mathcal{F}_{sop}$ et $\mathcal{L}_2 \in \mathcal{F}_{\mathsf{FDP}}$ sans variable, constante, prédicat et symbole de fonction communs, la clause C θ-subsume D si et seulement si $C_{sop} \succeq_\theta D_{sop}$ et $C_{\mathsf{FDP}} \succeq_\theta D_{\mathsf{FDP}}$.*

Preuve. Le lemme 11 est une conséquence directe de l'absence de variable et de symbole de prédicat communs. En effet, les deux langages $\mathcal{L}_1$ et $\mathcal{L}_2$ ne partagent ni variable ni symbole de prédicat. La partie C_{sop} ne peut donc jamais subsumer la partie D_{FDP} et réciproquement. Le même constat peut-être fait pour C_{FDP} et D_{sop}. Ainsi, si $C \succeq_\theta D$ alors il faut obligatoirement que $C_{sop} \succeq_\theta D_{sop}$ et $C_{\mathsf{FDP}} \succeq_\theta D_{\mathsf{FDP}}$. La réciproque est plus facile à établir car si $C \succeq_\theta D$ alors $C_{sop} \succeq_\theta D_{sop}$ et $C_{\mathsf{FDP}} \succeq_\theta D_{\mathsf{FDP}}$ puisque les langages $\mathcal{L}_1$ et $\mathcal{L}_2$ ne partagent ni variable, ni symbole de prédicat. □

La complexité d'un test de θ-subsomption visant à tester si $C \succeq_\theta D$ est de $\mathcal{O}(|C_{sop}| + |C_{\mathsf{FDP}}| \times |D_{\mathsf{FDP}}|)$ en temps. Comme la θ-subsomption, le moindre généralisé bénéficie de l'absence de symbole de prédicat commun aux parties SOP et FDP. Ainsi, il est possible, comme décrit dans le lemme suivant, de calculer séparément les moindres généralisés de chacune des parties.

Lemme 12. *Soit deux clauses $C, D \in \mathcal{L}_1 \times \mathcal{L}_2$ avec $C = (C_{sop}, C_{\mathsf{FDP}})$ et $D = (D_{sop}, D_{\mathsf{FDP}})$ où $\mathcal{L}_1 \in \mathcal{F}_{sop}$ et $\mathcal{L}_2 \in \mathcal{F}_{\mathsf{FDP}}$ sans variable, constante, prédicat, ni symbole de fonction communs, le moindre généralisé de C et D est $\mathsf{lgg}(C, D) = (\mathsf{lgg}_{\mathsf{SOP}}(C_{sop}, D_{sop}), \mathsf{lgg}_{\mathsf{split}}(C_{\mathsf{FDP}}, D_{\mathsf{FDP}}))$. Le langage de clauses $\mathcal{L}_1 \times \mathcal{L}_2$ est clos par moindre généralisation.*

Preuve. Les littéraux des parties SOP et FDP ne partagent pas de symbole de prédicat, ils ne peuvent donc pas être généralisés ensemble. De plus, les langages SOP et FDP ne partagent aucun terme, la généralisation de leurs termes leur est donc obligatoirement propre. Ainsi, la moindre généralisation est obtenue en moindre généralisant séparément les

parties SOP et FDP, puis en faisant l'union des moindres généralisés obtenus. Les langages SOP et les langages FDP sont clos par moindre généralisation, il en est donc de même d'un langage combinant les deux puisque les parties SOP et FDP sont traitées séparément. □

Ce type de clauses hérite des propriétés issues de la famille SOP ainsi que de celles de la famille FDP. Proposons maintenant quelques couplages intéressants que nous utiliserons lors de nos expériences. Ces codages sont obtenus en couplant l'un des codages SOP suivants :

$pc{+}peano$, $pc{+}df$, $peano{+}df$ et $pc{+}peano{+}df$

avec un codage FDP :

$fcns{+}root$, $fcns{+}leaf$, $fcns{+}root{+}leaf$, $fcns\text{-}label$, $fcns\text{-}label{+}root$, $fcns\text{-}label{+}leaf$ et $fcns\text{-}label{+}root{+}leaf$

et donne ainsi les couplages SOP+FDP suivants :

$fcns{+}sop{+}peano$, $fcns{+}sop{+}df$, $fcns{+}sop{+}peano{+}df$, $fcns{+}peano{+}df$, $fcns\text{-}label{+}sop{+}peano$, $fcns\text{-}label{+}sop{+}df$, $fcns\text{-}label{+}sop{+}peano{+}df$, $fcns\text{-}label{+}peano{+}df$, ...

Une clause représentant un arbre codé par l'un de ces codages contient les littéraux des clauses de chaque codage couplé codant cet arbre. Ainsi, plus le nombre de codages couplés est élevé, plus la clause risque de contenir d'informations sur l'arbre codé au détriment de sa taille. Nous espérons, grâce à cela, améliorer la prédiction des hypothèses apprises lors de nos apprentissages tout en conservant une complexité polynomiale raisonnable. Ces différents codages seront comparés dans la section 5.6 traitant des expériences.

Avant d'effectuer une comparaison de ces codages, nous devons être sûrs qu'il est possible d'effectuer un apprentissage avec ceux-ci. En effet, malgré la proposition d'algorithmes polynomiaux pour nos différents langages, nous n'avons, pour le moment, pas la certitude qu'il est possible de construire n'importe quelle clause de $\mathcal{L}^{\Sigma}_{fcns}$ ou de $\mathcal{L}^{T_{\Sigma\cup V}}_{pc}$ à partir des clauses respectant ces codages et représentant des arbres. Visant la tâche de classification de documents XML à partir d'un apprentissage à base de moindre généralisé et de langages de clauses, ce point est à éclaircir. Nous procédons donc dans la section suivante à l'étude de ces langages dans le cadre de l'identification à la limite.

5.5 Identification à la limite par moindre généralisation

Afin de réaliser notre apprentissage dans le paradigme de l'identification à la limite [Gold 1967], nous y étudions nos langages et utilisons les opérations de θ-subsomption, de moindre généralisation et de réduction polynomiales afin de constituer un framework d'apprentissage. Commençons cette section en prouvant un résultat liant l'identification à la limite à base de moindre généralisation par positifs à celle par positifs et négatifs. Nous étudions ensuite l'identification à la limite à base de moindre généralisation par positifs et par positifs et négatifs pour chaque langage proposé précédemment.

5.5.1 Identification à la limite et moindre généralisé

Il existe, pour tout ensemble de clauses de Horn, un moindre généralisé unique sous θ-équivalence. Ainsi, lors d'un apprentissage par moindre généralisation à partir d'exemples positifs seuls, nous savons qu'il existe obligatoirement une clause les couvrant. Ce moindre

généralisé ne peut contenir aucun littéral ce qui signifie alors que ce dernier couvre toutes les clauses du langage. Ceci traduit alors un échec de l'apprentissage.

Dans le cas d'un apprentissage par positifs et négatifs à base de moindre généralisé, l'existence d'une hypothèse n'est pas assurée :

Exemple 48. *Soit trois clauses C_1, C_2 et C_3 d'un même langage telles que $C_1 \succeq_\theta C_3$. L'ensemble des exemples positifs contient C_1 et C_2 tandis que C_3 est l'exemple négatif.*

La clause C_1 couvre C_3, ainsi nous savons par la définition du moindre généralisé que $\mathsf{lgg}(C_1, C_2) \succeq_\theta C_3$. Il est donc impossible de construire un moindre généralisé de C_1 et C_2 rejetant C_3. Il n'existe donc pas d'hypothèse couvrant l'ensemble des positifs, incluant C_1 et C_2, et rejetant l'exemple négatif C_3.

Ces constats maintenant faits, nous nous intéressons au lien existant entre l'identification à la limite par positifs seuls à partir d'un moindre généralisé et celle par positifs et négatifs. Cette relation est explicitée par le théorème suivant :

Théorème 22. *Soit un langage de clauses $\mathcal{L}$, si $\mathcal{L}$ est identifiable (polynomialement en temps et en données) à la limite à base de moindre généralisé par positifs seuls alors $\mathcal{L}$ est identifiable (polynomialement) à la limite à base de moindre généralisé par positifs et négatifs.*

Preuve. Nous allons montrer que l'identification (polynomiale) à la limite par positifs seuls d'un langage de clauses $\mathcal{L}$ à l'aide d'un moindre généralisé implique l'identification (polynomiale) à la limite par positifs et négatifs de ce langage à l'aide d'un algorithme de moindre généralisation.

Nous savons que pour toute hypothèse $h \in \mathcal{L}$, l'ensemble de clauses $\mathcal{L}$ peut-être divisé en deux sous-ensembles distincts que sont les exemples positifs $\mathcal{L}_h^+$ et les exemples négatifs $\mathcal{L}_h^-$ (cf. chapitre 3 section 3.4). Ces ensembles respectent la contrainte suivante : $\forall e \in \mathcal{L}_h^+,\ h \succeq e$ et $\forall e \in \mathcal{L}_h^-,\ h \not\succeq e$.

Si le langage $\mathcal{L}$ est identifiable polynomialement à la limite par positifs seuls à l'aide d'un moindre généralisé, alors il existe un algorithme $\mathcal{A}$ de moindre généralisation. Cet algorithme retourne une solution en temps polynomial par rapport à la taille de son entrée telle que cette solution est consistante avec cette entrée. Pour chaque hypothèse $h \in \mathcal{L}$, il existe un ensemble d'exemples positifs EC_h^+, appelé échantillon caractéristique, dont la taille est polynomiale dans celle de h. Cet ensemble respecte la contrainte suivante : $\mathcal{A}(EC_h^+) = \mathcal{A}(\mathcal{L}_h^+) = \mathsf{lgg}(\mathcal{L}_h^+) = h$ (modulo équivalence). Ainsi, l'hypothèse apprise par $\mathcal{A}$ à partir de EC_h^+ reconnait l'ensemble des exemples positifs. Tout exemple non reconnu appartient alors à $\mathcal{L}_h^-$.

L'identification polynomiale à la limite d'un langage à l'aide d'un moindre généralisé par positifs et négatifs nécessite également un algorithme $\mathcal{A}'$. Cet algorithme retourne, en temps polynomial dans la taille de son entrée, une hypothèse consistante avec cette entrée. Similairement à l'identification polynomiale à la limite par positifs seuls, il existe, pour toute hypothèse h, un échantillon caractéristique EC_h^{+-} de taille polynomiale dans celle de h. L'hypothèse h apprise lors de l'identification à limite par moindre généralisation des exemples positifs seuls reconnait exactement les exemples positifs de $\mathcal{L}_h^+$. Ceci est équivalent à rejeter tous les exemples négatifs $\mathcal{L}_h^-$. Cette hypothèse h est donc également la solution d'un apprentissage par positifs et négatifs. Les contraintes de polynomialité respectées par l'algorithme $\mathcal{A}$ et l'ensemble des exemples EC_h^+, l'égalité suivante est donc vraie : $\mathcal{A}'(EC_h^{+-}) = \mathcal{A}(EC_h^+) = \mathcal{A}(\mathcal{L}_h^+) = \mathsf{lgg}(\mathcal{L}_h^+) = h$.

Ainsi, si un langage $\mathcal{L}$ est identifiable polynomialement en temps et données à la limite par positifs seuls à l'aide d'un moindre généralisé, alors il est identifiable polynomialement en temps et données à la limite par positifs et négatifs à l'aide d'un moindre généralisé.

□

Ce résultat est intéressant puisque dans le cas où l'identification à la limite d'un langage par moindre généralisé à partir d'exemples positifs seuls s'avère vrai, il en découle que l'identification à la limite d'un langage par moindre généralisé à partir d'exemples positifs et négatifs l'est également. Ainsi, nous procédons à l'étude des langages précédemment proposés en commençant par l'identification à la limite par positifs seuls à l'aide d'un algorithme de moindre généralisation.

5.5.2 Le langage $\mathcal{L}_{pc}^{T_{\Sigma\cup V}}$

Avant d'étudier l'apprentissage à la limite, par positifs seuls puis par positifs et négatifs, de clauses de $\mathcal{L}_{pc}^{T_{\Sigma\cup V}}$ à partir de clauses $\mathcal{L}_{pc}^{T_\Sigma}$, nous faisons une observation sur la taille de l'alphabet qui est l'ensemble des constantes autorisées dans une clause de $\mathcal{L}_{pc}^{T_\Sigma}$. Nous avons à différencier le cas où $|\Sigma| = 1$ des autres cas. Dans le cas où l'alphabet ne contient qu'une constante, le moindre généralisé de deux atomes $p(t_1)$ et $p(t_2)$ ne peut être égal à $p(X)$ que si t_1 et/ou t_2 sont des variables. Dès lors, une clause de $\mathcal{L}_{pc}^{T_{\Sigma\cup V}}$ contenant au moins une variable ne peut être apprise à partir d'exemples de $\mathcal{L}_{pc}^{T_\Sigma}$ par moindre généralisation. Cependant, s'intéresser à un langage $\mathcal{L}_{pc}^{T_{\Sigma\cup V}}$ n'autorisant qu'un symbole de prédicat a peu d'intérêt puisque cela revient à dire que tous les nœuds possèdent le même label. Dans ce cas, les labels peuvent être omis et le langage considéré sera alors $\mathcal{L}_{pc}^{T_V}$. Pour cette raison, nous considérons par la suite que $|\Sigma| > 1$.

À partir de positifs seuls

Nous étudions en premier lieu l'identification à la limite à partir d'exemples positifs par moindre généralisation. Nous commençons par définir la construction de l'échantillon caractéristique d'une hypothèse de $\mathcal{L}_{pc}^{T_{\Sigma\cup V}}$. Afin de simplifier la représentation des clauses présentes dans l'échantillon, nous faisons le choix de regrouper les littéraux connectés en composantes à la manière d'une multi-clause. Nous regroupons également, dans une autre composante, tous les atomes fondés. Les symboles de prédicat du langage $\mathcal{L}_{pc}^{T_{\Sigma\cup V}}$ sont unaires. Ainsi, un atome de ce langage ne peut contenir, au plus, qu'une variable. Les atomes présents dans une composante sont alors connectés par une même variable. Le nombre de variables dans une clause de $\mathcal{L}_{pc}^{T_{\Sigma\cup V}}$ ne peut excéder son nombre de littéraux. Soit $X_1,\ X_2, \cdots, X_n$ les variables présentes dans une clause $h \in \mathcal{L}_{pc}^{T_{\Sigma\cup V}}$. Nous notons C_i la composante connectée par la variable X_i et C_c la composante non connectée contenant les atomes fondés de la clause.

Nous pouvons maintenant donner la construction de notre ensemble caractéristique. Soit une hypothèse $h \in \mathcal{L}_{pc}^{T_{\Sigma\cup V}}$ où $h = C_c \cup C_1 \cup \cdots \cup C_n$ et $n < |h|$, on construit l'ensemble $E_h = \{e_0, e_1, \cdots, e_n\} \subseteq \mathcal{L}_{pc}^{T_\Sigma}$ tel que :

— $e_0 = C_c \cup C_1\theta_1 \cup \cdots \cup C_n\theta_n$ avec $\theta_i = \{X_i/a\}$ où $i \in [1..n]$ et $a \in \Sigma$

— et pour tout $i \in [1..n]$, $e_i = C_c \cup C_1\theta_1 \cup \cdots \cup C_i\theta_i \cup \cdots \cup C_n\theta_n$ avec
$\theta_k = \begin{cases} \{X_k/a\} & \text{si } k = i \\ \{X_k/b\} & \text{sinon} \end{cases}$ où $k \in [1..n]$, $a, b \in \Sigma$ et $a \neq b$.

Par la suite, et afin de simplifier les notations, nous posons $C_i^a = C_i\{X_i/a\}$ et $C_i^b = C_i\{X_i/b\}$. L'exemple suivant illustre cette construction.

Exemple 49. *Soit l'alphabet $\Sigma = \{a, b, c\}$ et la clause de $\mathcal{L}_{pc}^{T_{\Sigma\cup V}}$:*

$$\begin{array}{rcl} h & = & 0(X_1), \\ & & \text{0-0}(a),\ \text{0-1}(c),\ \text{0-2}(b), \\ & & \text{0-0-0}(X_1),\ \text{0-2-0}(c),\ \text{0-2-1}(X_2). \end{array}$$

La décomposition de h en composantes est la suivante :

$$\begin{array}{rcl} C_c & = & \{\text{0-0}(a),\ \text{0-1}(c),\ \text{0-2}(b),\ \text{0-2-0}(c)\} \\ C_1 & = & \{0(X_1),\ \text{0-0-0}(X_1)\} \\ C_2 & = & \{\text{0-2-1}(X_2)\} \end{array}$$

telle que $h = C_c \cup C_1 \cup C_2$. L'échantillon caractéristique issu de h contient alors trois clauses e_0, e_1 et e_2 comme suit :

$$\begin{array}{rcl} e_0 & = & C_c \cup C_1^a \cup C_2^a \\ & = & \{\text{0-0}(a),\ \text{0-1}(c),\ \text{0-2}(b),\ \text{0-2-0}(c)\} \cup \{0(a),\ \text{0-0-0}(a)\} \cup \{\text{0-2-1}(a)\} \\ e_1 & = & C_c \cup C_1^a \cup C_2^b \\ & = & \{\text{0-0}(a),\ \text{0-1}(c),\ \text{0-2}(b),\ \text{0-2-0}(c)\} \cup \{0(a),\ \text{0-0-0}(a)\} \cup \{\text{0-2-1}(b)\} \\ e_2 & = & C_c \cup C_1^b \cup C_2^a \\ & = & \{\text{0-0}(a),\ \text{0-1}(c),\ \text{0-2}(b),\ \text{0-2-0}(c)\} \cup \{0(b),\ \text{0-0-0}(b)\} \cup \{\text{0-2-1}(a)\} \end{array}$$

Nous vérifions maintenant que cet ensemble d'exemples constitue bien un échantillon caractéristique de h rendant possible son identification polynomiale à la limite par moindre généralisation.

Lemme 13. *Soit un alphabet Σ tel que $|\Sigma| > 1$, toute clause de $h \in \mathcal{L}_{pc}^{T_{\Sigma \cup V}}$ est identifiable polynomialement à la limite en temps et donnée par moindre généralisation et à partir d'exemples de $\mathcal{L}_{pc}^{T_\Sigma}$ formant une présentation positive.*

Preuve. Soit un alphabet Σ tel que $|\Sigma| > 1$, une clause $h \in \mathcal{L}_{pc}^{T_{\Sigma \cup V}}$ et l'ensemble E_h de clauses de $\mathcal{L}_{pc}^{T_\Sigma}$ défini par la construction. On veut montrer que h est polynomialement identifiable à la limite à partir de E_h par moindre généralisation. Pour cela, nous devons vérifier que le calcul d'un moindre généralisé est réalisable en temps polynomial dans la taille de son entrée, que la clause obtenue est consistante avec chaque exemple de son entrée et que la taille de l'échantillon caractéristique est polynomiale par rapport à la taille de l'hypothèse dont il est issu.

Nous commençons par la taille de E_h. Chaque clause $e_i \in E_h$ contient, par sa construction, le même nombre de littéraux que h. Le nombre de clauses de E_h dépend du nombre de variables de h. Celui-ci est borné par $|h|$ puisque chaque atome ne contient pas plus d'une variable. La taille de E_h est alors, dans le pire des cas, en $\mathcal{O}(|h|^2)$ si chaque littéral de h contient une variable distincte des autres. La taille de E_h est donc bien polynomiale par rapport à celle de h.

Vérifions maintenant la complexité du calcul du moindre généralisé de E_h. Nous savons que le moindre généralisé de deux clauses SOP ne peut croître avec le nombre de clauses généralisées. On sait également qu'il est calculable en $\mathcal{O}(min(|C|, |D|))$ pour deux clauses SOP C et D. Enfin, le moindre généralisé d'un ensemble de clauses est correct avec chaque clause de cet ensemble. On peut donc calculer en temps polynomial le moindre généralisé d'un ensemble de clauses SOP correct avec cet ensemble. Plus précisément, ce calcul peut être effectué en temps $\mathcal{O}((n-1) \times |h|^2)$ puisque chaque exemple de E_h est de taille $|h|$.

Reste à prouver que pour une clause $h \in \mathcal{L}_{pc}^{T_{\Sigma \cup V}}$, $\mathsf{lgg}(E_h) = h$. Nous réalisons cela par induction. Chaque composante contient des littéraux dont les symboles de prédicat lui sont propres. Seul le moindre généralisé de chaque composante de la première clause avec l'unique composante correspondante de la seconde clause est donc à faire. Le calcul du moindre généralisé de e_0 et e_1 est le suivant :

$$\begin{aligned} \mathsf{lgg}(e_0, e_1) &= \mathsf{lgg}(C_c, C_c) \cup \mathsf{lgg}(C_1^a, C_1^b) \cup \mathsf{lgg}(C_2^a, C_2^a) \cup \cdots \cup \mathsf{lgg}(C_n^a, C_n^a) \\ &= C_c \cup C_1 \cup C_2^a \cup \cdots \cup C_n^a \end{aligned}$$

Remarquons que le $\mathsf{lgg}(C_1^a, C_1^b) = C_1$ tandis que les autres composantes restent inchangées. Le calcul du moindre généralisé de e_2 avec $\mathsf{lgg}(e_0, e_1)$ est le suivant :

$$\begin{aligned} \mathsf{lgg}(e_0, e_1, e_2) &= \mathsf{lgg}(\mathsf{lgg}(e_0, e_1), e_2) \\ &= \mathsf{lgg}(C_c, C_c) \cup \mathsf{lgg}(C_1, C_1^b) \cup \mathsf{lgg}(C_2^a, C_2^b) \cup \mathsf{lgg}(C_3^a, C_3^a) \cup \cdots \\ &\quad \cup \mathsf{lgg}(C_n^a, C_n^a) \\ &= C_1 \cup C_2 \cup C_3^a \cup \cdots \cup C_n^a \end{aligned}$$

Par induction, on a $lgg(e_0, e_1, \cdots, e_n) = C_c \cup C_1 \cup \cdots \cup C_n = h$. L'ensemble E_h est donc bien un ensemble caractéristique de h.

L'ensemble E_h est de taille polynomiale par rapport à celle de h, le calcul du moindre généralisé d'un ensemble E est de complexité en temps quadratique vis-à-vis de la taille E et $\mathsf{lgg}(E_h) = h$, on en conclut que $\mathcal{L}_{pc}^{T_{\Sigma \cup V}}$ est polynomialement identifiable par moindre généralisé à partir d'exemples de $\mathcal{L}_{pc}^{T_\Sigma}$.

□

L'identification polynomiale à la limite de $\mathcal{L}_{pc}^{T_{\Sigma \cup V}}$ par moindre généralisé à partir d'exemples de $\mathcal{L}_{pc}^{T_\Sigma}$ prouvée dans le cas des positifs seuls, nous nous intéressons maintenant au cas utilisant positifs et négatifs.

À partir de positifs et négatifs

Notre approche de cette tâche se base sur la réutilisation de notre résultat précédent. Ainsi, nous abordons la tâche d'identification à la limite par positifs et négatifs comme une extension de celle par positifs seuls. L'identification à la limite à l'aide d'un moindre généralisé par positifs et négatifs du langage $\mathcal{L}_{pc}^{T_{\Sigma \cup V}}$ à partir d'exemples de $\mathcal{L}_{pc}^{T_\Sigma}$ est ainsi une conséquence du théorème 22 appliqué au lemme 13.

Lemme 14. *Soit un alphabet Σ tel que $|\Sigma| > 1$, toute clause de $h \in \mathcal{L}_{pc}^{T_{\Sigma \cup V}}$ est identifiable polynomialement à la limite en temps et donnée à partir d'exemples positifs et négatifs issus de $\mathcal{L}_{pc}^{T_\Sigma}$.*

Ce résultat suppose que l'échantillon caractéristique contienne comme ensemble d'exemples positifs les exemples de l'échantillon caractéristique défini précédemment pour l'identification à la limite par positifs seuls à l'aide d'un moindre généralisé de ce même langage. L'ensemble des exemples négatifs n'est pas utilisé. Il doit être, au plus, de taille polynomiale dans la taille de l'hypothèse.

Nous avons prouvé que le langage $\mathcal{L}_{pc}^{T_{\Sigma \cup V}}$, pour un alphabet Σ d'au moins deux symboles, est identifiable polynomialement à la limite en temps et en données à partir d'exemples positifs seuls et d'exemples positifs et négatifs. Ces exemples sont issus de $\mathcal{L}_{pc}^{T_\Sigma}$. Un apprentissage efficace est donc réalisable pour ce langage de clauses et est applicable théoriquement à la classification de documents XML.

5.5.3 Le langage $\mathcal{L}_{fcns}^{\Sigma}$

Nous nous intéressons maintenant à l'identification polynomiale à la limite par moindre généralisation du langage $\mathcal{L}_{fcns}^{\Sigma}$ à partir d'exemples de $\mathcal{L}_{fcns}^{T_\Sigma}$.

Identification par positifs seuls

Nous avons vu précédemment que le moindre généralisé de clauses de $\mathcal{L}^{\Sigma}_{fcns}$ peut être de taille exponentielle dans la norme de l'ensemble des exemples à partir duquel il a été construit. Il est alors impossible de construire une hypothèse de taille exponentielle dans le nombre des exemples en temps polynomial dans la norme de cet ensemble. Or, ce point est un critère nécessaire à l'identification polynomiale à la limite par moindre généralisation. Nous avons donc le lemme suivant :

Lemme 15. *Soit un alphabet Σ, toute clause de $h \in \mathcal{L}^{\Sigma}_{fcns}$ n'est pas identifiable polynomialement à la limite par moindre généralisation à partir d'exemples de $\mathcal{L}^{T_\Sigma}_{fcns}$.*

L'identification polynomiale à la limite par moindre généralisé à partir de positifs seuls est impossible à cause de la contrainte de polynomialité. Nous l'omettons donc et regardons si ce langage est identifiable à la limite par positifs seuls. Nous proposons ainsi de construire, pour une hypothèse $h \in \mathcal{L}^{\Sigma}_{fcns}$, l'ensemble d'exemples E_h comme échantillon caractéristique de h.

Une multi-clause $h = \{C_1, \cdots, C_n\}$ de $\mathcal{L}^{\Sigma}_{fcns}$ est une clause déconnectée. Chacune de ses variables peut être associée à un unique prédicat unaire $a/1$ avec $a \in \Sigma$. Elle n'appartient ainsi pas obligatoirement à $\mathcal{L}^{T_\Sigma}_{fcns}$. Or, l'ensemble E_h est constitué de clauses connectées de $\mathcal{L}^{T_\Sigma}_{fcns}$. Chaque variable de chacune de ces clauses est associée à un prédicat unaire $a/1$ avec $a \in \Sigma$. Afin de construire les clauses de l'ensemble caractéristique à partir de h, deux étapes sont nécessaires. La première consiste à connecter les différentes composantes de la clause h afin d'obtenir des clauses connectées. La seconde consiste à ajouter à chaque variable de chacune de ces clauses connectées un atome unaire $a/1$ avec $a \in \Sigma$ si elle n'en dispose pas.

Commençons par supprimer de h la composante de la forme :

$$(fc(X_1, X_2),\ ns(X_2, X_3))$$

s'il existe une autre composante dont l'un des atomes a pour prédicat ns. Cette composante est une conséquence directe de la présence d'un atome dont le prédicat est ns. La moindre généralisation d'exemples de $\mathcal{L}^{T_\Sigma}_{fcns}$ non limités à des arbres d'arité 1 la contient donc systématiquement. Il n'est ainsi pas nécessaire de la considérer. Continuons ensuite en désignant, dans chaque composante, des variables dont le rôle est d'assurer une connexion avec une autre composante. Pour chaque composante $C_i \in h$, nous notons A_i la variable de C_i pour laquelle il n'existe pas de variable X telle que $fc(X, A_i), ns(X, A_i) \in C_i$, et B_i une variable de C_i pour laquelle il n'existe pas de variable X telle que $fc(B_i, X), ns(B_i, X) \in C_i$. Il existe, pour une composante C_i , qu'une unique variable A_i tandis que plusieurs variables B_i sont envisageables. Deux composantes différentes, C_i et C_j, peuvent ensuite être connectées de deux manières possibles. Soit la variable A_i est connectée à la variable B_j, soit la variable B_i est connectée à la variable A_j. Les variables A_i et B_j ainsi que A_j et B_i sont connectées à l'aide d'un ensemble d'atomes utilisant le même prédicat : fc ou ns. Nous notons respectivement L^{fc}_{ij} et L^{ns}_{ij} les ensembles d'atomes avec pour prédicat fc et ns qui lient la variable B_i à la variable A_j. L'ensemble L^{fc}_{ij} est ainsi de la forme :

$$fc(B_i, X_{i,2}),\ fc(X_{i,2}, X_{i,3}),\ \cdots,\ fc(X_{i,k}, X_{i,k+1}),\ fc(X_{i,k+2}, A_j)$$

tandis que l'ensemble L^{ns}_{ij} est de la forme :

$$ns(B_i, X_{i,2}),\ ns(X_{i,2}, X_{i,3}),\ \cdots,\ ns(X_{i,k}, X_{i,k+1}),\ ns(X_{i,k+2}, A_i)$$

où k est la taille du plus grand ensemble d'atomes connectés de même prédicat présents dans h. Une composante est ainsi connectée a, au plus, deux autres composantes et chaque composante est utilisée une seule fois dans chaque clause exemple. Nous disposons maintenant

de plusieurs clauses connectées de la forme :

$$C_{i_1} \cup L^s_{i_1 i_2} \cup C_{i_2} \cup \cdots \cup L^s_{i_{m-1} i_m} \cup C_{i_m}$$

où s est soit fc ou ns et $\{i_1, \cdots, i_m\} = \{1, \cdots, m\}$. Ces clauses ne représentent pas toutes un arbre. En effet, si la composante C_{i_1} ne contient pas un atome de la forme $fc(A_{i_1}, X)$, alors il est nécessaire de l'ajouter. Nous ajoutons ainsi l'ensemble L_0 suivant aux clauses connectées précédemment définies telle que $L_0 = fc(X, A_{i_1})$ si $fc(A_{i_1}, X) \notin C_{i_1}$ sinon $\emptyset$ et obtenons l'ensemble des clauses suivantes :

$$\begin{array}{rcl} E^h_{connect} & = & \{\{L_0 \cup C_{i_1} \cup L^s_{i_1 i_2} \cup C_{i_2} \cup \cdots \cup L^s_{i_{m-1} i_m} \cup C_{i_m}\} \mid \\ & & \{i_1, \cdots, i_m\} = \{1, \cdots, m\},\ h = \{C_1, \cdots, C_m\} \text{ et } s \in \{fc, ns\}\} \end{array}$$

Cet ensemble contient $2 \times m!$ clauses avec une taille inférieure ou égale à $m \times k + |h| + 1$. Chaque clause correspond à un arrangement sans répétition connectant les composantes de h selon le protocole décrit ci-dessous. Chaque variable d'une de ces clauses n'est cependant pas forcément étiquetée. Ainsi, nous passons à la deuxième étape.

La seconde étape de la construction consiste en l'étiquetage des variables sans label. Une variable non étiquetée est une variable non utilisée comme argument d'un atome unaire $a/1$ avec $a \in \Sigma$. Nous ajoutons ainsi pour chacune de ces variables un atome correspondant. Ces atomes ne doivent pas être conservés après moindre généralisation, l'échantillon caractéristique doit donc contenir des clauses pour lesquelles ces variables possèdent des étiquettes différentes. Soit une clause $C \in E^h_{connect}$, nous notons $\mathcal{N}_C$ l'ensemble des variables non étiquetées de C, i.e. $\mathcal{N}_C = \{X \in vars(C) \mid \nexists a(X) \in C \text{ avec } a \in \Sigma\}$. On construit ensuite $\mathcal{A}_C$ l'ensemble de tous les ensembles de taille $|\mathcal{N}_C|$ contenant pour chaque variable $X \in \mathcal{N}_C$ un atome unaire avec, pour prédicat, un label $a \in \Sigma$ et, comme argument, cette variable X. Cet ensemble contient l'ensemble des arrangements avec répétitions consistant à lier un label Σ à une variable de $\mathcal{N}_C$. L'ensemble $\mathcal{A}_C$ contient ainsi $|\Sigma|^{|\mathcal{N}_C|}$ ensembles d'atomes. La manière dont sont étiquetés les nœuds maintenant définie, nous pouvons donner l'ensemble des exemples issus de h destinés à être l'échantillon caractéristique :

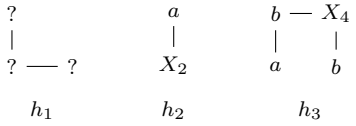
$$E_h = \{CL \cup A \mid CL \in E^h_{connect} \text{ et } A \in \mathcal{A}_{CL}\}$$

Cet ensemble contient $2 \times m! \times |\Sigma|^{|vars(\mathcal{N}_h)|}$ clauses. Chacune de ses clauses posséde au moins $m \times k + |h|$ littéraux. Illustrons cette construction.

Exemple 50. *Soit l'hypothèse $h = \{h_1, h_2, h_3\} \in \mathcal{L}^{\Sigma}_{fcns}$ où :*

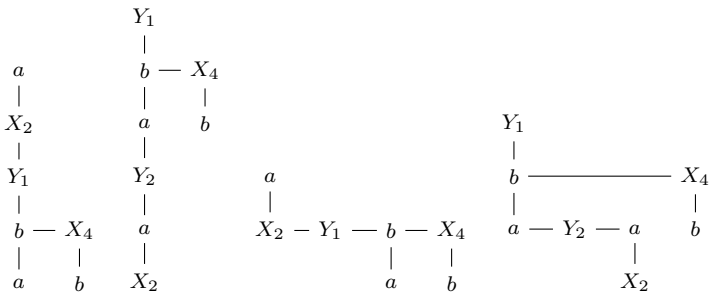
$$\begin{array}{rcl} h_1 & = & fc(X_1, X_2),\ ns(X_2, X_3). \\ h_2 & = & fc(X_1, X_2),\ a(X_1). \\ h_3 & = & ns(X_1, X_2),\ b(X_1),\ fc(X_1, X_3),\ a(X3),\ fc(X_3, X_4),\ b(X4). \end{array}$$

Ces composantes ont pour représentations arborescentes :

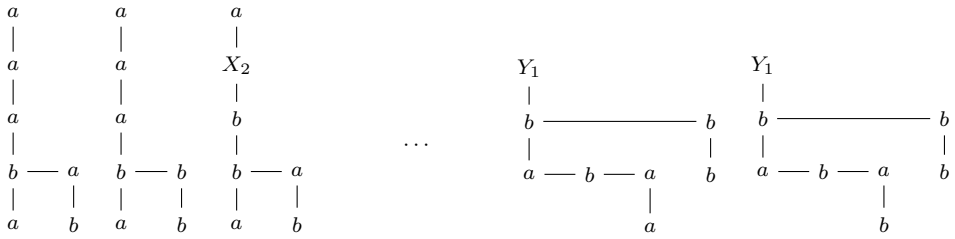


h_1 h_2 h_3

La composante h_1 est une conséquence de h_3, elle est donc omise. Nous donnons maintenant la forme arborescente des clauses de $E_{connect}$ obtenues suite à la première étape :



On construit ensuite, à partir de ces clauses, l'ensemble des clauses de $\mathcal{L}_{fcns}^{T_\Sigma}$ en ajoutant une étiquette à chaque nœud n'en possédant pas.



Nous avons ainsi construit l'ensemble E_h, échantillon caractéristique de l'hypothèse h.

Cette construction nous permet d'obtenir le résultat suivant :

Lemme 16. *Soit un alphabet Σ, toute clause $h \in \mathcal{L}_{fcns}^{\Sigma}$ est identifiable à la limite par moindre généralisation à partir d'exemples positifs seuls de $\mathcal{L}_{fcns}^{T_\Sigma}$.*

Preuve. Démontrons que le moindre généralisé de l'ensemble d'exemples E_h est équivalent à l'hypothèse h à partir de laquelle il a été créé.

L'ensemble caractéristique E_h est défini comme suit : $E_h = \{CL \cup A \mid CL \in E_{connect}$ et $A \in \mathcal{A}_{CL}\}$. Il peut être réécrit sous la forme d'une union d'ensemble de clauses $E_h = E_h^{CL_1} \cup E_h^{CL_2} \cup \cdots E_h^{CL_{2\times m!}}$. Chaque ensemble de clauses regroupe les clauses construites à partir d'une même clause $CL_i \in E_{connect}$ et ne partage ainsi pas de clause en commun. Chaque clause d'un même groupe $E_h^{CL_i}$ est subsumée par la clause CL_i correspondante. Le moindre généralisé d'un ensemble de clauses contient donc les atomes présents dans la clause CL_i à un renommage de variables prés. Chaque clause d'un ensemble de clauses $E_h^{CL_i}$ est l'un des arrangements possibles avec répétitions liant un label Σ à une variable de $\mathcal{N}_{CL_i}$. Le moindre généralisé réduit d'un ensemble $E_h^{CL_i}$ ne contient donc aucun atome obtenu par moindre généralisation des atomes de $A \in \mathcal{A}_{CL_i}$ et est équivalent à la clause CL_i. L'ensemble des moindres généralisés des ensembles de clauses $E_h^{CL_1}, E_h^{CL_2}, \cdots E_h^{CL_{2\times m!}}$ est alors équivalent à l'ensemble de clauses $E_{connect}^h$. Continuons donc avec ce dernier.

Nous procédons maintenant au calcul du moindre généralisé des clauses de $E_{connect}^h$. Les clauses de $E_{connect}^h$ sont divisibles en deux sous-ensembles distincts de clauses en fonction du choix du symbole de prédicat des atomes liant les composantes dans une même clause. L'un de ses ensembles, noté $E_{connect,fc}^h$, contient les clauses dont les composantes sont liées par des atomes utilisant le symbole de prédicat fc et l'autre, noté $E_{connect,ns}^h$, contient celles utilisant le symbole de prédicat ns. On a alors $E_{connect} = E_{connect,fc}^h \cup E_{connect,ns}^h$. Chaque clause de l'ensemble $E_{connect,fc}^h$ correspond à un arrangement sans permutation de l'ordre dans lequel

les composantes C_i de h sont connectées les unes aux autres. Ainsi, aucune composante C_i ne conserve les mêmes voisins dans ces clauses, tout comme $E^h_{connect,ns}$. Les liens créés entre les composantes diffèrent par le symbole de prédicat entre les ensembles $E^h_{connect,fc}$ et $E^h_{connect,ns}$. Ces derniers ne peuvent être conservés. Ainsi, seules les composantes de h sont conservées lors d'une moindre généralisation. Chaque clause de h est apprenable par moindre généralisation de l'ensemble E_h. Le langage *fcns* est donc identifiable à la limite par moindre généralisation à partir d'exemples appartenant à $\mathcal{L}^{T_\Sigma}_{fcns}$.

□

Le langage $\mathcal{L}^{\Sigma}_{fcns}$ est identifiable à la limite par moindre généralisation. La contrainte de polynomialité ne peut être respectée du fait de l'explosion de la taille du moindre généralisé.

Apprentissage par positifs et négatifs de $\mathcal{L}^{\Sigma}_{fcns}$

Le résultat d'identification à la limite par moindre généralisation à partir d'exemples positifs établi, il peut être étendu à l'aide du théorème 22 pour l'identification à la limite par moindre généralisation à partir d'exemples positifs et négatifs.

Lemme 17. *Soit un alphabet Σ, toute clause $h \in \mathcal{L}^{\Sigma}_{fcns}$ est identifiable à la limite par moindre généralisation à partir d'exemples positifs et négatifs de $\mathcal{L}^{T_\Sigma}_{fcns}$.*

Une identification par positifs seuls ne peut être polynomiale en temps et en données pour le langage $\mathcal{L}^{\Sigma}_{fcns}$ à partir d'exemples de $\mathcal{L}^{T_\Sigma}_{fcns}$. Au vu de ce résultat et des précédents, nous désirons prouver que l'apprentissage d'une clause de $\mathcal{L}^{\Sigma}_{fcns}$ à partir d'un ensemble d'exemples positifs et négatifs de $\mathcal{L}^{T_\Sigma}_{fcns}$ n'est pas trivial et ne peut donc être fait en temps polynomial. Dans ce but, nous transformons notre problème d'apprentissage d'une clause de $\mathcal{L}^{\Sigma}_{fcns}$ à partir d'exemples positifs et négatifs de $\mathcal{L}^{T_\Sigma}_{fcns}$ en problème décisionnel visant à tester l'existence d'une clause de $\mathcal{L}^{\Sigma}_{fcns}$ reconnaissant les positifs et rejetant les négatifs. Construire une telle clause confirme son existence, ainsi ce problème décisionnel précise la dureté de ce problème d'apprentissage.

Le problème décisionnel est le suivant : Soit un ensemble d'exemples $E \subseteq \mathcal{L}^{T_\Sigma}_{fcns}$ tel que $E = E^+ \cup E^-$ et $E^+ \cap E^- = \emptyset$, existe-il une clause h de $\mathcal{L}^{\Sigma}_{fcns}$ t.q. $\forall e \in E$, $h \succeq e$ si $e \in E^+$ et $h \not\succeq e$ sinon. Ceci est équivalent à tester si l'ensemble $E \in CONS$ où $CONS = \{E_\varphi \subseteq \mathcal{L}^{T_\Sigma}_{fcns} \mid \exists h \in \mathcal{L}^{\Sigma}_{fcns}$ t.q. $\forall e \in E_\varphi, h \succeq e$ si $e \in E^+_\varphi$ sinon $h \not\succeq e\}$. Le lien entre ce problème décisionnel et notre tâche d'apprentissage est facile à voir avec l'algorithme 13.

Algorithme 13 Tester s'il existe une hypothèse h consistante avec un ensemble E d'exemples. L'algorithme $\mathcal{A}$ la construit à partir de E si elle existe.

function *CONS*(E)
1: let $h = \mathcal{A}(E)$;
2: **if** $h \neq \varnothing$ **then return** *true* ;
3: **else return** *false* ;
end function

De cette manière, l'existence d'une hypothèse pour un ensemble d'exemples donné signifie que l'on a dû la construire auparavant afin de pouvoir affirmer son existence. Ce résultat est présenté dans le lemme suivant.

Lemme 18. $\forall \varphi \in 3CNF\ .\ E_\varphi \in CONS \iff \varphi \in 3SAT_{1in3}$.

Rappelons que tester l'appartenance d'une formule à l'ensemble $3SAT_{1in3}$ a été prouvé comme NP-complet [Schaefer 1978]. Ainsi, ce lemme affirme qu'il existe des instances du problème *CONS* dont la résolution est exponentielle en temps dans la taille de l'entrée. Il peut être prouvé à l'aide de la construction détaillée ci-dessous. Elle a pour but de représenter les évaluations des disjonctions d'une formule $3CNF$ sous la forme d'un ensemble de clauses représentant des arbres respectant le codage *first-child nextsibling*. Nous avons ainsi besoin d'une formule $3CNF$ $\varphi = (l_{11} \vee l_{12} \vee l_{13}) \wedge \cdots \wedge (l_{m1} \vee l_{m2} \vee l_{m3})$ telle que $l_{11}, l_{12}, l_{13}, l_{21}, \cdots, l_{m1}, l_{m2}, l_{m3} \in \{x_1, \bar{x}_1, \cdots, x_n, \bar{x}_n\}$ où $x_1, \cdots, x_n$ sont les variables présentes dans φ. On note $v(l_{ij}) \in \{true, false\}$ l'évaluation du littéral l_{ij} de la disjonction $D_i = l_{i1} \vee l_{i2} \vee l_{i3}$ avec $i \in [1..m]$. Notre but est de représenter la formule $3CNF$ en arbres. Nous commençons par définir les étiquettes autorisées pour les nœuds des arbres et obtenons l'alphabet :

$$\Sigma = \{x_1, \cdots, x_n, d, v_0, v_{1,1}, v_{1,2}, v_{1,3}, \cdots, v_{i,1}, v_{i,2}, v_{i,3}, \cdots, v_{m,1}, v_{m,2}, v_{m,3}, \cdots, v_n, t, f\}$$

Nous pouvons maintenant construire l'ensemble de clauses E_φ comme représentation clausale de la formule φ :

$$E_\varphi = \{P_0, P_1, \cdots, P_m, N\} \subseteq \mathcal{L}_{fcns}^{T_\Sigma}$$

où P_0 est défini comme suit :

$$\begin{array}{rcl} P_0 & = & fc(X_0, X1),\ v_0(X_0), \\ & & fc(X_1, T_1),\ ns(T_1, F_1),\ x_1(X_1),\ t(T_1),\ f(F_1), \\ & & ns(X_1, X_2),\ fc(X_2, T_2),\ ns(T_2, F_2),\ x_2(X_2),\ t(T_2),\ f(F_2), \\ & & \cdots, \\ & & ns(X_{n-1}, X_n),\ fc(X_n, T_n),\ ns(T_n, F_n),\ x_n(X_n),\ t(T_n),\ f(F_n). \end{array}$$

et P_i, avec $i \in [1..m]$, est la représentation clausale de la disjonction D_i de φ. Plus précisément, une clause P_i contient les différentes évaluations possibles rendant la disjonction D_i vraie. Elle respecte le codage *first-child next-sibling* et représente une évaluation de la disjonction sous la forme d'un sous-arbre. S'agissant d'une formule $3SAT_{1in3}$, il n'existe que 3 évaluations possibles pour une même disjonction et donc 3 sous-arbres associés. Chacun a pour racine une variable $V_{1 \leqslant j \leqslant 3}$ et possède n fils. Chaque fils correspond à une variable de φ. On note $X_{j,k}$ le nœud avec pour père V_j et correspondant à la variable x_k. Ces 3 sous-arbres appartiennent à la même disjonction, leurs racines sont donc liées par une relation de fraternité. Chaque variable d'une disjonction peut prendre une valeur booléenne rendant la formule vraie ou fausse. Les autres variables peuvent être évaluées à vrai ou faux sans que cela ne perturbe l'évaluation de D_i. Afin de représenter cela, chaque nœud correspondant à une variable est la racine d'un sous arbre ayant deux fils. Le premier fils est annoté *true* et le second *false*. En fonction de l'évaluation de la disjonction, ces nœuds sont étiquetés ou non.

Pour une disjonction $D_i = (l_{i1} \vee l_{i2} \vee l_{i3})$ dont les littéraux $l_{i1},\ l_{i2}$ et l_{i3} utilisent respectivement les variables $x_a,\ x_b$ et x_c, si le premier littéral est vrai, on a alors les atomes suivants dans la représentation clausales de D_i :

$$v(l_{i1})(T_{1,x_a}),\ v(l_{i2})(F_{1,x_b}),\ v(l_{i3})(F_{1,x_c})$$

avec $v(l_{i1}) = t$ et $v(l_{i2}) = v(l_{i3}) = f$. Toute variable x_d différant de $x_a,\ x_b$ et x_c est représentée par les atomes $t(T_{1,x_d}),\ f(F_{1,x_d})$. On a ainsi un ensemble d'atomes représentant l'évaluation de D_i dont le premier littéral est évalué à vrai. On note $eval_{i,k}$ l'ensemble des atomes formant l'évaluation de D_i où le k$^{\text{e}}$ littéral est évalué à vrai. La sémantique de la clause P_i donnée, nous pouvons maintenant écrire cette dernière :

$$
\begin{aligned}
P_i \quad = \quad & fc(R,V_1),\ ns(V_1,V_2),\ ns(V_2,V_3),\\
& c(R),\\
& v_{i,1}(V1),\ fc(V_1,X_{1,1}),\ ns(X_{1,1},X_{1,2}),\ \cdots,\ ns(X_{1,n-1},X_{1,n}),\\
& x_1(X_{1,1}),\ x_2(X_{1,2}),\ \cdots,\ x_n(X_{1,n}),\\
& fc(X_{1,1},T_{1,1}),\ ns(T_{1,1},F_{1,1}),\cdots,\ fc(X_{1,n},T_{1,n}),\ ns(T_{1,n},F_{1,n}),\\
& eval_{i,1},\\
& v_{i,2}(V_2),\ fc(V_2,X_{2,1}),\ ns(X_{2,1},X_{2,2}),\ \cdots,\ ns(X_{2,n-1},X_{2,n}),\\
& x_1(X_{2,1}),\ x_2(X_{2,2}),\ \cdots,\ x_n(X_{2,n}),\\
& fc(X_{2,1},T_{2,1}),\ ns(T_{2,1},F_{2,1}),\ \cdots,fc(X_{2,n},T_{2,n}),\ ns(T_{2,n},F_{2,n}),\\
& eval_{i,2},\\
& v_{i,3}(V_3),\ fc(V_3,X_{3,1}),\ ns(X_{3,1},X_{3,2}),\ \cdots,\ ns(X_{3,n-1},X_{3,n}),\\
& x_1(X_{3,1}),\ x_2(X_{3,2}),\ \cdots,\ x_n(X_{3,n}),\\
& fc(X_{3,1},T_{3,1}),ns(T_{3,1},F_{3,1}),\cdots,fc(X_{3,n},T_{3,n}),ns(T_{3,n},F_{3,n})\\
& eval_{i,3}.
\end{aligned}
$$

L'ensemble des littéraux en **bleu** représente l'arbre illustrant l'évaluation de φ où le premier littéral de D_i est vrai. L'ensemble **vert** celui où le deuxième littéral est vrai et l'ensemble **rouge** celui où le troisième littéral est vrai. Enfin, les atomes en **noir** assurent la connexion entre les différentes évaluations afin de former un seul arbre.

Nous définissons maintenant l'unique exemple négatif, la clause N. Cet exemple contient les évaluations incomplètes et donc impossibles de la formule φ. Une évaluation est incomplète si l'une de ses variables n'est ni vraie ni fausse. Comme pour les exemples positifs, l'évaluation d'une disjonction est représentée par un sous-arbre. Cet exemple contient n sous-arbres, un pour chaque variable de φ, tels que les nœuds qui représentent l'évaluation sont non étiquetés. On a ainsi la clause N suivante :

$$
\begin{aligned}
N \quad = \quad & fc(R,V_1),\ ns(V_1,V_2),\ ns(V_2,V_3),\ \cdots,\ ns(V_{n-1},V_n),\\
& c(R),\\
& v_1(V_1),\ fc(V_1,X_{1,1}),\ ns(X_{1,1},X_{1,2}),\ \cdots,\ ns(X_{1,n-1},X_{1,n}),\\
& x_1(X_{1,1}),\ x_2(X_{1,2}),\ \cdots,x_n(X_{1,n}),\\
& fc(X_{1,1},T_{1,1}),\ ns(T_{1,1},F_{1,1}),\\
& fc(X_{1,2},T_{1,2}),\ ns(T_{1,2},F_{1,2}),\ t(T_{1,2}),\ f(F_{1,2})\\
& \cdots,\\
& fc(X_{1,n},T_{1,n}),\ ns(T_{1,n},F_{1,n}),\ t(T_{1,n}),\ f(F_{1,n})\\
& \cdots,\\
& v_k(V_k),\ fc(V_k,X_{k,1}),\ ns(X_{k,1},X_{k,2}),\cdots,\ ns(X_{k,n-1},X_{k,n}),\\
& x_1(X_{k,1}),\ x_2(X_{k,2}),\ \cdots,\ x_n(X_{k,n}),\\
& fc(X_{k,1},T_{k,1}),\ ns(T_{k,1},F_{k,1}),\ t(T_{k,1}),\ f(F_{k,1})\\
& \cdots,\\
& fc(X_{k,k-1},T_{k,k-1}),\ ns(T_{k,k-1},F_{k,k-1}),t(T_{k,k-1}),\ f(F_{k,k-1})\\
& fc(X_{k,n-1},T_{k,n-1}),\ ns(T_{k,n-1},F_{k,n-1}),\\
& fc(X_{k,k+1},T_{k,k+1}),\ ns(T_{k,k+1},F_{k,k+1}),t(T_{k,k+1}),\ f(F_{k,k+1})\\
& \cdots\\
& fc(X_{k,n},T_{k,n}),\ ns(T_{k,n},F_{k,n}),t(T_{k,n}),\ f(F_{k,n})\\
& \cdots,\\
& v_n(V_n),\ fc(V_n,X_{n,1}),\ ns(X_{n,1},X_{n,2}),\cdots,\ ns(X_{n,n-1},X_{n,n}),\\
& x_1(X_{n,1}),\ x_2(X_{n,2}),\ \cdots,\ x_n(X_{n,n}),\\
& fc(X_{n,1},T_{n,1}),\ ns(T_{n,1},F_{n,1}),\ t(T_{n,1}),\ f(F_{n,1})\\
& \cdots,\\
& fc(X_{n,n-1},T_{n,n-1}),\ ns(T_{n,n-1},F_{n,n-1}),\ t(T_{n,n-1}),\ f(F_{n,n-1})\\
& fc(X_{n,n},T_{n,n}),\ ns(T_{n,n},F_{n,n}).
\end{aligned}
$$

Chaque ensemble coloré de N correspond à une évaluation particulière de la formule φ où ont été omis les labels d'une variable donnée.

Illustrons cette construction avec l'exemple suivant : Soit la formule $\varphi = (x_1 \vee \bar{x}_2 \vee \bar{x}_3) \wedge (\bar{x}_2 \vee \bar{x}_3 \vee x_4) \in \mathit{3SAT_{1in3}}$. Les clauses de E sont, de par leur taille, illisibles. Ainsi, nous présentons à la figure 5.1 les représentations arborescentes de ces clauses.

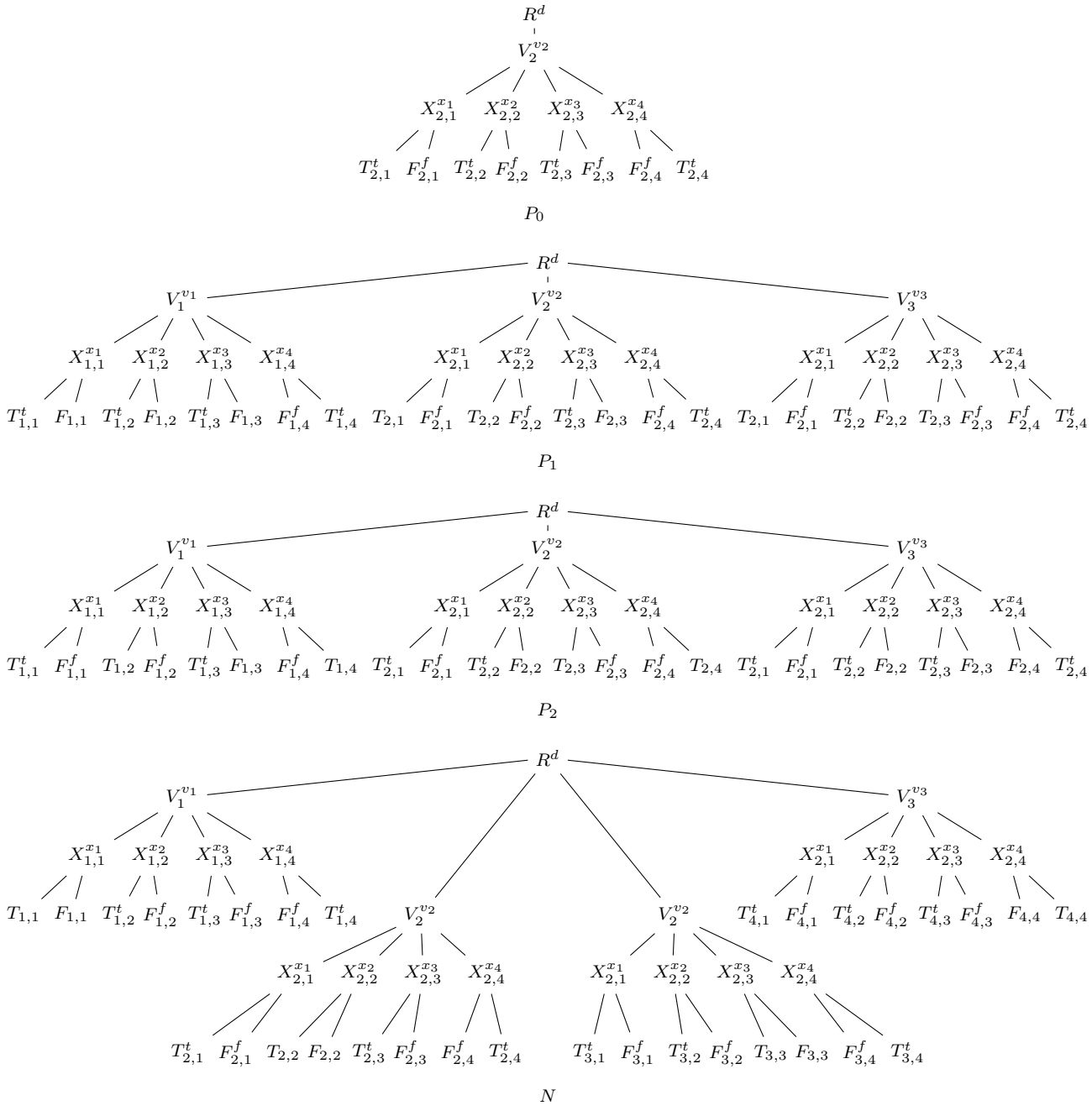


FIGURE 5.1 – Représentations arborescentes des exemples de l'ensemble $E_\varphi = \{P_0, P_1, P_2, N\}$ construit pour la formule $\varphi = (x_1 \vee \bar{x}_2 \vee \bar{x}_3) \wedge (\bar{x}_2 \vee \bar{x}_3 \vee x_4) \in \mathit{3SAT_{1in3}}$

Nous prouvons maintenant le lemme 18.

Preuve. Soit $\varphi = (l_{11} \vee l_{12} \vee l_{13}) \wedge ... \wedge (l_{m1} \vee l_{m2} \vee l_{m3}) \in 3CNF$ t.q. $l_{11}, ..., l_{m3} \in \{x_1, \bar{x}_1, ..., x_n, \bar{x}_n\}$ et E_φ l'ensemble de clauses décrit précédemment.

($\Leftarrow$) Commençons par prouver que si $E_\varphi \in CONS$ alors $\varphi \in 3SAT_{1in3}$. Nous supposons que $E_\varphi \in CONS$. On peut donc construire le moindre généralisé des clauses $P_0, \ldots, P_m$, noté $h = \mathsf{lgg}(P_0, \ldots, P_m)$. Cette clause h est une multi-clause constituée de plusieurs composantes car, parmi les exemples positifs, se trouve la clause suivante :

$$P_0 = fc(X_0, X1), v_0(X_0), x_1(X_1), fc(X_1, T_1), t(T_1), ns(T_1, F_1), f(F_1),$$
$$ns(X_1, X_2), x_2(X_2), fc(X_2, T_2), t(T_2), ns(T_2, F_2), f(F_2), ...,$$
$$ns(X_{n-1}, X_n), x_n(X_n), fc(X_n, T_n), t(T_n), ns(T_n, F_n), f(F_n).$$

Elle a pour rôle de séparer les sous-arbres correspondant aux évaluations en composantes indépendantes.

Grâce à la définition de $CONS$, on a $h \not\succeq N$. Il existe donc, au moins, une composante de h telle que chaque nœud X_i possède au moins un fils t ou f. Un exemple positif contient trois sous-arbres correspondant aux différentes évaluations de la conjonction. Chacun de ces sous-arbres est construit de sorte qu'exactement un littéral de la disjonction, dont il est issu, soit vrai. Enfin, la clause h est le moindre généralisé issu des exemples positifs. De ce fait, toute composante de h θ-subsume tout exemple positif. On peut donc construire une fonction d'évaluation $v : \{x_1, ..., x_n\} \to \{true, false\}$ telle que :

$$v(x_i) = \begin{cases} t & \text{si le nœud } X_i \text{ est suivi d'un noeud étiqueté par } t \\ f & \text{sinon.} \end{cases}$$

Cette fonction d'évaluation est construite de façon à ce qu'au plus un littéral de chaque clause soit satisfait. On a ainsi $\varphi \in 3SAT_{1in3}$.

($\Leftarrow$) Montrons maintenant que si $\varphi \in 3SAT_{1in3}$, alors $E_\varphi \in CONS$. Pour ce faire, on suppose que $\varphi \in 3SAT_{1in3}$. Il existe ainsi une évaluation v de φ telle qu'exactement un littéral de chaque clause C_i soit vérifié. La clause h est alors de la forme :

$$\{fc(X_0, X1), x_1(X_1), fc(X_1, T_1), ns(T_1, F_1),$$
$$ns(X_1, X_2), x_2(X_2), fc(X_2, T_2), ns(T_2, F_2), ...,$$
$$ns(X_{n-1}, X_n), x_n(X_n), fc(X_n, T_n), ns(T_n, F_n)\}$$

On lui ajoute les littéraux suivants :

$$\begin{cases} t(T_i) & \text{si } v(x_i) = t \\ f(F_i) & \text{sinon} \end{cases} \quad \text{avec } i \in [1..n].$$

Ainsi, $h \not\succeq N$ et $h \succeq P_i$. On a donc $E_\varphi \in CONS$.

Nous avons montré que si $\varphi \in 3SAT_{1in3}$, alors $E_\varphi \in CONS$ et que si $E_\varphi \in CONS$, alors $\varphi \in 3SAT_{1in3}$. On en déduit donc $E_\varphi \in CONS \iff \varphi \in 3SAT_{1in3}$. □

Grâce au lemme 18, nous pouvons expliciter la difficulté du problème $CONS$.

Théorème 23. *$CONS$ est NP-Complet.*

Preuve. Soit un ensemble d'exemples $E_\varphi \subseteq CONS$, on sait que l'on peut tester une hypothèse h en temps polynomial dans la norme de E_φ. De plus $\forall \varphi \in 3CNF.\ E_\varphi \in CONS \iff \varphi \in 3SAT_{1in3}$. Donc $CONS$ est NP-Complet. □

Ce problème de décision est NP-complet. La construction d'une clause distinguant les exemples positifs des négatifs est donc également NP-complet puisqu'elle permet d'en affirmer l'existence. Nous en déduisons le résultat suivant pour l'identification à la limite.

Théorème 24. *Le langage $\mathcal{L}^{\Sigma}_{fcns}$ n'est pas identifiable polynomialement à la limite en temps et en données (avec correction) par positifs et négatifs issus de $\mathcal{L}^{T_\Sigma}_{fcns}$ à l'aide du moindre généralisé.*

Ce théorème traduit la difficulté de certaines instances de notre problème. Cependant, comme souvent dans les problèmes NP-Complet, toute instance de ce problème n'est pas difficile. Afin de vérifier la fréquence d'apparition d'instances difficiles dans le cas d'exemples issus de problèmes réels, nous réalisons des expériences dans la section 5.6. Elles vont nous permettre de mieux qualifier la difficulté de l'apprentissage vis-à-vis de données réelles.

5.6 Expérimentations

Dans ce chapitre, plusieurs algorithmes polynomiaux de moindre généralisation et de θ-subsomption pour différents langages de clauses ont été proposés. Nous allons naturellement en tirer profit pour réaliser nos apprentissages pour la classification de documents XML. Ces expériences ont pour but d'étudier l'efficacité d'un apprentissage de cette nature en utilisant nos représentations clausales et nos algorithmes. Nous utilisons les critères d'évaluation définis lors du chapitre 3 à la section 3.5 que sont la F-mesure, le rappel et la précision. Nous étudions également l'évolution de la taille des moindres généralisés obtenus lors de nos apprentissages. Les documents XML utilisés pour ces expériences sont issus de trois corpus distincts, un artificiel et deux réels. Nous sommes donc amenés à réaliser trois apprentissages différents, un pour chaque corpus. Passons maintenant à leur présentation.

5.6.1 Corpus de documents

Les expériences présentées portent sur trois corpus différents. Le premier traite de résultats footballistiques et correspond à notre corpus artificiel. Le second comporte des pages web de chercheurs en informatique. Enfin, le troisième contient des documents issus de Wikipedia. Les documents XML des deux derniers corpus ont été extraits du web et correspondent à des documents issus d'Internet.

Afin de réaliser l'apprentissage, chaque corpus est divisé par validation croisée 5 fois. Il y a donc cinq apprentissages à effectuer par corpus, chacun sur 80% des données. Les 20% restants sont utilisés en test. Nous répétons ces opérations 10 fois et faisons la moyenne des résultats obtenus. Plus de détails sur la validation croisée se trouvent dans le chapitre 3 à la section 3.5. Nous présentons maintenant ces différents corpus.

Données footballistiques

Ces données constituent un corpus sur les résultats sportifs d'équipes françaises de football. Y sont décrits les différentes rencontres ainsi que les résultats sportifs associés aux championnats. Ce corpus de documents a été généré artificiellement à partir d'une base de données. Il contient au total 170 pages XHTML, chacune est associée à une unique classe. Les documents sont repartis en 9 classes, chacune contient en moyenne 19 documents. Un arbre représentant l'un de ces documents contient en moyenne 296 nœuds, a une profondeur de 7 ainsi qu'une largeur de 6.

Chaque classe correspond à une manière de représenter des informations dans une structure de documents particulière. Les différentes classes contiennent ainsi les mêmes données

représentées différemment. La figure 5.2 nous donne un aperçu des documents présents dans ce corpus. On a ainsi plusieurs classes contenant des documents caractérisés par leur structure d'information :

Classe L_0 : Cette classe contient des tableaux verticaux, chaque descripteur de match se trouve sur la même colonne. Ainsi, toutes les informations d'un match sont présentes sur une même ligne.

Classe L_1 : Cette classe contient des tableaux horizontaux, chaque descripteur de match se trouve sur la même ligne. Ainsi, toutes les informations d'un match sont présentes sur une même colonne.

Classe L_2 : Cette classe contient des tableaux horizontaux à deux colonnes telle qu'une ligne ne contient qu'un descripteur d'un match. Tous les descripteurs d'un même match se suivent les uns en dessous des autres dans le même ordre. Les informations d'un match sont ainsi représentées par un nombre fixe de lignes de la première colonne.

Classe L_3 : Cette classe contient une liste non ordonnée de listes non ordonnées de descripteurs de matchs. Chaque sous-liste regroupe les descripteurs, présentés dans le même ordre, d'un même match.

Classe L_4 et L_5 : Cette classe contient respectivement une liste ordonnée et une non ordonnée de descripteurs de matchs. Un match est ainsi décrit sur plusieurs lignes, chaque descripteur occupe une ligne. Leur ordre est toujours le même.

Classe L_6 : Cette classe contient une liste non ordonnée de descripteurs de matchs. Un match est ainsi décrit sur une seule ligne contenant tous ses descripteurs les uns derrière les autres.

Classe L_7 : Cette classe contient une liste de définitions, chaque élément de cette liste est un descripteur d'un match auquel est associée une valeur. Un match est donc décrit par plusieurs éléments de la liste, ces éléments respectent toujours le même ordre.

Classe L_8 : Cette classe contient une liste de définitions, chaque élément de cette liste contient tous les descripteurs d'un même match, organisés de la même manière.

La nombre moyen de nœuds d'un document de ce corpus est de 300. Ainsi, les clauses des codages $\mathcal{L}_{pc}^{T_\Sigma}$ et $\mathcal{L}_{fcns}^{T_\Sigma}$ vont contenir entre 300 et au moins 600 littéraux. Or, la taille d'un moindre généralisé non réduit de deux clauses est le produit des tailles de ces clauses. Nous allons donc manipuler des clauses non réduites d'une taille avoisinant les 100 000 littéraux. Certains de nos codages permettent la construction de moindres généralisés exponentiels dans le nombre de clauses, il convient donc d'étudier la question de la taille du moindre généralisé réduit.

Nos travaux sont destinés à l'apprentissage de la classification de documents XML. Ainsi, l'objectif de l'apprentissage pour ce corpus est d'apprendre un ensemble de clauses capable d'assigner chacun de ses documents à une unique classe.

Pages personnelles

Le second corpus de documents contient également des pages XHTML. Ces documents XML correspondent aux pages web personnelles de chercheurs en informatique. Contraire-

Club	League	Season	Round	Date	Rank	Points	Wins	Draws	Losses	Goals Scored	Goals Conceded	Goal Difference
Marseille	1	2002-2003	20	12/18/2002	1	35	10	5	5	21	19	2
Bordeaux	1	2002-2003	13	11/02/2002	11	17	4	5	4	13	12	1
Bastia	1	2002-2003	11	10/19/2002	12	14	4	2	5	13	15	-2
Sedan	1	1999-2000	19	-	6	28	8	4	7	27	28	-1
Istres	2	2003-2004	24	02/14/2004	1	47	14	5	5	28	11	17
Lens	1	1999-2000	3	-	4	6	2	0	1	2	1	1
Strasbourg	1	2002-2003	15	11/16/2002	10	22	6	4	5	20	25	-5
Montpellier	1	1999-2000	19	-	18	14	3	5	11	20	31	-11
Guingamp	1	2001-2002	23	30th January 2002	15	24	6	6	11	22	39	-17

Classe L_0

Club	Sochaux	Bastia	Toulouse	Montpellier	Bastia	Sochaux
League	1	1	1	2	1	2
Season	2003-2004	1999-2000	2003-2004	2000-2001	2003-2004	1999-2000
Round	11	5	18	30	32	19
Date	10/25/2003	-	12/13/2003	-	04/10/2004	-
Rank	10	10	20	2	13	7
Points	15	7	12	57	38	28
Wins	4	2	2	15	9	8
Draws	3	1	6	12	11	4
Losses	4	2	10	3	12	7
Goals Scored	13	8	13	39	33	22
Goals Conceded	12	9	29	19	39	20
Goal Difference	1	-1	-16	20	-6	2

Classe L_1

Sochaux
League 1, 2003-2004, Round 11 (10/25/2003): Rank 10, 15 Points, 4-3-4, 13-12=1.
Bastia
League 1, 1999-2000, Round 5 (-): Rank 10, 7 Points, 2-1-2, 8-9=-1.
Toulouse
League 1, 2003-2004, Round 18 (12/13/2003): Rank 20, 12 Points, 2-6-10, 13-29=-16.
Montpellier
League 2, 2000-2001, Round 30 (-): Rank 2, 57 Points, 15-12-3, 39-19=20.
Bastia
League 1, 2003-2004, Round 32 (04/10/2004): Rank 13, 38 Points, 9-11-12, 33-39=-6.
Sochaux
League 2, 1999-2000, Round 19 (-): Rank 7, 28 Points, 8-4-7, 22-20=2.

Classe L_8

+ Sochaux, League 1, 2003-2004, Round 11 (10/25/2003): Rank 10, 15 Points, 4-3-4, 13-12=1.
+ Bastia, League 1, 1999-2000, Round 5 (-): Rank 10, 7 Points, 2-1-2, 8-9=-1.
+ Toulouse, League 1, 2003-2004, Round 18 (12/13/2003): Rank 20, 12 Points, 2-6-10, 13-29=-16.
+ Montpellier, League 2, 2000-2001, Round 30 (-): Rank 2, 57 Points, 15-12-3, 39-19=20.
+ Bastia, League 1, 2003-2004, Round 32 (04/10/2004): Rank 13, 38 Points, 9-11-12, 33-39=-6.
+ Sochaux, League 2, 1999-2000, Round 19 (-): Rank 7, 28 Points, 8-4-7, 22-20=2.

Classe L_6

Club	Sochaux
League	1
Season	2003-2004
Round	11
Date	10/25/2003
Rank	10
Points	15
Wins	4
Draws	3
Losses	4
Goals Scored	13
Goals Conceded	12
Goal Difference	1
Club	Bastia
League	1
Season	1999-2000
Round	5
Date	-
Rank	10
Points	7
Wins	2
Draws	1
Losses	2
Goals Scored	8
Goals Conceded	9
Goal Difference	-1
Club	Toulouse
League	1
Season	2003-2004

Classe L_2

+ Tuple 1
  + Club: Marseille
  + League: 1
  + Season: 2002-2003
  + Round: 20
  + Date: 12/18/2002
  + Rank: 1
  + Points: 35
  + Wins: 10
  + Draws: 5
  + Losses: 5
  + Goals Scored: 21
  + Goals Conceded: 19
  + Goal Difference: 2
+ Tuple 2
  + Club: Bordeaux
  + League: 1
  + Season: 2002-2003
  + Round: 13
  + Date: 11/02/2002
  + Rank: 11
  + Points: 17
  + Wins: 4
  + Draws: 5
  + Losses: 4

Classe L_3

+ Sochaux
+ League: 1
+ 2003-2004
+ Round: 11
+ 10/25/2003
+ Rank: 10
+ Points: 15
+ Wins: 4
+ Draws: 3
+ Losses: 4
+ Goals Scored: 13
+ Goals Conceded: 12
+ Goal Difference: 1
+ Bastia
+ League: 1
+ 1999-2000
+ Round: 5
+ -
+ Rank: 10
+ Points: 7
+ Wins: 2
+ Draws: 1

Classe L_4 et L_5

Club
Sochaux
League
1
Season
2003-2004
Round
11
Date
10/25/2003
Rank
10
Points
15
Wins
4
Draws
3
Losses
4
Goals Scored
13
Goals Conceded
12
Goal Difference
1
Club
Bastia
League
1
Season
1999-2000

Classe L_7

FIGURE 5.2 – Exemples de documents XML présents dans le corpus footballistique

ment au corpus précédent, ces pages n'ont pas toutes été générées artificiellement. Ainsi, certaines pages ont été écrites par des humains, tandis que d'autres correspondent à des pages issues de CMS (Content Management System) et ont été produites automatiquement. Les documents de ce corpus se rapprochent du type de documents pouvant être obtenus à partir d'internet.

Donnons maintenant quelques statistiques de ce corpus. Son nombre de pages est de 841. Chaque page est associée à une unique classe correspondant au nom du propriétaire de cette page. Il y a au total 15 classes contenant chacune en moyenne 53 documents XHTML. En moyenne, ces arbres ont une profondeur de 8, une largeur de 3 et 220 nœuds. Cela signifie que nos clauses-exemples vont contenir plusieurs centaines de littéraux et, qu'avant réduction, nous allons manipuler des clauses généralisées de plusieurs milliers de littéraux. Nous remarquons que certaines personnes utilisent le même CMS, ainsi certaines pages issues de différentes personnes auront d'importantes similarités.

Pour ce corpus de documents, la tâche de classification consistera alors à retrouver le propriétaire de pages web à partir de la structure de ces documents. Nous la pensons possible en supposant que chaque individu possède des habitudes personnelles. Ainsi, il devrait en être de même pour l'écriture de pages web. Cependant, la création de pages web est une activité fortement normée, notamment par le W3C. De plus, ce corpus possède des pages générées par des CMS. Nous nous attendons donc à ce que l'apprentissage soit difficile.

Corpus wikipedia

Le dernier corpus de documents a été constitué pour le challenge INEX 2010 [Vries 2011, Alexander 2011]. Il contient 20697 documents XML extraits de la base de données des documents XML anglophones de Wikipedia. Les classes de ce corpus sont des thèmes décrivant les documents. On retrouve notamment des classes comme : `American Musicians`, `Art`, `Belief`. Ce corpus est initialement défini pour une tâche de classification multi-labels, ainsi un exemple peut être associé à une ou plusieurs classes. Afin de pouvoir exploiter ce corpus, nous faisons le choix de ne considérer qu'une classe par document (la première par ordre alphabétique comme fournie dans le challenge INEX). Chaque classe correspond alors à un thème sur lequel porte le document.

Donnons maintenant quelques statistiques sur ces documents. Ils sont classés en 29 catégories. Chacune contient en moyenne 714 documents. Un arbre XML a, dans ce corpus, une profondeur moyenne de 21 nœuds, une largeur moyenne de 2 nœuds et contient généralement 496 nœuds. Les tags peuvent être à la fois des éléments sémantiques comme : `adult`, `athlete`, `idea`, `autocracy`, ... ou des éléments de structure comme : `article`, `title` `imdb_id`, `Image_capt`, ... Ainsi, les balises des documents de ce corpus mélangent à la fois les éléments sémantiques et de structure. Le but de l'apprentissage est alors d'apprendre des hypothèses qui associent chaque document de ce corpus à sa classe.

Les différents corpus présentés, intéressons-nous à leurs apprentissages ainsi qu'à l'étude de la taille des moindres généralisés produits.

5.6.2 Apprentissage par moindre généralisation

Nous réalisons en premier lieu l'apprentissage d'hypothèses pour la classification de documents XML à partir de la moindre généralisation et mettons en concurrence les différents codages présentés. Nous portons notre attention sur deux points : la taille du moindre généralisé construit et l'évaluation des apprentissages réalisés. Cette dernière est réalisée à l'aide des scores de *rappel*, *précision* et *F-mesure* dont les définitions sont données aux chapitre 3 section 3.5.

Taille du moindre généralisé

Le but des expériences proposées est d'observer l'évolution de la taille des moindres généralisées ainsi que la fréquence de leurs explosions lorsque cela est possible. La taille d'un moindre généralisé de clauses SOP ne peut rester que constante ou diminuer au fur et à mesure des généralisations. À l'inverse, celle d'un moindre généralisé de clauses FDP est susceptible de connaître une augmentation la rendant exponentielle dans le nombre de clauses généralisées.

Nous commençons avec le corpus footballistique. La figure 5.3 illustre l'évolution de la la taille des moindres généralisés issus de clauses SOP après une quarantaine d'exemples.

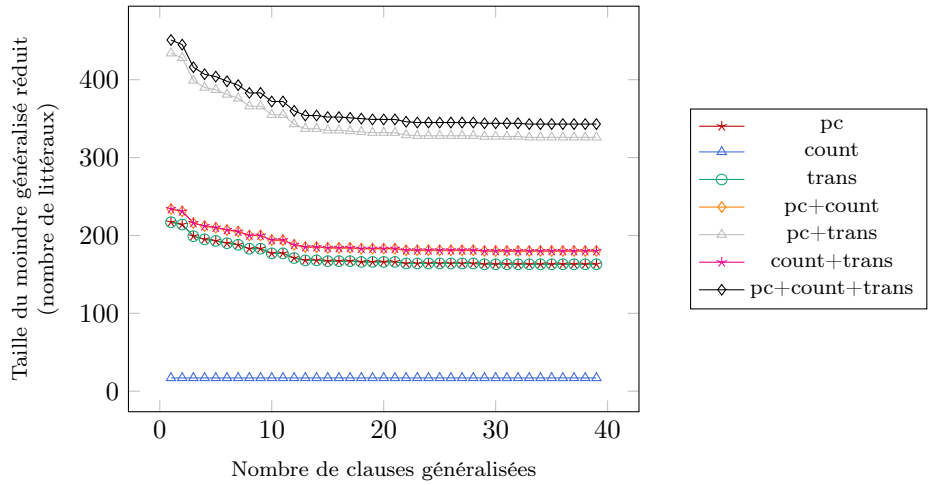


FIGURE 5.3 – Évolution des moindres généralisés du corpus 1 pour les codages SOP

Les courbes observées sont obtenues en moyennant la taille des moindres généralisés toutes classes et tous runs confondus en fonction du nombre d'exemples. On y observe principalement deux nuances :

1. La première correspond à la courbe du codage *count*. Le nombre de littéraux d'une clause de ce codage est le nombre de labels de l'arbre dont il est issu. Les moindres généralisés de clauses de ce langage ont une taille globalement constante. Cela signifie que chaque label de l'alphabet apparait au moins une fois dans chaque exemple.

2. Tous les autres codages appartiennent à la seconde nuance. L'écart entre les tailles des moindres généralisés de ces codages est constant. Il s'explique par la combinaison de langages SOP ainsi que leur nombre. La taille des moindres généralisés d'un de ces codages diminue au cours des premiers exemples puis tend à rester constante ou continue à décroître légèrement.

Comme attendu théoriquement, nous observons que la taille des moindres généralisés de ces codages diminue au cours des premiers exemples puis tend à rester constante. Ainsi, les premières généralisations élaguent les informations qui différent entre les exemples.

La figure 5.4 illustre l'évolution de la taille des moindres généralisés issus de clauses MQD après une quarantaine d'exemples. Comme vu précédemment, les courbes sont obtenues en moyennant les tailles des moindres généralisés toutes classes et tous runs confondus.

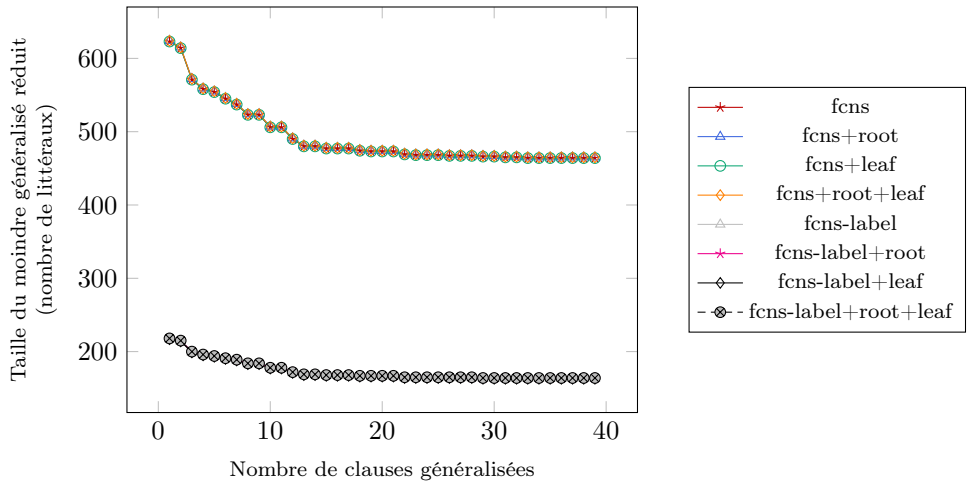


FIGURE 5.4 – Évolution des moindres généralisés du corpus 1 pour les codages MQD

Nous regroupons les courbes en deux catégories. La première contient les courbes des codages avec labels dont les tailles sont les plus élevées : *fcns*, *fcns+root*, *fcns+leaf* et *fcns+root+leaf*. La seconde possède les courbes des codages sans label : *fcns-label*, *fcns-label+root*, *fcns-label+leaf* et *fcns-label+root+leaf*. La taille des moindres généralisés d'une de ces courbes diminue au cours des dix premiers exemples puis devient constante comme précédemment. Ces moindres généralisés ne présentent pas d'explosion de leur taille dans le nombre d'exemples. La présence des labels dans un codage MQD basé sur *fcns* augmente significativement la taille des clauses contrairement à l'ajout de prédicats indiquant les feuilles ou la racine d'un arbre.

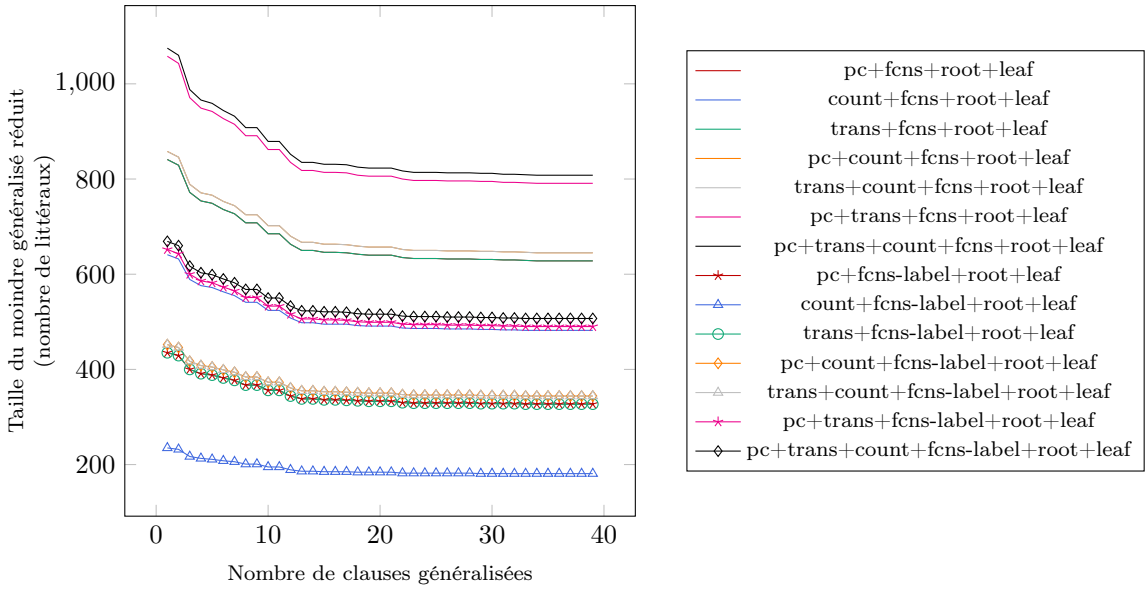


FIGURE 5.5 – Évolution des moindres généralisés du corpus 1 pour les codages SOP+MQD

Enfin, la figure 5.5 présente l'évolution de la taille des moindres généralisés de clauses de langages couplant SOP et MQD pour le corpus de foot. Sans surprise, la taille d'un

moindre généralisé associant un codage SOP et un codage MQD est la somme des tailles des moindres généralisés des clauses issues de chacun de ces codages. Comme précédemment, aucune explosion de leur taille en fonction du nombre d'exemples généralisés n'est observée. Ainsi, malgré un résultat présentant l'existence d'une telle explosion, nous ne le constatons pas expérimentalement pour notre premier corpus de documents.

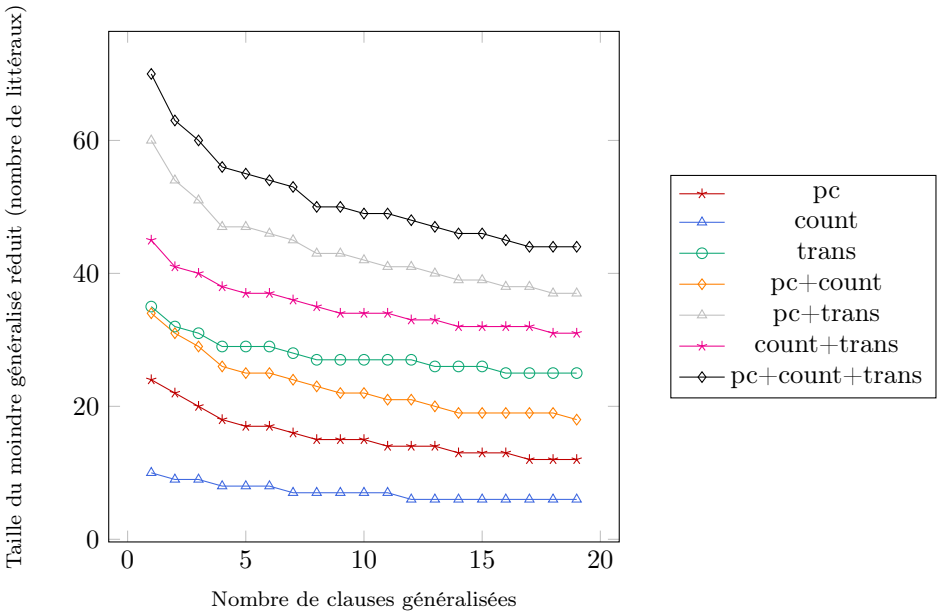


FIGURE 5.6 – Évolution des moindres généralisés du corpus 2 pour les codages SOP

Étudions maintenant l'évolution de la taille de moindres généralisés d'exemples du corpus 2 portant sur les pages web de chercheurs. Les moindres généralisés pour les différents codages SOP de la figure 5.6 ont un comportement similaire à ceux de la figure 5.3. Ainsi, la taille de ces derniers décroît au cours des premières généralisations puis tend à rester constante. La figure 5.7 présente une légère explosion de la taille des moindres généralisés au cours de la généralisation des cinq premiers exemples. Cette explosion n'est visible que pour les codages autorisant les labels des nœuds d'un arbre. Ainsi, nous voyons clairement que la présence des labels pénalise d'un point de vue computationnel l'apprentissage. Ce même phénomène est observable à la figure 5.4 pour le corpus 1. Ce constat n'est pas étonnant puisque nous avons montré au cours de ce chapitre que la présence de labels dans ces clauses permet une explosion de la taille du moindre généralisé de ces clauses. La taille de ces moindres généralisés tend ensuite à diminuer puis à devenir constante. Ainsi, l'explosion combinatoire prévue théoriquement a eu lieu au cours de la généralisation des premiers exemples de ce corpus. Continuons avec la figure 5.8. Comme précédemment, la taille d'un moindre généralisé associant un codage SOP et un codage MQD est la somme des tailles des moindres généralisés des clauses issues de chacun de ces codages. L'explosion de la taille des moindres généralisés persiste donc pour les codages avec labels.

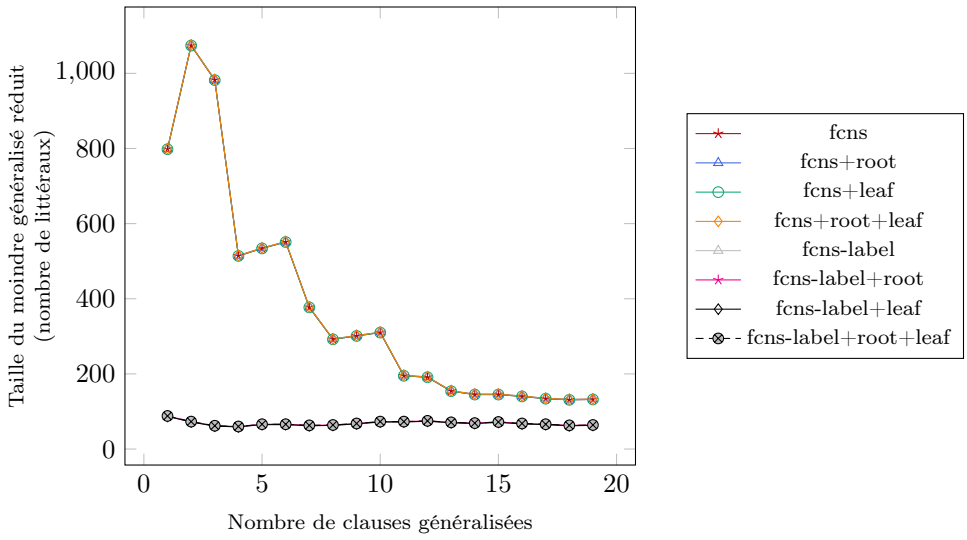


FIGURE 5.7 – Évolution des moindres généralisés du corpus 2 pour les codages MQD

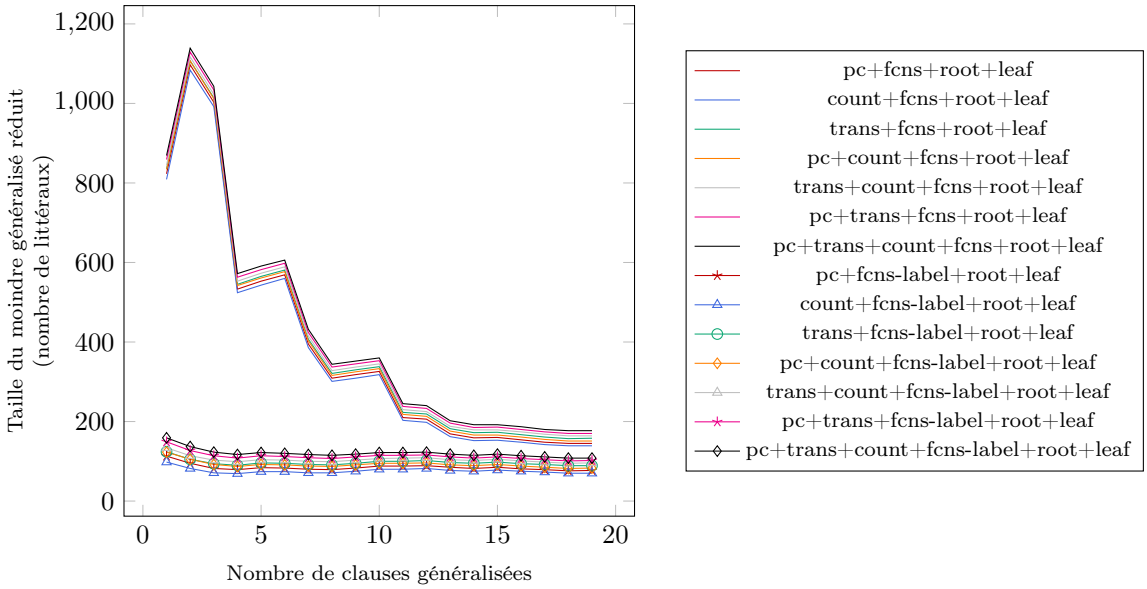


FIGURE 5.8 – Évolution des moindres généralisés du corpus 2 pour les codages SOP+MQD

Terminons avec l'étude de la taille des moindres généralisés produits par un apprentissage à base de moindre généralisé à partir de documents du corpus INEX. Ce corpus est conséquent. Pour cette raison nous n'étudions que les codages SOP dont la complexité est peu élevée ainsi que quelques codages MQD présentant une absence d'explosion de la taille des moindres généralisés dans le nombre d'exemples. Le temps de calcul nécessaire à l'obtention des graphes dépassant l'ordre du mois pour les codages MQD avec labels, cela nous confirme l'existence d'une explosion de la taille du moindre généralisé et justifie notre décision de les exclure pour le corpus INEX.

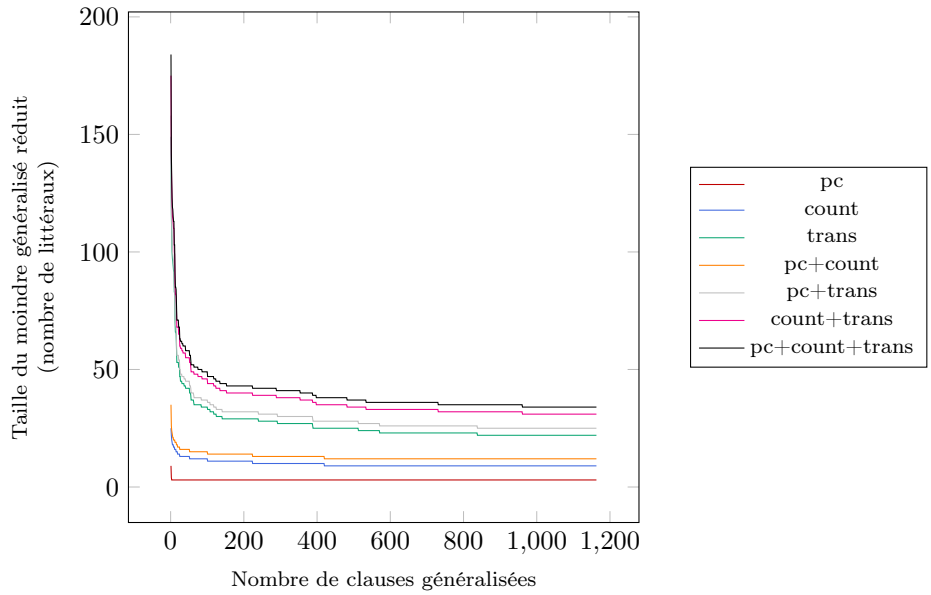


FIGURE 5.9 – Évolution des moindres généralisés du corpus 3 pour les codages SOP

La figure 5.9 présente la taille des moindres généralisés obtenus avec les différents codages SOP pour ce corpus. Le comportement des clauses correspond à ce qui est attendu et observé avec les corpus précédents. En effet, la taille des moindres généralisés diminue au cours des premiers exemples puis se stabilise afin de devenir constante.

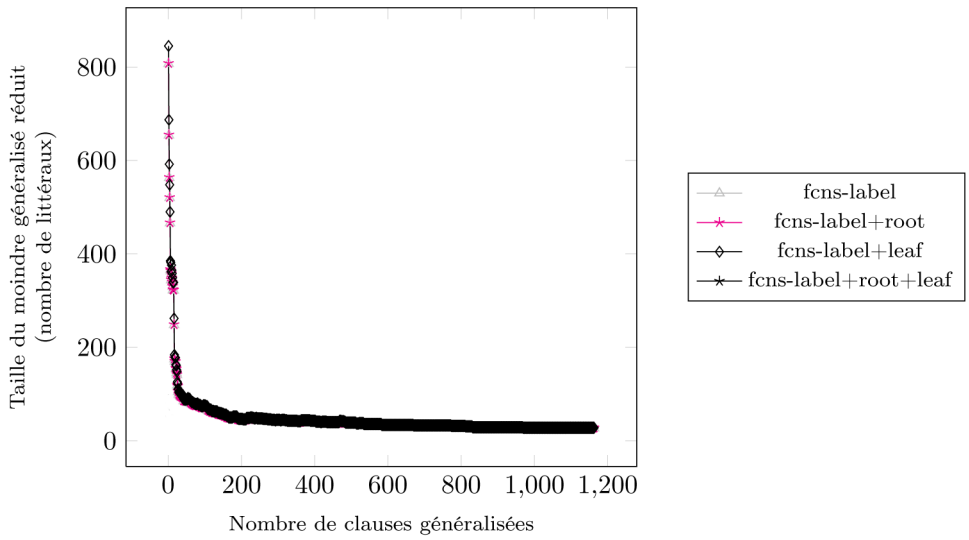


FIGURE 5.10 – Évolution des moindres généralisés du corpus 3 pour les codages MQD sans label

La figure 5.10 présente la taille des moindres généralisés obtenus avec les codages MQD sans label. Le comportement de ces codages pour ce corpus est similaire à ceux observés précédemment pour les autres corpus. Quel que soit le codage, nous n'observons donc pas d'explosion de la taille des moindres généralisés dans le nombre d'exemples. En effet, la

taille des moindres généralisés diminue au cours des 200 premiers exemples puis tend à rester constante.

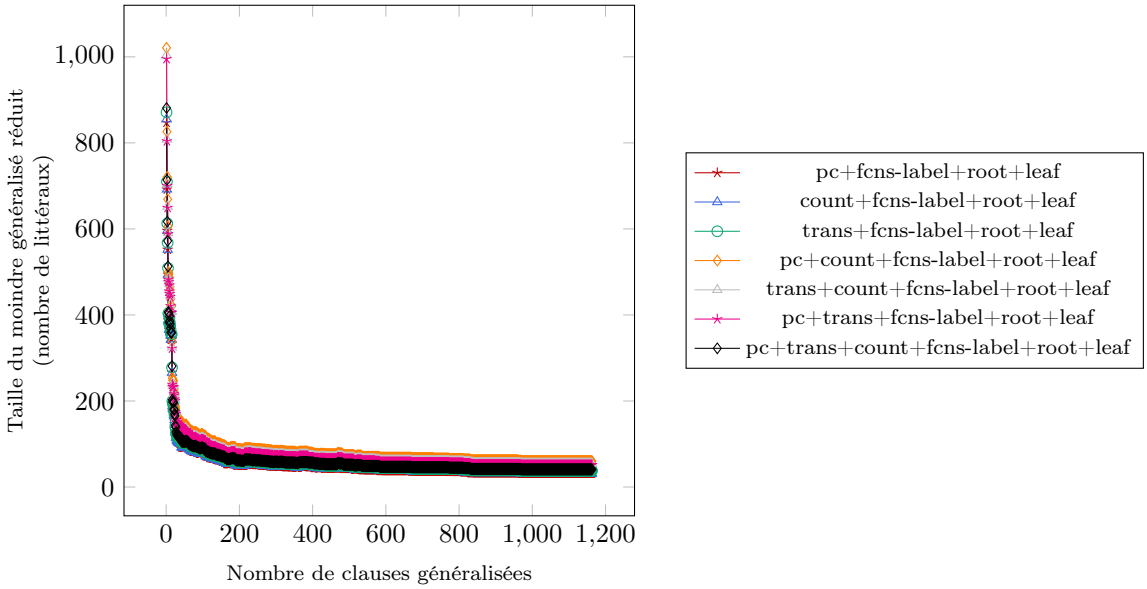


FIGURE 5.11 – Évolution des moindres généralisés du corpus 3 pour les codages SOP+MQD sans label

Nous terminons avec la figure 5.11 présentant l'évolution de la taille des moindres généralisés pour des codages couplant SOP et MQD sans label. À l'instar des autres corpus, nous n'observons pas d'explosion de la taille des moindres généralisés dans le nombre d'exemples. La taille de ces derniers diminue fortement au cours des premières généralisations puis se stabilise.

Vis-à-vis du langage $\mathcal{L}^{T_\Sigma}_{nested}$, nous n'avons pu présenter de graphique d'évolution de la taille des moindres généralisés de ces clauses. Lors de nos expériences, le temps de calcul du moindre généralisé de clauses de ce langage s'est avéré élevé, et ce, indépendamment du corpus étudié. Ainsi, après plus de 3 mois, aucun résultat n'a été obtenu quelque soit le corpus. Ce constat confirme le résultat théorique d'explosion de la taille des moindres généralisés dans le nombre d'exemples généralisés.

Dans [Kietz 1993], il est établi que, dans le cadre des clauses de Horn, une hypothèse moindre généralisée peut croître de manière exponentielle avec le nombre d'exemples. Nous avons montré qu'une explosion exponentielle est théoriquement possible au sein de notre langage d'arbres $\mathcal{L}_{fcns}$. Cependant, nous avons observé expérimentalement que la taille de ces clauses ne croît qu'avec les premiers exemples généralisés et décroît ensuite. Enfin, concernant le codage SOP, il a été observé une diminution de la taille du moindre généralisé comme prévue théoriquement. La taille des moindres généralisés construits ne semble pas pénaliser les apprentissages envisagés, nous pouvons donc maintenant considérer l'apprentissage des classes de ces corpus et évaluer leurs qualités.

Évaluation de l'apprentissage

Donnons maintenant les résultats d'apprentissage pour nos différents corpus. Ils permettent de quantifier l'importance d'un codage choisi pour représenter les documents XML. Ainsi, nous allons comparer les codages SOP, MQD et les couplages de ces langages. Nous

présentons ci-dessous différentes tables récapitulant les résultats de rappel, précision et F-mesure obtenus sur ces corpus de documents. Sont indiqués la taille moyenne d'une hypothèse apprise, son nombre de composantes connectées, son nombre d'atomes toutes composantes confondues, les mesures de rappel, de précision et de F-mesure ainsi que le temps d'exécution. Ces résultats sont des moyennes obtenues toutes classes et tous runs confondus en fonction du nombre d'exemples.

Nous commençons avec les résultats issus du corpus 1.

	pc	**count**	**trans**	**pc +trans**	**count +trans**	**pc +count**	**pc +count +trans**
Temps d'exécution (j :h :m :s :ms)	13	12	14	21	22	22	29
Rappel	1.0	1.0	1.0	1.0	1.0	1.0	1.0
Précision	1.0	1.0	1.0	1.0	1.0	1.0	1.0
F-mesure	1.0	1.0	1.0	1.0	1.0	1.0	1.0
Nombre total d'atomes par clause	163.0	17.0	163.0	326.0	180.0	180.0	343.0
Nombres de composantes	163.0	17.0	163.0	326.0	180.0	180.0	343.0
Nombre total d'hypothèses par classe	1.0	1.0	1.0	1.0	1.0	1.0	1.0

TABLE 5.1 – Résultats d'apprentissage des codages SOP pour le corpus 1

Ce corpus ne présente pas de véritable difficulté pour l'apprentissage d'une hypothèse par moindre généralisé. En effet, quelque soit le codage utilisé, la F-mesure, le rappel et la précision sont à 1 comme constaté dans les tables 5.1, 5.2 et 5.3. Ainsi, les hypothèses apprises reconnaissent exactement chaque classe d'une manière complète et correcte.

	fcns	**fcns +leaf**	**fcns +root**	**fcns +root +leaf**	**fcns -label**	**fcns -label +leaf**	**fcns -label +root**	**fcns -label +root +leaf**
Temps d'exécution (j :h :m :s :ms)	2 :43 :709	2 :47 :439	2 :40 :24	2 :15 :10	35 :357	35 :700	35 :344	35 :5
Rappel	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0
Précision	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0
F-mesure	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0
Nombre total d'atomes par clause	464.0	464.0	465.0	465.0	163.0	163.0	164.0	164.0
Nombres de composantes	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0
Nombre total d'hypothèses par classe	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0

TABLE 5.2 – Résultats d'apprentissage des codages MQD pour le corpus 1

Notons que la taille des hypothèses est proportionnelle à celles des documents codés.

Le nombre de littéraux supprimés lors d'un apprentissage est donc faible. L'emploi d'un codage SOP permet d'obtenir un résultat 1000 à 10000 fois plus rapidement qu'avec un codage MQD ou SOP+MQD. Ainsi, pour une F-mesure équivalente, l'emploi d'un codage SOP est à favoriser au détriment d'un codage MQD pour ce corpus. Reste à vérifier si cela est également vrai pour les corpus suivants.

	pc +fcns +root +leaf	**count +fcns +root +leaf**	**trans +fcns +root +leaf**	**pc +trans +fcns +root +leaf**	**pc +count +fcns +root +leaf**	**trans +count +fcns +root +leaf**	**pc +trans +count +fcns +root +leaf**
Temps d'exécution (j :h :m :s :ms)	2 :13 :327	2 :44 :48	2 :43 :444	2 :45 :449	2 :44 :269	2 :14 :171	2 :42 :638
Rappel	1.0	1.0	1.0	1.0	1.0	1.0	1.0
Précision	1.0	1.0	1.0	1.0	1.0	1.0	1.0
F-mesure	1.0	1.0	1.0	1.0	1.0	1.0	1.0
Nombre total d'atomes par clause	628.0	482.0	628.0	791.0	645.0	645.0	808.0
Nombres de composantes	164.0	18.0	164.0	327.0	181.0	181.0	344.0
Nombre total d'hypothèses par classe	1.0	1.0	1.0	1.0	1.0	1.0	1.0
	pc +fcns -label +root +leaf	**count +fcns -label +root +leaf**	**trans +fcns -label +root +leaf**	**pc +trans +fcns -label +root +leaf**	**pc +count +fcns -label +root +leaf**	**trans +count +fcns -label +root +leaf**	**pc +trans +count +fcns -label +root +leaf**
Temps d'exécution (j :h :m :s :ms)	35 :403	34 :552	35 :518	34 :198	34 :300	34 :730	31 :65
Rappel	1.0	1.0	1.0	1.0	1.0	1.0	1.0
Précision	1.0	1.0	1.0	1.0	1.0	1.0	1.0
F-mesure	1.0	1.0	1.0	1.0	1.0	1.0	1.0
Nombre total d'atomes par clause	327.0	181.0	327.0	490.0	344.0	344.0	507.0
Nombres de composantes	164.0	18.0	164.0	327.0	181.0	181.0	344.0
Nombre total d'hypothèses par classe	1.0	1.0	1.0	1.0	1.0	1.0	1.0

TABLE 5.3 – Résultats d'apprentissage des codages SOP+MQD pour le corpus 1

Présentons maintenant les résultats expérimentaux des tables 5.4, 5.5 et 5.6 obtenus pour le corpus 2 des pages personnelles de chercheurs.

	pc	count	trans	pc +trans	count +trans	pc +count	pc +count +trans
Temps d'exécution (j :h :m :s :ms)	2	4	5	5	7	5	6
Rappel	0.91	0.87	0.92	0.91	0.87	0.86	0.86
Précision	1.0	1.0	1.0	1.0	1.0	1.0	1.0
F-mesure	0.947	0.923	0.954	0.947	0.923	0.915	0.915
Nombre total d'atomes par clause	12.5	6.36	25.28	37.78	31.64	18.86	44.14
Nombres de composantes	12.5	6.36	25.28	37.78	31.64	18.86	44.14
Nombre total d'hypothèses par classe	1.0	1.0	1.0	1.0	1.0	1.0	1.0

TABLE 5.4 – Résultats d'apprentissage des codages SOP pour le corpus 2

	fcns	fcns +leaf	fcns +root	fcns +root +leaf	fcns -label	fcns -label +leaf	fcns -label +root	fcns -label +root +leaf
Temps d'exécution (j :h :m :s :ms)	1 :20 :136	1 :0 :606	1 :21 :550	1 :23 :907	1 :570	1 :696	1 :671	1 :740
Rappel	0.635	0.635	0.635	0.635	0.77	0.77	0.77	0.77
Précision	0.98	0.98	0.98	0.98	0.653	0.653	0.653	0.653
F-mesure	0.754	0.754	0.754	0.754	0.694	0.694	0.694	0.694
Nombre total d'atomes par clause	132.12	132.12	133.12	133.12	63.62	63.62	64.62	64.62
Nombres de composantes	17.72	17.72	17.72	17.72	8.78	8.78	8.78	8.78
Nombre total d'hypothèses par classe	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0

TABLE 5.5 – Résultats d'apprentissage des codages MQD pour le corpus 2

L'évaluation du corpus 2 montre que les codages de la famille SOP permettent une F-mesure plus élevée ainsi qu'un temps d'exécution plus court que les codages des autres familles. Ainsi, des arbres dont certains labels sont variabilisés permettent de mieux classer qu'un ensemble de motifs d'arbres provenant du codage *first-child next-sibling*. Le temps d'exécution sur le corpus 2 est de l'ordre de quelques secondes à quelques minutes en fonction du codage utilisé, l'apprentissage a donc été rapidement effectué. Cependant, ce temps d'exécution augmente considérablement lorsque l'on utilise un codage MQD au détriment d'un codage SOP. L'explosion de la taille des moindres généralisés observés pour les codages MQD avec labels est visible sur le temps d'exécution de leurs apprentissages. Ces derniers sont environ 60 fois plus long que ceux réalisés avec des codages sans label. Enfin, les apprentissages basés sur un codage SOP+MQD sont plus performants que ceux n'utilisant qu'un codage MQD mais moins que ceux se limitant à un codage SOP. Ceci se justifie par

	pc +fcns +root +leaf	**count +fcns +root +leaf**	**trans +fcns +root +leaf**	**pc +trans +fcns +root +leaf**	**pc +count +fcns +root +leaf**	**trans +count +fcns +root +leaf**	**pc +trans +count +fcns +root +leaf**
Temps d'exécution (j :h :m :s :ms)	1 :23 :361	1 :3 :52	1 :21 :43	1 :0 :429	1 :20 :579	1 :12 :356	1 :1 :561
Rappel	0.635	0.635	0.635	0.635	0.635	0.635	0.635
Précision	0.98	0.98	0.98	0.98	0.98	0.98	0.98
F-mesure	0.754	0.754	0.754	0.754	0.754	0.754	0.754
Nombre total d'atomes par clause	145.62	139.48	158.4	170.9	151.98	164.76	177.26
Nombres de composantes	30.22	24.08	43.0	55.5	36.58	49.36	61.86
Nombre total d'hypothèses par classe	1.0	1.0	1.0	1.0	1.0	1.0	1.0
	pc +fcns -label +root +leaf	**count +fcns -label +root +leaf**	**trans +fcns -label +root +leaf**	**pc +trans +fcns -label +root +leaf**	**pc +count +fcns -label +root +leaf**	**trans +count +fcns -label +root +leaf**	**pc +trans +count +fcns -label +root +leaf**
Temps d'exécution (j :h :m :s :ms)	1 :689	1 :755	1 :738	1 :725	1 :611	1 :613	1 :748
Rappel	0.77	0.765	0.77	0.77	0.765	0.765	0.765
Précision	1.0	1.0	1.0	1.0	1.0	1.0	1.0
F-mesure	0.855	0.849	0.855	0.855	0.849	0.849	0.849
Nombre total d'atomes par clause	77.12	70.98	89.9	102.4	83.48	96.26	108.76
Nombres de composantes	21.28	15.14	34.06	46.56	27.64	40.42	52.92
Nombre total d'hypothèses par classe	1.0	1.0	1.0	1.0	1.0	1.0	1.0

TABLE 5.6 – Résultats d'apprentissage des codages SOP+MQD pour le corpus 2

la mesure de rappel qui nous indique qu'une hypothèse SOP est moins sélective qu'une hypothèse MQD ou SOP+MQD mais demeure tout aussi précise. Ce point peut s'expliquer par une description trop précise du concept à apprendre réalisée par les hypothèses ou par un ensemble d'exemples ne reflétant pas assez le concept à apprendre.

Nous terminons avec le corpus de documents Wikipedia dont les résultats sont présentés à la table 5.7. Ce corpus contient une quantité de documents trop importante. Ainsi, nous ne considérons que les représentations SOP ainsi que les codages MQD sans label pour un apprentissage.

	pc	count	trans	pc +trans	count +trans	pc +count	pc +count +trans
Temps d'exécution (j :h :m :s :ms)	35	44	63	63	93	89	132
Rappel	1.0	1.0	0.999	0.999	0.999	1.0	0.999
Précision	0.514	0.514	0.514	0.514	0.514	0.514	0.514
F-mesure	0.679	0.679	0.679	0.679	0.679	0.679	0.679
Nombre total d'atomes par clause	3.0	9.0	22.44	25.44	31.44	12.0	34.44
Nombres de composantes	3.0	9.0	22.44	25.44	31.44	12.0	34.44
Nombre total d'hypothèses par classe	1.0	1.0	1.0	1.0	1.0	1.0	1.0

TABLE 5.7 – Résultats d'apprentissage des codages SOP pour le corpus 3

Malgré une quantité de documents conséquente, l'apprentissage d'une hypothèse par moindre généralisé est effectué rapidement avec un temps d'exécution ne dépassant pas l'ordre de la seconde. La F-mesure reste raisonnable puisqu'elle avoisine les 70%. Contrairement aux corpus précédents, la précision n'est plus de 1. Ceci nous indique que de nombreux documents ne devant pas être reconnus ont été acceptés par les hypothèses apprises. Cette mesure est néanmoins contrebalancée par celle du rappel. Cette dernière nous indique qu'une grande majorité des documents qui auraient dû être reconnus par les hypothèses le sont.

	fcns -label	fcns -label +leaf	fcns -label +root	fcns -label +root +leaf
Temps d'exécution (j :h :m :s :ms)	6 :27 :70	6 :24 :963	6 :27 :733	6 :34 :720
Rappel	0.998	0.998	0.998	0.998
Précision	0.514	0.514	0.514	0.514
F-mesure	0.678	0.678	0.678	0.678
Nombre total d'atomes par clause	27.3	28.3	32.6	33.6
Nombres de composantes	4.6	4.6	4.6	4.6
Nombre total d'hypothèses par classe	1	1	1	1

TABLE 5.8 – Résultats d'apprentissage des codages MQD sans label pour le corpus 3

	pc +fcns -label +root +leaf	**count +fcns -label +root +leaf**	**trans +fcns -label +root +leaf**	**pc +trans +fcns -label +root +leaf**	**pc +count +fcns -label +root +leaf**	**trans +count +fcns -label +root +leaf**	**pc +trans +count +fcns -label +root +leaf**
Temps d'exécution (j :h :m :s :ms)	8 :34 :270	8 :52 :417	8 :44 :212	8 :33 :85	7 :54 :92	8 :51 :7	9 :1 :624
Rappel	0.991	0.985	0.991	0.991	0.985	0.985	0.985
Précision	0.541	0.531	0.541	0.541	0.531	0.531	0.531
F-mesure	0.698	0.689	0.698	0.698	0.689	0.689	0.689
Nombre total d'atomes par clause	77.21	70.89	89.1	102.2	83.38	96.63	108.76
Nombres de composantes	36.6	42.6	56.04	59.04	65.04	45.6	68.0
Nombre total d'hypothèses par classe	7.6	13.6	27.14	30.04	16.04	16.6	39.04
Nombre total d'hypothèses par classe	1	1	1	1	1	1	1

TABLE 5.9 – Résultats d'apprentissage des codages SOP+MQD sans label pour le corpus 3

Le même constat peut être effectué pour les tables 5.8 et 5.9 présentant respectivement les scores obtenus avec les codages MQD sans label et ceux couplant SOP et MQD sans label. Remarquons que ces scores sont égaux ou très légèrement supérieurs à ceux obtenus avec un codage SOP. Cependant, les codages SOP présentent un avantage certain au niveau du temps d'exécution puisque, à F-mesure égale, l'apprentissage avec un codage MQD sans label est 10000 fois plus lent.

L'apprentissage par moindre généralisé donne donc de bons résultats sur ces corpus. Néanmoins, nous pensons que ces derniers peuvent être améliorés. En effet, l'emploi du moindre généralisé seul implique qu'exactement une hypothèse est apprise. Le moindre généralisé produit à partir d'un ensemble d'exemples doit couvrir chaque exemple de cet ensemble. Il peut alors souffrir d'une surgénéralisation si les exemples présentent une grande diversité. L'hypothèse apprise est alors amenée à couvrir des exemples qui n'auraient pas dû l'être. C'est ce que nous laisse penser l'évaluation du corpus 3. Ainsi, nous allons nous intéresser à une approche disjonctive. Elle rend possible l'apprentissage de plusieurs hypothèses et tire profit d'exemples négatifs, chose impossible avec l'emploi du moindre généralisé seul.

5.6.3 Apprentissage disjonctif par moindre généralisation

Algorithme d'apprentissage disjonctif

Le calcul du moindre généralisé s'effectue en effet uniquement à partir d'exemples issus de la classe positive et ne tire pas profit des exemples négatifs. Ainsi, afin de réaliser un apprentissage disjonctif par moindre généralisé à partir de positifs et négatifs, nous allons instancier un algorithme de type DLG [Webb 1992]. Il s'agit d'un algorithme glouton permettant la production de règles. Celui-ci prend en entrée deux ensembles d'exemples, l'un de positifs et l'autre de négatifs.

L'algorithme DLG repose sur l'utilisation d'algorithmes produisant des moindres généralisés maximalement corrects [Torre 2004]. Un moindre généralisé d'exemples positifs est correct vis-à-vis d'exemples positifs et négatifs lorsqu'il ne couvre pas d'exemple d'une autre classe que la sienne. Un moindre généralisé correct est maximal vis-à-vis de ces ensembles si l'on ne peut plus le généraliser avec un exemple supplémentaire de l'ensemble correspondant à sa classe d'apprentissage sans perdre la propriété de correction. L'algorithme 14 détaille une méthode de calcul d'un moindre généralisé correct.

Algorithme 14 Moindre Généralisé Correct

function $\mathsf{MGC}(E^+, E^-)$
1: $E^+ := E^+ \setminus \{h\}$;
2: **while** $E^+ \neq \emptyset$ **do**
3: $E^+ := E^+ \setminus \{e^+\}$;
4: $h' := h$;
5: $h := \mathsf{lgg}(h, e^+)$;
6: **for each** $e^- \in E^-$ **do**
7: **if** $h \succeq_\theta e^-$ **then**
8: $h := h'$;
9: **continue** ;
10: **end if**
11: **end for**
12: **end while**
13: **return** h ;
end function

Le calcul d'un moindre généralisé maximalement correct commence à partir d'un exemple, appelé graine, issu d'une des deux classes. On tente ensuite de le généraliser avec les autres exemples de même classe fournis en entrée de l'algorithme. Ces exemples sont généralisés les uns après les autres. Le choix du premier exemple et des suivants peut se faire de différentes manières et correspond aux lignes 1 et 5 de l'algorithme 14. Une première manière consiste à considérer que l'ensemble des exemples à généraliser est ordonné. On cherche alors à généraliser ces exemples les uns après les autres en suivant l'ordre imposé. Si, après l'une de ces généralisations, le moindre généralisé obtenu n'est plus correct avec l'ensemble des exemples généralisés, alors on reprend la généralisation précédente, on oublie l'exemple courant et on continue le procédé avec les exemples suivants. Sinon, le processus de généralisation continue avec l'exemple suivant. Ce calcul est terminé lorsqu'il est impossible de généraliser un exemple supplémentaire avec le moindre généralisé courant sans que la correction du moindre généralisé ne soit perdue.

Dans le cas où tous les exemples ne peuvent être utilisés, on voit que l'ordre des exemples généralisés importe. Une autre approche consiste alors à choisir aléatoirement l'ordre des exemples à généraliser comme dans [Webb 1991]. Cette approche peut être complétée en effectuant ensuite plusieurs fois le calcul du moindre généralisé dans le but de sélectionner le plus prometteur par exemple.

Le calcul d'un moindre généralisé maximalement correct à présent posé, nous pouvons expliquer le fonctionnement de DLG présenté par l'algorithme 15. L'algorithme DLG se base sur le calcul d'un moindre généralisé et a pour but de produire un ensemble de règles couvrant les exemples positifs et rejetant les négatifs. Plus précisément, ce dernier utilise l'algorithme 14 qui permet de produire un moindre généralisé maximalement correct vis-à-vis des exemples positifs et négatifs.

Ce processus d'apprentissage consiste à calculer un moindre généralisé maximalement correct, supprimer les exemples positifs couverts par ce moindre généralisé et recommencer l'opération jusqu'à ce que l'ensemble des positifs soit vide. Lorsque l'apprentissage est fini, on a alors appris un ensemble de clauses correctes vis-à-vis des exemples positifs et négatifs.

Algorithme 15 DLG

function $\mathsf{DLG}(E^+, E^-)$
1: $H := \emptyset$;
2: **while** $E^+ \neq \emptyset$ **do**
3: $h := \mathsf{MGC}(E^+, E^-)$;
4: **for each** $e^+ \in E^+$; **do**
5: **if** $h \succeq_\theta e^+$ **then**
6: $E := E \setminus \{e^+\}$;
7: **end if**
8: **end for**
9: $H := H \cup \{h\}$;
10: **end while**
11: **return** H ;
end function

Cette approche permet l'apprentissage de plusieurs hypothèses. Cependant, il est important de remarquer qu'elle ne cherche pas à construire des hypothèses visant à couvrir le plus d'exemples possible. Il existe évidemment d'autres méthodes d'apprentissage basées sur l'utilisation de moindre généralisé orientées dans cette direction. Nous retrouvons notamment AdaBoostMG [Torre 2004], une instanciation de AdaBoost [Freund 1997, Freund 1999] qui est un algorithme de Boosting, ou encore GloboBoost [Torre 2005], une adaptation au boosting de Globo [Torre 1999]. Le lecteur intéressé par le boosting peut se pencher sur les articles précédemment cités ainsi que sur [Torre 2009]. Ce dernier présente l'utilisation d'un moindre généralisé comme apprenant faible afin de bénéficier de techniques de boosting pour la classification. Ces techniques sont souvent plus performantes que DLG. Elles permettent l'apprentissage de clauses plus simples. Ces clauses assurent également une meilleure prédiction. Néanmoins, ces différents gains se font au détriment de la complexité de ces algorithmes qui sont moins rapides que DLG. Ainsi, l'algorithme DLG semble être un bon candidat au vu de la complexité du problème que nous traitons.

Apprentissage

Donnons maintenant les résultats expérimentaux portant sur nos trois corpus. Ces résultats vont nous permettre de comparer les différents codages ainsi que leurs influences sur la tâche de classification de documents XML. Comme précédemment, nous les présentons dans différentes tables contenant les scores de rappel, précision et F-mesure ainsi que le temps d'exécution des apprentissages, le nombre d'hypothèses apprises et le nombre de composantes d'une hypothèse. Ces résultats sont des moyennes tous runs et classes condondus en fonction du nombre d'exemples.

	pc	count	trans	pc +trans	count +trans	pc +count	pc +count +trans
Temps d'exécution (j :h :m :s :ms)	18	21	18	34	36	36	44
Rappel	1.0	1.0	1.0	1.0	1.0	1.0	1.0
Précision	1.0	1.0	1.0	1.0	1.0	1.0	1.0
F-mesure	1.0	1.0	1.0	1.0	1.0	1.0	1.0
Nombre total d'atomes par clause	163.0	17.0	163.0	326.0	180.0	180.0	343.0
Nombres de composantes	163.0	17.0	163.0	326.0	180.0	180.0	343.0
Nombre total d'hypothèses par classe	1.0	1.0	1.0	1.0	1.0	1.0	1.0

TABLE 5.10 – Résultats d'apprentissage des codages SOP pour le corpus 1

Le corpus footballistique ne présente aucune difficulté pour l'apprentissage disjonctif d'hypothèses par moindre généralisation. En effet, quelque soit le codage utilisé, la F-mesure, le rappel et la précision sont à 1 comme constaté dans les tables 5.10, 5.11 et 5.12. Ainsi, les hypothèses apprises reconnaissent exactement chaque classe d'une manière complète et correcte et sont les mêmes que celles apprises par moindre généralisation seule. Ce constat n'est pas surprenant puisque, lors de l'apprentissage par moindre généralisation, une seule hypothèse a été apprise en dépit du codage. Elle assure une F-mesure de 1 . Ainsi, il est normal qu'une seule hypothèse soit également apprise lors d'un apprentissage disjonctif par moindre généralisé.

	fcns	fcns +leaf	fcns +root	fcns +root +leaf	fcns -label	fcns -label +leaf	fcns -label +root	fcns -label +root +leaf
Temps d'exécution (j :h :m :s :ms)	3 :12 :103	2 :48 :590	3 :8 :722	3 :10 :836	48 :804	48 :621	49 :40	48 :836
Rappel	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0
Précision	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0
F-mesure	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0
Nombre total d'atomes par clause	464.0	464.0	465.0	465.0	163.0	163.0	164.0	164.0
Nombres de composantes	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0
Nombre total d'hypothèses par classe	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0

TABLE 5.11 – Résultats d'apprentissage des codages MQD pour le corpus 1

	pc +fcns +root +leaf	**count +fcns +root +leaf**	**trans +fcns +root +leaf**	**pc +trans +fcns +root +leaf**	**pc +count +fcns +root +leaf**	**trans +count +fcns +root +leaf**	**pc +trans +count +fcns +root +leaf**
Temps d'exécution (j :h :m :s :ms)	3 :13 :453	3 :10 :863	3 :11 :588	3 :9 :56	3 :12 :748	3 :9 :446	2 :44 :927
Rappel	1.0	1.0	1.0	1.0	1.0	1.0	1.0
Précision	1.0	1.0	1.0	1.0	1.0	1.0	1.0
F-mesure	1.0	1.0	1.0	1.0	1.0	1.0	1.0
Nombre total d'atomes par clause	628.0	482.0	628.0	791.0	645.0	645.0	808.0
Nombres de composantes	164.0	18.0	164.0	327.0	181.0	181.0	344.0
Nombre total d'hypothèses par classe	1.0	1.0	1.0	1.0	1.0	1.0	1.0
	pc +fcns -label +root +leaf	**count +fcns -label +root +leaf**	**trans +fcns -label +root +leaf**	**pc +trans +fcns -label +root +leaf**	**pc +count +fcns -label +root +leaf**	**trans +count +fcns -label +root +leaf**	**pc +trans +count +fcns -label +root +leaf**
Temps d'exécution (j :h :m :s :ms)	48 :413	47 :897	48 :596	47 :691	47 :861	47 :958	48 :144
Rappel	1.0	1.0	1.0	1.0	1.0	1.0	1.0
Précision	1.0	1.0	1.0	1.0	1.0	1.0	1.0
F-mesure	1.0	1.0	1.0	1.0	1.0	1.0	1.0
Nombre total d'atomes par clause	327.0	181.0	327.0	490.0	344.0	344.0	507.0
Nombres de composantes	164.0	18.0	164.0	327.0	181.0	181.0	344.0
Nombre total d'hypothèses par classe	1.0	1.0	1.0	1.0	1.0	1.0	1.0

TABLE 5.12 – Résultats d'apprentissage des codages SOP+MQD pour le corpus 1

Présentons maintenant les résultats expérimentaux des tables 5.13, 5.14 et 5.15 obtenus pour le corpus 2 des pages personnelles de chercheurs. Concernant les codages SOP, nous n'observons pas d'amélioration des scores de l'apprentissage disjonctif par rapport à l'apprentissage par moindre généralisation seule. En effet, les scores de F-mesure, rappel et précision sont identiques pour les deux types d'apprentissage. Une seule hypothèse a été apprise pour chaque apprentissage disjonctif. Ainsi, ce type d'apprentissage s'assimile à l'apprentissage d'un unique moindre généralisé. Cette observation explique que les scores soient les mêmes pour ces différents types d'apprentissage.

L'observation faite pour les codages SOP peut-être reprise pour la table 5.15 et une partie des codages de la table 5.14. En effet, les apprentissages basés sur les codages MQD usant de labels, c'est-à-dire **fcns**, **fcns**+**leaf**, **fcns**+**root**, et **fcns**+**root**+**leaf**, et ceux

couplant SOP et MQD, n'ont appris qu'une seule hypothèse chacun et ont les mêmes scores qu'un apprentissage par moindre généralisé seul. Ainsi, l'apprentissage disjonctif avec l'un de ces codages est similaire à celui d'un unique moindre généralisé avec ce même codage.

	pc	**count**	**trans**	**pc +trans**	**count +trans**	**pc +count**	**pc +count +trans**
Temps d'exécution (j :h :m :s :ms)	4	7	6	7	12	9	10
Rappel	0.91	0.87	0.92	0.91	0.87	0.86	0.86
Précision	1.0	1.0	1.0	1.0	1.0	1.0	1.0
F-mesure	0.947	0.923	0.954	0.947	0.923	0.915	0.915
Nombre total d'atomes par clause	12.5	6.36	25.28	37.78	31.64	18.86	44.14
Nombres de composantes	12.5	6.36	25.28	37.78	31.64	18.86	44.14
Nombre total d'hypothèses par classe	1.0	1.0	1.0	1.0	1.0	1.0	1.0

TABLE 5.13 – Résultats d'apprentissage des codages SOP pour le corpus 2

Cette remarque n'est cependant pas vraie pour les codages MQD sans label. En effet, ces apprentissages apprennent environ deux hypothèses. Leurs scores de F-mesure, rappel et précision sont meilleurs que ceux de leurs homologues des apprentissages non disjonctifs. La F-mesure pour ces codages passe de 0.694 à 0.819 et surpasse ainsi celles des apprentissages à base de codages avec labels.

	fcns	**fcns +leaf**	**fcns +root**	**fcns +root +leaf**	**fcns -label**	**fcns -label +leaf**	**fcns -label +root**	**fcns -label +root +leaf**
Temps d'exécution (j :h :m :s :ms)	1 :19 :243	1 :21 :35	1 :19 :257	1 :19 :58	2 :466	2 :510	2 :463	2 :475
Rappel	0.635	0.635	0.635	0.635	0.725	0.725	0.725	0.725
Précision	0.98	0.98	0.98	0.98	0.98	0.98	0.98	0.98
F-mesure	0.754	0.754	0.754	0.754	0.819	0.819	0.819	0.819
Nombre total d'atomes par clause	132.12	132.12	133.12	133.12	57.7	57.7	58.7	58.7
Nombres de composantes	17.72	17.72	17.72	17.72	6.08	6.08	6.08	6.08
Nombre total d'hypothèses par classe	1.0	1.0	1.0	1.0	1.86	1.86	1.86	1.86

TABLE 5.14 – Résultats d'apprentissage des codages MQD pour le corpus 2

	pc +fcns +root +leaf	**count +fcns +root +leaf**	**trans +fcns +root +leaf**	**pc +trans +fcns +root +leaf**	**pc +count +fcns +root +leaf**	**trans +count +fcns +root +leaf**	**pc +trans +count +fcns +root +leaf**
Temps d'exécution (j :h :m :s :ms)	59 :687	1 :19 :803	1 :2 :618	59 :740	1 :11 :213	1 :2 :496	1 :19 :148
Rappel	0.635	0.635	0.635	0.635	0.635	0.635	0.635
Précision	0.98	0.98	0.98	0.98	0.98	0.98	0.98
F-mesure	0.754	0.754	0.754	0.754	0.754	0.754	0.754
Nombre total d'atomes par clause	145.62	139.48	158.4	170.9	151.98	164.76	177.26
Nombres de composantes	30.22	24.08	43.0	55.5	36.58	49.36	61.86
Nombre total d'hypothèses par classe	1.0	1.0	1.0	1.0	1.0	1.0	1.0
	pc +fcns -label +root +leaf	**count +fcns -label +root +leaf**	**trans +fcns -label +root +leaf**	**pc +trans +fcns -label +root +leaf**	**pc +count +fcns -label +root +leaf**	**trans +count +fcns -label +root +leaf**	**pc +trans +count +fcns -label +root +leaf**
Temps d'exécution (j :h :m :s :ms)	2 :282	2 :249	2 :258	2 :267	2 :261	2 :225	2 :102
Rappel	0.77	0.765	0.77	0.77	0.765	0.765	0.765
Précision	1.0	1.0	1.0	1.0	1.0	1.0	1.0
F-mesure	0.855	0.849	0.855	0.855	0.849	0.849	0.849
Nombre total d'atomes par clause	77.12	70.98	89.9	102.4	83.48	96.26	108.76
Nombres de composantes	21.28	15.14	34.06	46.56	27.64	40.42	52.92
Nombre total d'hypothèses par classe	1.0	1.0	1.0	1.0	1.0	1.0	1.0

TABLE 5.15 – Résultats d'apprentissage des codages SOP+MQD pour le corpus 2

Terminons avec le corpus 3 du challenge INEX. Les résultats obtenus à partir d'apprentissages disjonctifs et de codages SOP sont similaires à ceux des apprentissages par moindre généralisation. Le nombre d'atomes des clauses apprises ainsi que le temps d'exécution sont cependant plus élevés. Les hypothèses apprises sont finalement plus concises mais prennent également plus de temps à être construites.

	pc	count	trans	pc +trans	count +trans	pc +count	pc +count +trans
Temps d'exécution (j :h :m :s :ms)	4 :259	22 :65	1 :44 :956	1 :49 :294	1 :50 :260	27 :355	1 :56 :396
Rappel	0.587	0.594	0.482	0.485	0.559	0.619	0.555
Précision	0.817	0.793	0.779	0.776	0.788	0.796	0.787
F-mesure	0.683	0.679	0.595	0.597	0.654	0.696	0.65
Nombre total d'atomes par clause	53.27	21.007	138.805	144.777	129.101	30.037	136.075
Nombres de composantes	53.27	21.007	138.805	144.777	129.101	30.037	136.075
Nombre total d'hypothèses par classe	210.62	170.5	198.04	190.9	143.1	153.42	141.52

TABLE 5.16 – Résultats d'apprentissage des codages SOP pour le corpus 3

Comme nous l'indiquent les tables 5.17 et 5.18, le même constat peut être appliqué aux codages MQD sans label ainsi qu'aux codages couplant SOP et MQD sans label. Remarquons qu'à F-mesure équivalente les codages SOP sont 10 à 100 fois plus rapide.

	fcns -label	fcns -label +leaf	fcns -label +root	fcns -label +root +leaf
Temps d'exécution (j :h :m :s :ms)	43 :7 :70	41 :40 :972	43 :7 :733	43 :54 :720
Rappel	0.998	0.998	0.998	0.998
Précision	0.514	0.514	0.514	0.514
F-mesure	0.678	0.678	0.678	0.678
Nombre total d'atomes par clause	47.3	47.3	48.6	48.6
Nombres de composantes	14.6	14.6	14.6	14.6
Nombre total d'hypothèses par classe	153.04	153.04	153.04	153.04

TABLE 5.17 – Résultats d'apprentissage des codages MQD sans label pour le corpus 3

	pc +fcns -label +root +leaf	**count +fcns -label +root +leaf**	**trans +fcns -label +root +leaf**	**pc +trans +fcns -label +root +leaf**	**pc +count +fcns -label +root +leaf**	**trans +count +fcns -label +root +leaf**	**pc +trans +count +fcns -label +root +leaf**
Temps d'exécution (j :h :m :s :ms)	43 :58 :981	44 :16 :788	:45 :40 :676	45 :44 :14	45 :44 :980	44 :22 :83	45 :51 :123
Rappel	0.991	0.985	0.991	0.991	0.985	0.985	0.985
Précision	0.541	0.531	0.541	0.541	0.531	0.531	0.531
F-mesure	0.698	0.689	0.698	0.698	0.689	0.689	0.689
Nombre total d'atomes par clause	77.21	70.89	89.1	102.2	83.38	96.63	108.76
Nombres de composantes	21.28	15.14	34.06	46.56	27.64	40.42	52.92
Nombre total d'hypothèses par classe	210.62	170.5	198.04	183.9	153.10	153.42	149.52

TABLE 5.18 – Résultats d'apprentissage des codages SOP+MQD sans label pour le corpus 3

5.6.4 Conclusion

Grâce à ces expériences, nous avons montré que nos méthodes permettent effectivement l'apprentissage de classifieurs d'arbres. Nous apprenons une théorie en, au plus, quelques heures là où l'on estime que les algorithmes de [Plotkin 1970] mettront plusieurs années à répondre à un test de subsomption. Cette tâche impossible avec les algorithmes généraux de [Plotkin 1970] l'est donc avec nos algorithmes.

En général, la représentation SOP permet, un léger gain au niveau de la F-mesure vis-à-vis des clauses MQD. Ceci s'explique par la capacité qu'a le langage SOP à variabiliser des labels d'un arbre, chose impossible pour les langages basés sur le codage **first-child next-sibling**. Ainsi, l'apprentissage de motifs d'arbres permis par des langages basés sur sur le codage **first-child next-sibling** est moins efficace que cette variabilisation. Nos représentations SOP et MQD sur ces corpus sont néanmoins très proches en terme de qualité des prédictions faites (erreur de l'ordre de 10 %). Les codages SOP présentent cependant un avantage au niveau du temps d'exécution qui peut être jusqu'à 1000 fois inférieur. Le couplage de codage SOP et MQD ne permet pas, vis-à-vis de notre tâche, un gain au niveau de nos mesures. En effet, le partie MQD d'un couplage réalise généralement un moins bon score ce qui tend à pénaliser l'ensemble du codage. Enfin, contrairement à ce qui était attendu théoriquement, nous n'avons pas ou peu constaté d'explosion de la taille du moindre généralisé dans le nombre d'exemples pour les codages MQD. Nos codages, au travers de tâches réelles, semblent donc intéressants puisqu'ils ne souffrent pas d'une complexité élevée et sont appropriés pour une tâche de classification de documents XML.

5.7 Travaux apparentés

Dans le chapitre 4, nous avons présenté plusieurs approches en PLI ainsi que dans d'autres domaines proches de nos travaux. Nous effectuons dans cette section une com-

paraison de notre approche avec ces derniers et montrons les avantages respectifs de ces méthodes ainsi que de nos travaux vis-à-vis de notre problème de classification. Nous regroupons ces différentes approches en deux catégories. La première correspond à des restrictions syntaxiques imposées sur la forme des clauses, la seconde à des restrictions sur la subsomption ainsi qu'à des approches alternatives issues d'autres domaines.

5.7.1 Restrictions syntaxiques

Les clauses déterminées

Commençons tout d'abord avec les travaux de [Thomas 2005] portant sur la tâche d'extraction d'information. Pour cette tâche, Bernd propose une représentation des arbres très proche de nos objectifs avec, par exemple, l'utilisation des axes XPath pour lier les nœuds. Comme nous, c'est la θ-subsomption et le calcul de moindre généralisé qui sont utilisés pour l'apprentissage mais pour rendre le calcul traitable, une heuristique propre à la tâche d'extraction est appliquée. Celle-ci reformule les clauses afin d'obtenir des descriptions où chaque symbole de prédicat n'est utilisé qu'une fois. Autrement dit, ce ne sont ni plus, ni moins que des clauses SOP qui sont manipulées par l'algorithme d'apprentissage. Contrairement à Bernd, nous avons proposé un framework complet tirant parti au maximum des propriétés de cette sous-classe de clauses déterminées. Ainsi, l'utilisation de nos algorithmes est préférable.

Les clauses ij-déterminées

Nous continuons avec les travaux de [Muggleton 1990]. Muggleton et Feng ont conçu la méthode Golem qui manipule des clauses ij-déterminées. Comme leur nom l'indique, il s'agit avant tout de clauses déterminées, les paramètres i et j contrôlant respectivement la longueur maximale d'une chaîne de variables et le nombre maximal de variables à instancier pour en définir une autre. Une présentation plus complète de ces clauses ainsi que les différents résultats obtenus se trouvent dans le chapitre 4 sous-section 4.3.3.

En tout état de cause, les clauses ij-déterminées, avec i et j fixés, forment une sous-classe des clauses déterminées. Elles n'atteignent donc pas l'expressivité de nos clauses MQD. De plus, il n'est pas immédiat de trouver un codage d'arbres utilisant des clauses ij-déterminées, avec les bonnes valeurs de i et de j. Rappelons qu'il est conseillé par Muggleton et Feng de conserver des valeurs pour i et j très petites. Cette limitation s'explique par la complexité en temps du calcul de moindre généralisé proposé par Muggleton et Feng qui dépend d'un polynôme de degré i^j. Or, le paramètre j d'une clause déconnectée correspond à l'infini. Sachant que les clauses MQD peuvent contenir plusieurs composantes déconnectées les unes des autres, l'emploi des clauses ij-déterminées n'est donc pas adapté pour les langages proposés. Le calcul du moindre généralisé présent dans Golem n'est donc pas intéressant pour notre tâche, ni pour nos codages. Il est préférable de favoriser notre approche.

Les clauses k-locales

Kietz et Lübbe ont proposé un algorithme de θ-subsomption efficace pour les *clauses k-locales* [Kietz 1994a]. Ces clauses peuvent avoir une partie déterminée mais aussi et surtout des parties non déterminées appelées *locales*. Une définition plus formelle se trouve dans le chapitre 4 sous-section 4.3.2. Ces parties locales sont similaires à nos clauses QD au sein des MQD. Chaque locale est indépendante du point de vue des variables vis-à-vis des autres locales.

Cependant, là où nous exigeons un quasi-déterminisme, les *locales* sont contraintes de ne pas avoir plus de k littéraux ou k variables propres. Les locales peuvent alors être non déterminées et permettent la représentation d'une partie des clauses appartenant à $\mathcal{L}_{child}^{T_\Sigma}$ lorsque k est fixé ou bien la totalité si k est infini. Cette liberté sur la forme des clauses autorisées a un coût sur la complexité de la θ-subsomption. Ainsi, gérer des parties non déterminées présentes dans les locales ramène un test de θ-subsomption à une complexité exponentielle. Nous retrouvons alors la complexité de la θ-subsomption originale de [Plotkin 1970]. Les auteurs conviennent qu'il faut choisir des valeurs de k très petites. Ainsi, nos méthodes sont plus efficaces et nos représentations plus appropriées au codage des arbres.

Les clauses ordonnées

D'autres approches ont visé à étudier l'ordre des littéraux dans une clause. Ainsi les notions de clauses séquentielles [van der Laag 1995], clauses SeqLog [Lee 2004] et clauses ordonnées [Kuwabara 2005] consistent à imposer un ordre sur les littéraux d'une clause afin de la transformer en une séquence de littéraux. Cette notion d'ordre est omniprésente en PLI. On la retrouve dans la résolution SLD ou encore dans les classes de clauses déterminées. L'ordre des littéraux d'une clause a une incidence sur le calcul de la θ-subsomption. Ainsi, la subsomption entre clauses ordonnées revient à trouver une fonction monotone croissante de l'ensemble ordonné des littéraux de la première clause vers l'ensemble ordonné des littéraux de la seconde clause. Cette θ-subsomption entre clauses ordonnées est un problème NP-complet [Kuwabara 2005]. Sa complexité s'explique par la présence de littéraux non déterminés que l'ordre entre les littéraux d'une clause n'exclut pas contrairement à nos clauses MQD, QD et SOP

La θ-subsomption pour clauses MQD peut être ramenée à une fonction totale des composantes d'une clause vers celle d'une autre. Cependant, contrairement aux clauses ordonnées, une clause MQD est définie par un ensemble de composantes QD pour lesquelles il n'est pas nécessaire de fixer un ordre. Ainsi, là où les clauses ordonnées exigent un ordre entre les littéraux, une clause MQD se doit d'être partiellement ordonnée puisque seuls les littéraux au sein d'une même composante le sont. Aucun ordre n'existe entre les composantes d'une même clause. Nous donnons un exemple de la difficulté à trouver un ordre pour les composantes d'une clause MQD.

Exemple 51. *Les clauses de $\mathcal{L}_{fcns}^{\Sigma}$ sont des multi-clauses, les littéraux de leurs composantes sont donc déjà ordonnés. Afin de comparer ces clauses à celles ordonnées, nous devons définir un ordre entre leurs composantes. Nous séparons les composantes en deux catégories. La première regroupe celles dont le premier littéral a pour prédicat fc et la seconde celles dont le premier littéral a pour prédicat ns. Une preuve en annexe (cf. section A.1) démontre que le langage T_Σ est dénombrable. On en déduit que le langage $T_{\Sigma\cup\{?\}}$ d'arbres de T_Σ partiellement étiqueté l'est aussi. Le symbole ? représente les nœuds non étiquetés. Les composantes du second groupe sont assimilables à des arbres où seul le premier littéral fc est manquant. Nous ordonnons ainsi ces composantes avec l'ordre établi pour le langage $T_{\Sigma\cup\{?\}}$. L'union de deux ensembles dénombrables est dénombrable, il existe donc un ordre pour les composantes d'une clause de $\mathcal{L}_{fcns}^{\Sigma}$. Cet ordre consiste à classer les composantes représentant des arbres par ordre croissant selon leur profondeur, leur largeur et le positionnement des nœuds vers la droite (plus de détails en annexes).*

Soit les clauses $\mathscr{C} = \{C_1, C_2\}$, $\mathscr{D} = \{D_1, D_2\}$ et $\mathscr{E} = \{E_1, E_2\}$ telles que :

$$
\begin{array}{lcl}
C_1 & = & fc(X_1, X_2),\ b(X_2). \\
C_2 & = & fc(X_1, X_2),\ fc(X_2, X_3),\ a(X_1),\ c(X_3).
\end{array}
$$

$$
\begin{array}{lcl}
D_1 & = & fc(X_1, X_2),\ a(X_1),\ b(X_3). \\
D_2 & = & fc(X_1, X_2),\ fc(X_2, X_3),\ fc(X_3, X_4),\ a(),\ b(X_3).
\end{array}
$$

$$
\begin{array}{lcl}
E_1 & = & fc(X_1, X_2),\ fc(X_2, X_3),\ a(X_1),\ c(X_3). \\
E_2 & = & fc(X_1, X_2),\ fc(X_2, X_3),\ fc(X_3, X_4),\ b(X_3).
\end{array}
$$

Nous notons $\succeq_{ord}$ la θ-subsomption pour clauses ordonnées. On a $\mathscr{C} \succeq_\theta \mathscr{D}$ et $\mathscr{C} \succeq_\theta \mathscr{E}$. Dans le cas des clauses ordonnées, l'ordre proposé précédemment permet la construction des clauses $C = C_1, C_2$, $D = D_1, D_2$ et $E = E_1, E_2$ pour lesquelles on a alors $C \succeq_{ord} D$ et $C \not\succeq_{ord} E$. Ainsi, cet ordre ne convient pas pour la θ-subsomption des clauses ordonnées. Il est alors possible de considérer d'autres ordres. On peut, par exemple, construire la clause ordonnée $E' = E_2, E_1$ pour laquelle nous avons $C \succeq_{ord} E'$. L'ordre des composantes dépend ainsi des clauses utilisées lors du test de θ-subsomption. La θ-subsomption pour clauses MQD peut se ramener à un mapping des composantes de la première clause vers celles de la seconde. Un test de θ-subsomption consiste alors à trouver un ordre pour les composantes des clauses utilisées. Il ne semble cependant pas facile de trouver cet ordre.

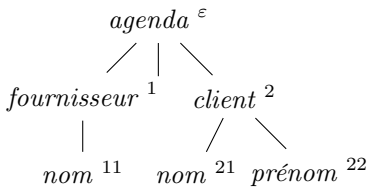
Kuwabara a également investigué la question de l'existence du moindre généralisé pour tout ensemble de clauses ordonnées et y a apporté une réponse négative. Il a cependant montré l'existence d'un nombre possiblement exponentiel de généralisations minimales. Cette absence de θ-subsomption polynomiale et de moindre généralisé nous incite à favoriser nos codages ainsi que nos algorithmes. De ce fait, l'utilisation lors d'un apprentissage de ce type de clauses ne semble pas approprié. Nos algorithmes sont plus efficaces dans le cadre des tâches visées.

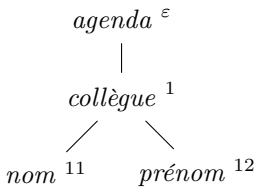
5.7.2 La subsomption

La θ-subsomption sous Identité d'Objet

Citons également les travaux de [Ferilli 2002] sur l'*identité d'objets* (OI). Une présentation détaillée de cette approche peut-être trouvée dans le chapitre 4 sous-section 4.1.3. Celle-ci stipule qu'il y a subsomption entre deux clauses uniquement si les variables de la première peuvent être mises en correspondance avec des termes *différents* de la seconde. Cette contrainte a deux effets. Premièrement, la θ-subsomption est de complexité moindre que dans le cas général des clauses de Horn mais reste au pire exponentielle. Deuxièmement, la θ-subsomption sous OI induit des moindres généralisés multiples ce qui complique l'algorithme d'apprentissage qui doit alors gérer plusieurs hypothèses concurrentes au même instant. Enfin, s'il y a des domaines où la contrainte OI est naturelle, il ne nous semble pas qu'elle soit pertinente pour la classification de documents XML et la généralisation d'arbres qui nous intéressent. Ainsi, dans l'exemple suivant, on peut voir que la présence de la contrainte OI ne permet pas au moindre généralisé de conserver l'intégralité des informations communes à chaque clause.

Exemple 52. *Soit deux clauses de $\mathcal{L}_{fcns}^{T_\Sigma}$ et leurs arbres associés :*

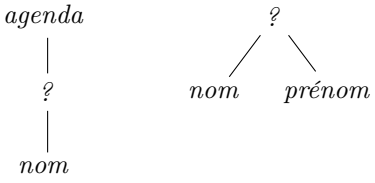




$$C = fc(X_\varepsilon, X_1),\ ns(X_1, X_2)\ fournisseur(X_1),\ client(X_2),\ agenda(X_\varepsilon),\ fc(X_1, X_{11}),\ fc(X_2, X_{21}),\ ns(X_{21}, X_{22}),\ nom(X_{11}),\ nom(X_{21}),\ prénom(X_{22}).$$

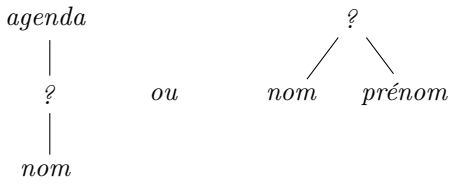
$$D = fc(X_\varepsilon, X_1),\ agenda(X_\varepsilon),\ collège(X_1),\ fc(X_1, X_{11}),\ ns(X_{11}, X_{12}),\ nom(X_{11}),\ prénom(X_{12}).$$

Elles ont pour moindre généralisé la clause suivante :



$$\mathsf{lgg}(C, D) = fc(Y_1, Y_2), fc(Y_2, Y_3),\ agenda(Y_1),\ nom(Y_3),\ fc(Y_4, Y_5), ns(Y_5, Y_6),\ nom(Y_5),\ prénom(Y_6).$$

et sous OI, l'une des clauses suivantes :



$$\mathsf{lgg}(C, D) = fc(Y_1, Y_2), fc(Y_2, Y_3),\ agenda(Y_1),\ nom(Y_3).$$

ou

$$\mathsf{lgg}(C, D)\quad fc(Y_1, Y_2), ns(Y_2, Y_3),\ nom(Y_2),\ prénom(Y_3).$$

Le moindre généralisé de Plotkin conserve deux informations. La première specifie qu'un agenda contient un élément qui possède un nom. La seconde spécifie qu'un élément dispose d'un nom et d'un prénom. Dans le cas de la θ-subsomption sous OI, un choix doit être fait entre ces deux informations, le filtrage est donc moins précis. Désirant reconnaître les éléments d'une classe le plus précisément possible, cette approche est donc moins adaptée.

L'approche CSP

Les travaux de [Maloberti 2001] et l'algorithme Django proposent de réduire un calcul de θ-subsomption en un système de contraintes à satisfaire. Les méthodes développées dans le cadre des CSP, enrichies de différentes heuristiques sont alors mises en œuvre pour répondre à la question de subsomption. Nous ne savons pas si nos représentations induiraient des systèmes de contraintes avec des particularités que la méthode saurait détecter et exploiter. On imagine que si c'est le cas, la complexité obtenue serait comparable à la nôtre et que, sinon, Django peut revenir à un calcul de complexité exponentielle. Par conséquent, nous jugeons préférable d'utiliser nos algorithmes lorsque les clauses sont MQD, et de réserver Django aux cas où les clauses manipulées n'ont pas de propriété exploitable.

D'autres approches [Horváth 2010, Kuželka 2013], inspirées de ce dernier, ont proposé des relations de subsomption et de moindre généralisation se ramenant à des problèmes CSP. Nous nous intéressons plus particulièrement à [Kuželka 2013] qui a proposé une opération

de subsomption, appelée x-subsomption, ainsi qu'une opération de généralisation, nommée *bounded least general generalization*, polynomiales. Les deux sont respectivement des approximations de la θ-subsomption ainsi que du moindre généralisé de Plotkin et peuvent induire la construction d'une généralisation exponentielle dans le nombre d'exemples. Nos algorithmes sont équivalents à ceux de Plotkin pour nos langages de clauses. Ceux-ci peuvent être également calculés en temps polynomial. Ces approches sont donc moins adaptées à notre problème.

La subsomption par les graphes

Terminons avec [Liquiere 2007, Douar 2011, Douar 2012] qui ont proposé une nouvelle relation de généralité ainsi qu'un calcul de moindre généralisé adapté à celle-ci pour les graphes. Une présentation formelle de cette relation ainsi que de l'algorithme de généralisation est faite dans le chapitre 4 sous-section 4.2.4. Nos algorithmes, comme ceux présentés dans [Liquiere 2007], sont polynomiaux. La méthode de généralisation introduite par Liquière est basée sur un produit de graphes, elle a donc une complexité quadratique comme notre calcul de moindre généralisation. L'algorithme de subsomption de Liquière a, quant-à-elle, une complexité cubique et est plus précisément de $\mathcal{O}(ed^2)$ en temps où e est le nombre d'arcs du graphe et d la taille du plus grand sous-ensemble de nœuds avec le même label. Elle est ainsi moins intéressante que notre θ-subsomption pour les clauses MQD mais peut apporter un éventuel gain en complexité pour des graphes où la variable d est faible.

Un autre inconvénient de cette subsomption, plus important à nos yeux pour notre tâche, est la non-équivalence du problème d'homomorphisme de graphe avec celui de l'AC-Projection. Elle a pour conséquence que les moindres généralisations sous AC-Projection obtenues sont moins spécifiques que celles existantes sous θ-subsomption.

Exemple 53. *Nous reprenons les arbres de l'exemple 52 et obtenons la généralisation sous AC-Projection suivante :*

Cette généralisation est moins spécifique que celle obtenue avec le calcul du moindre généralisé de Plotkin. En effet, l'information selon laquelle un nœud agenda a comme petit-fils un nœud nom est perdue. Cette perte est causée en partie par le produit de graphe ne permettant la généralisation que pour des arêtes disposant de nœuds ayant les mêmes labels.

Nos algorithmes ont une complexité plus intéressante que ceux proposés pour l'AC-Projection. De plus, la tâche de classification consiste à reconnaitre au mieux un ensemble d'exemples, point que permet notre moindre généralisé. Nous pensons donc que ces derniers sont plus appropriés pour notre tâche de classification.

5.8 Conclusion

Nous avons proposé plusieurs sous-familles des clauses de Horn destinées à la classification, entre autres, d'arbres et de documents XML. Pour chacune, nous avons proposé une réécriture des algorithmes de θ-subsomption et de moindre généralisé avec une complexité polynomiale. Ces algorithmes permettent de constituer un framework polynomial d'apprentissage basé sur l'utilisation du moindre généralisé et de la θ-subsomption. La table 5.19 récapitule toutes les complexités établies dans ce chapitre.

	θ-subsomption	moindre-généralisé	moindre-généralisé réduit
Horn	$\|D\|^{\mathcal{O}(\|C\|)}$	$\mathcal{O}(\|C\| \times \|D\|)$	$(\|C\| \times \|D\|)^{\mathcal{O}(\|C\| \times \|D\|)}$
$\mathcal{L}_{pc}^{T_\Sigma}$	$\mathcal{O}(\|C\|)$	$\mathcal{O}(min(\|C\|, \|D\|))$	$\mathcal{O}(min(\|C\|, \|D\|))$
SOP	$\mathcal{O}(\|C\|)$	$\mathcal{O}(min(\|C\|, \|D\|))$	$\mathcal{O}(min(\|C\|, \|D\|))$
D	$\mathcal{O}(\|C\| \times \|D\|)$	$\mathcal{O}(\|C\| \times \|D\|)$	$\mathcal{O}((\|C\| \times \|D\|)^3)$
QD	$\mathcal{O}(\|C\| \times \|D\|)$	$\mathcal{O}(\|C\| \times \|D\|)$	-
$\mathcal{L}_{fcns}^{T_\Sigma}$	$\mathcal{O}(\|\mathscr{C}\| \times \|\mathscr{D}\|)$	$\mathcal{O}((\|\mathscr{C}\| \times \|\mathscr{D}\|)^2)$	$\mathcal{O}((\|\mathscr{C}\| \times \|\mathscr{D}\|)^4)$
FDP	$\mathcal{O}(\|\mathscr{C}\| \times \|\mathscr{D}\|)$	$\mathcal{O}((\|\mathscr{C}\| \times \|\mathscr{D}\|)^2)$	$\mathcal{O}((\|\mathscr{C}\| \times \|\mathscr{D}\|)^4)$
MQD	$\mathcal{O}(\|\mathscr{C}\| \times \|\mathscr{D}\|)$	-	-
SOP + FDP	$\mathcal{O}\left(\begin{array}{c} min(\|C_{sop}\|, \|D_{sop}\|) \\ + \quad \|C_{\mathsf{FDP}}\| \times \|D_{\mathsf{FDP}}\| \end{array}\right)$	$\mathcal{O}\left(\begin{array}{c} \|C_{sop}\| \times \|D_{sop}\| \\ + \quad (\|C_{\mathsf{FDP}}\| \times \|D_{\mathsf{FDP}}\|)^2 \end{array}\right)$	$\mathcal{O}\left(\begin{array}{c} min(\|C_{sop}\|, \|D_{sop}\|) \\ + \quad (\|C_{\mathsf{FDP}}\| \times \|D_{\mathsf{FDP}}\|)^4 \end{array}\right)$

TABLE 5.19 – Récapitulatif des complexités des opérations en fonction des clauses.

Notre démarche a consisté à suivre la piste des clauses *déterminées*. Ainsi, nous avons défini plusieurs familles de classes : les SOP, les QD, les FDP et les MQD. La famille des SOP inclut le langage $\mathcal{L}_{pc}^{T_{\Sigma \cup V}}$, une restriction des clauses déterminées pour les arbres. Les QD contiennent les déterminées mais est une famille strictement plus large. Enfin, la famille des MQD regroupe en son sein les QD ainsi que les clauses FDP. Nous avons également vu que les clauses du langage $\mathcal{L}_{fcns}^{\Sigma}$ forment une sous-classe dédiée aux arbres des FDP et donc des MQD.

La figure 5.12 montre les différentes classes de clauses concernées ainsi que leurs relations d'inclusion.

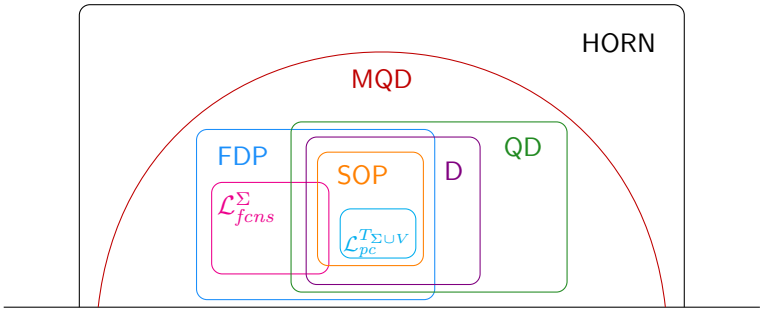


FIGURE 5.12 – Récapitulatif des classes de clauses.

Intéressés par une tâche de classification de documents XML, nous nous sommes arrêtés sur certaines classes permettant la représentation de documents XML ainsi que leur apprentissage. Nous avons montré que ces classes peuvent être utilisées de différentes manières et permettent la représentation de classes de graphes utilisables pour la représentation d'objets de la vie courante. L'espace des clauses MQD est vaste, nous sommes persuadés qu'il existe de nombreuses autres applications pour les langages de la famille $\mathcal{F}_{\mathsf{FDP}}$ mais également pour ceux de la famille $\mathcal{F}_{\mathsf{MQD}}$. Il conviendra alors de s'interroger sur les applications de telle ou telle famille de clauses. L'une de nos préoccupations principales reste la recherche de représentations toujours plus riches des arbres. Enfin, d'un point de vue expérimental, il sera intéressant de voir l'apport d'algorithmes de boosting là où notre approche gloutonne a échoué.

Chapitre 6

La PLI au service de la transformation

Sommaire

Le format de documents structurés XML est devenu un standard populaire, adopté par de nombreuses applications dans le cadre de la représentation, du stockage et de l'échange de documents et de données, en particulier sur le Web. Ainsi, les données sémantiquement riches comme les documents textuels ou multimédias sont aujourd'hui de plus en plus représentées à l'aide de formats semi-structurés et plus particulièrement de documents XML. Comme vu précédemment, les éléments présents dans ces documents sont organisés en fonction d'une structure qui reflète les relations logiques, sémantiques ou syntaxiques entre les différentes parties du document. Ces éléments, afin de constituer des documents valides du

point du vue du XML, respectent un ensemble de règles appelé *schéma* qui régit leurs formes en spécifiant les balises autorisées par exemple. Les éléments de ces structures demeurent spécifiques aux applications qui les manipulent. En effet, le XML est un méta-langage, les applications qui l'adoptent définissent et utilisent en réalité des langages fondés sur XML. Ils adoptent ainsi la syntaxe ainsi que la structure et les mécanismes définis par XML. Ils apportent de leur côté un vocabulaire (fonction du domaine auquel est associé le langage) permettant de décrire le contenu des documents de manière à rendre possible l'exploitation par les applications concernées. Chacune d'elles, malgré parfois un contexte commun, bénéficient de leurs propres structures. Dès lors, les données semi-structurées constituent une source hétérogène de documents XML ne partageant pas systématiquement le même schéma. Ces documents sont alors souvent convertis d'un langage à un autre ou combinés dans des documents utilisant plusieurs langages. Un exemple fréquent est l'affichage d'une page web sous un ordinateur ou sous un téléphone mobile. Les pages web destinées à ces périphériques sont généralement générées à partir d'informations issues d'une base de données, seules leurs structures diffèrent. Une page pour téléphone mobile possède une structure plus simple que celle disponible pour les ordinateurs afin de la rendre lisible sur un périphérique de petite taille. Il est ainsi possible de définir une transformation permettant le passage d'une page web normale à une page web mobile.

La gestion de cette hétérogénéité est un problème majeur pour l'exploitation de données semi-structurées. Il a déjà été soulevé depuis quelques années dans les domaines des Bases de Données et de la Recherche d'Information. Ceci s'explique par l'émergence du format XML notamment au travers des bases de données XML. Le besoin de transformer les documents pour les partager entre applications est devenu une préoccupation de premier plan tant pour la conception de systèmes utilisant les documents structurés que pour faciliter le partage d'informations entre applications. La problèmatique de la transformation de documents structurés peut être posée de la façon suivante :

> Étant donné un document d'une classe définie par une grammaire hors-contexte comme un schéma, quelle est la transformation, c'est-à-dire l'ensemble de règles qui utilisent des opérations de transformation de base, permettant de changer l'intégralité ou partie de ce document afin d'obtenir un document compatible avec une autre classe définie également par une grammaire et contenant les informations spécifiques à ce nouveau formalisme ?

Dans le cas de documents semi-structurés, une grammaire hors contexte est assimilable à un schéma pouvant être une DTD ou issu de langages comme XML *Schema* [Van der Vlist 2002] ou encore *Relax NG* [Van der Vlist 2004]. Un schéma définit alors, dans ce contexte, une famille d'arbres qui respecte un ensemble de contraintes. Il est ainsi fréquent de représenter un schéma à l'aide d'ensemble d'arbres.

Les transformations de documents XML sont habituellement définies manuellement en utilisant des langages complexes comme XSLT [Bray 2006]. Bien que des outils soient développés et existent pour aider les utilisateurs dans l'écriture de programmes de conversion, ce processus n'en demeure pas moins complexe et nécessite des compétences d'expert. Il est donc inadapté à des situations où les sources d'informations sont nombreuses et changent fréquemment. Cette émergence d'applications et de documents d'origines diverses met ainsi en avant l'importance d'un processus qui permettrait l'apprentissage automatique de transformations d'un format vers un autre.

Afin de résoudre ce problème, nous faisons le choix de ne considérer que les documents XML. Cette approche est assez naturelle puisqu'un schéma représente un ensemble de documents. Un document de cet ensemble présente l'avantage de proposer une instantiation d'un schéma et donc d'un contexte permettant d'apporter des informations absentes du

schéma. Ce choix a notamment été repris par les transducteurs d'arbres [Pankowski 2003, Maletti 2009, Razmara 2011] et de mots imbriqués bien formés [Alur 2012, Filiot 2012, Caralp 2013]. Ils permettent l'apprentissage de transformation arbres à arbres et mots imbriqués à mots imbriqués à partir d'ensemble d'arbres ou de mots imbriqués. Il est également motivé par le constat que les schémas de grandes collections sont souvent pauvres et ne spécifient que très peu de contraintes sur les documents valides. C'est notamment le cas du corpus INEX mais également des autres corpus présentés dans le chapitre 4. Dans ces conditions, l'écriture de feuilles XSLT pour la transformation de documents est très difficile, voire impossible, et très coûteuse en temps. Ce constat est d'autant plus vrai lorsque les documents pour lesquels une transformation doit être trouvée sont extraits du Web. En effet, beaucoup de sources, particulièrement sur le Web, ne possèdent pas de schéma ou n'en permettent pas l'accès. De ce fait, notre approche consiste à ne considérer qu'un ensemble de documents représentatifs des comportements autorisés dans ces structures. On cherche ensuite la transformation permettant le passage d'un document, voire une séquence de documents, en un autre, ou bien une autre séquence de documents de même taille.

Nous commençons ce chapitre en présentant dans la section 6.1 le problème de transformation adapté au cadre de la PLI. Nous précisons ensuite la forme des transformations apprises et introduisons la notion de script d'édition. Comme nous l'avons vu au cours des chapitres précédents, il est important en PLI de bénéficier de biais sémantiques et/ou syntaxiques propres au problème regardé. Ainsi, nous présentons également quelques biais présents dans la littérature dont l'emploi pour notre problème semble propice. Nous nous attaquons ensuite à un problème de transformation basique en section 6.2 et montrons la difficulté que représente cette tâche de transformation. Nous proposons ensuite en section 6.3 deux algorithmes qui permettent l'apprentissage de transformations en PLI et testons leurs efficacités expérimentalement en section 6.4. Enfin, en section 6.5, nous présentons quelques travaux vis-à-vis desquels nous nous positionnons.

6.1 La tâche de transformation

6.1.1 Approche du problème par la PLI

Nous nous intéressons uniquement à l'apprentissage de transformations à partir d'un ensemble d'exemples $E_{\mathcal{X}}$. Chaque exemple de $E_{\mathcal{X}}$ décrit la transformation d'un objet, appartenant à un ensemble d'objets $\mathcal{X}$, vers un autre objet de ce même ensemble. On a ainsi $E_{\mathcal{X}} \subseteq \mathcal{X}^2$. La tâche d'apprentissage consiste alors à inférer à partir de ces exemples une fonction $\mathsf{transfo} : \mathcal{X} \to \mathcal{X}$, appelée *transformation*, telle que $\forall (x, y) \in E_{\mathcal{X}},\ \mathsf{transfo}(x) = y$. Nous réalisons ainsi notre apprentissage à partir d'exemples positifs seuls.

La tâche de transformation visée portant sur des arbres, les objets de $\mathcal{X}$ en question seront donc naturellement des arbres ordonnés, d'arité non bornée et étiquetés sur un alphabet Σ donné (cf. chapitre 3 sous-section 3.1.3 pour la définition d'arbre) formant l'ensemble T_Σ. Cette tâche d'apprentissage de transformation d'arbres n'est pas facile. Pour cette raison, nous étudierons en préliminaire celle portant sur les mots. L'ensemble $\mathcal{X}$ correspondra dans ce cas à l'ensemble des mots $\Sigma^\star$ définis pour un alphabet donné Σ.

Afin de résoudre ce problème, nous faisons le choix d'utiliser la programmation logique inductive. Nous rattachons ce problème de transformation au cadre classique d'apprentissage en PLI vu lors du chapitre 3 et représentons les exemples de l'ensemble $E_{\mathcal{X}}$ par des atomes fondés issus d'un langage d'exemples $\mathcal{L}_e$. L'ensemble des atomes obtenus à partir de $E_{\mathcal{X}}$ forment l'ensemble E des exemples nécessaires à un problème d'apprentissage en PLI. Chaque atome fondé utilise un symbole de prédicat $\mathsf{transfo}/2$ d'arité binaire. Le premier argument d'un tel atome est un terme fondé qui décrit le premier objet avant transformation.

Le second argument est un terme également fondé représentant l'objet après transformation. Ainsi :

$$E = \{\mathsf{transfo}(\mathsf{object\text{-}to\text{-}term}(x), \mathsf{object\text{-}to\text{-}term}(y)) \mid (x, y) \in E_{\mathcal{X}}\}$$

où object-to-term est une fonction totale de codage sans perte permettant de représenter un objet de $\mathcal{X}$ sous la forme d'un terme fondé. Nous nous plaçons ainsi dans le cadre de la *sémantique définie* et plus précisément celle de l'*Example Setting* [Muggleton 1994b] vue lors du chapitre 3.

Dans le cas des mots, nous remplacerons la fonction object-to-term par la fonction word-to-term. Cette dernière permet la représentation d'un mot $w \in \Sigma^\star$ sous la forme d'un terme grâce à la représentation standard de liste en PLI. Ce terme utilise ainsi le symbole de prédicat $cons/2$ ainsi que la constante nil comme vu dans le chapitre 2 à la section 2.1. Ainsi :

$$\mathsf{word\text{-}to\text{-}term}(w) = \begin{cases} cons(u, \mathsf{word\text{-}to\text{-}term}(v)) & \text{si} & w = uv \text{ avec } u \in \Sigma \text{ et } v \in \Sigma^\star \\ nil & \text{sinon, i.e.} & w = \varepsilon \end{cases}$$

Dans le cas des arbres, un arbre ordonné étiqueté d'arité non bornée sera représenté par un terme usant des symboles de fonction $tree/2$ et $cons/2$ ainsi que de la constante $nil/0$. Le prédicat $tree/2$ est utilisé dans un terme qui représente un arbre. Il a pour premier argument un terme correspondant à l'étiquette de la racine de cet arbre. Le second argument est une la liste des termes représentant les sous-arbres de cette racine. On remplace ainsi la fonction object-to-term par la fonction tree-to-term suivante, elle transforme un arbre $t = (N_t, root_t, child_t, ns_t, lab^t_\Sigma)$ en un terme fondé :

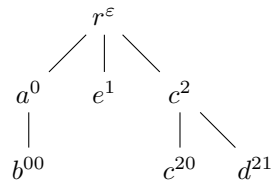
$$\mathsf{tree\text{-}to\text{-}term}(t) \;=\; tree(p, cons(s_0, cons(s_1, \cdots cons(s_n, nil) \cdots)))$$

où $p = lab^t_\Sigma(root_t)$ et $s_i = \mathsf{tree\text{-}to\text{-}term}(subtree_t(root_t.i))$ avec $root_t.i \in N_t$ et $i \in \mathbb{N}$. L'exemple 54 illustre ces constructions.

Exemple 54. *Soit le mot "chat" construit à partir de l'alphabet latin, sa représentation sous forme de terme fondé est la suivante :*

$$cons(c, cons(h, cons(a, cons(t, nil))))$$

Soit l'arbre $t \in T_\Sigma$ avec $\Sigma = \{r, a, b, c, d, e\}$.



$t \in T_\Sigma$

Le terme représentant cet arbre est le suivant :

$$tree(r, cons(tree(a, cons(tree(e, nil), nil)), cons(tree(e, nil)cons(tree(c, cons(tree(c, nil), cons(tree(d, nil), nil))), nil))))$$

où r, a, b, c et d sont des constantes.

Le but de l'apprentissage est alors d'apprendre une ou plusieurs hypothèses h du langage d'hypothèses $\mathcal{L}_h$. Elles formeront une définition du symbole de prédicat transfo/2 et devront

être consistantes avec les exemples de E ainsi qu'avec la théorie du domaine T, i.e. $\forall e \in E,\ h \wedge T \vdash e$. La théorie du domaine sera bien évidemment à définir en fonction du problème. Comme vu avec la classification, il est nécessaire pour des raisons de complexité de réduire l'espace de recherche d'un problème de PLI. Cette démarche est réalisée à l'aide de biais syntaxiques et sémantiques appliqués au langage des hypothèses et à celui de la théorie du domaine. Nous présenterons dans ce qui suit quelques biais intéressants pour notre problème puis définirons les langages d'hypothèses et de théorie du domaine envisagés en fonction de ces derniers.

6.1.2 Biais sémantiques pour la restriction de l'espace de recherche

Nous avons vu dans les chapitres précédents des biais syntaxiques dont le but est de limiter la forme des clauses autorisées comme hypothèses. C'est notamment le cas des clauses i, j-déterminées de [Muggleton 1990] dont la profondeur et le degré sont bornés par des constantes ou bien encore les k-locales de [Kietz 1994a]. Une autre forme de biais restreignant l'espace des hypothèses est le *biais sémantique*. Son but est de réduire la taille de l'espace de recherche en limitant le comportement des clauses. Le déterminisme [Quinlan 1991] ainsi que notre quasi-déterminisme et multi-quasi-détermisme en font partie. D'autres biais sémantiques ont également été considérés, notamment les notions de *déclaration de mode*, *type* ainsi que de *programme fonctionnel*.

Déclaration de mode

La notion de déclaration de mode a été utilisée dans plusieurs systèmes d'apprentissage comme Progol [Muggleton 1995], ProGolem [Muggleton 2010], ALEPH [Srinivasan 2004], April [Santos 2003] ou bien HOC-learner [Santos 2009]. Une déclaration de mode permet de contraindre l'utilisation d'un symbole de prédicat dans une clause en imposant la nature de ses arguments appelée *type*. Il permet également de préciser si les arguments sont destinés à être des *sorties*, des *entrées* ou des *paramètres*. Ceci a pour but de ramener l'utilisation d'un symbole de prédicat à celle d'une fonction.

Un *type* correspond à un ensemble dénombrable (possiblement infini) de termes fondés construits à partir d'un ensemble (possiblement infini) de symboles de fonction et de constantes. Dans le cas des mots, on note $\mathcal{T}_{\Sigma^\star}$ l'ensemble des termes représentant l'ensemble des mots construits à partir d'un alphabet Σ c'est-à-dire :

$$\mathcal{T}_{\Sigma^\star} = \{\mathsf{word\text{-}to\text{-}term}(x) \mid \forall x \in \Sigma^\star\}$$

Tandis que dans le cas des arbres, on note $\mathcal{T}_{T_\Sigma}$ l'ensemble des termes représentant l'ensemble des arbres dont les nœuds sont étiquetés sur un alphabet Σ :

$$\mathcal{T}_{T_\Sigma} = \{\mathsf{tree\text{-}to\text{-}term}(x) \mid \forall x \in T_\Sigma\}$$

La taille d'un type $\mathcal{T}$, notée $|\mathcal{T}|$, est le nombre d'éléments que contient cet ensemble.

Les *entrées* $(+)$, *paramètres* $(\#)$ ou *sorties* $(-)$ d'un symbole de prédicat ou d'un symbole de fonction p/n sont des entiers de $\{1, \cdots, n\}$ tels qu'il existe un mapping des entiers $\{1, \cdots, n\}$ vers $\{+, \#, -\}$. Ils font référence à la position d'arguments d'un atome de la forme $p(t_1, \cdots, t_n)$ ou d'un terme $f(t_1, \cdots, t_n)$. Par abus de langage, nous appellerons également ces arguments entrées, paramètres et sorties. Les entrées et les sorties sont des variables tandis que les paramètres sont des termes fondés. Il est possible de définir plusieurs triplets entrées/paramètres/sorties pour un même symbole de prédicat.

Exemple 55. *Soit l'opérateur mathématique* $+ : \mathbb{Z} \times \mathbb{Z} \to \mathbb{Z}$*. Nous pouvons définir le symbole de prédicat* $plus/3$ *correspondant. L'argument* t_3 *d'un atome de la forme* $plus(t_1, t_2, t_3)$ *est la somme des deux premiers arguments de cet atome. L'addition a comme arguments des rationnels* $\mathbb{Z}$*, nous pouvons alors définir le type* $\mathcal{T}_{\mathbb{Z}} = \{i/0 \mid i \in \mathbb{Z}\}$ *les représentant. Enfin, il est possible de définir plusieurs triplets entrées/paramètres/sorties :* $\{1,2\}/\emptyset/\{3\}$*,* $\{1\}/\{2\}/\{3\}$*,* $\emptyset/\{1,2,3\}/\emptyset$*,* $\{2\}/\{1\}/\{3\}$*,* $\{1,3\}/\emptyset/\{2\}$*, ...*

Nous donnons maintenant la définition de mode proposée par [Srinivasan 2004] et reprise dans la plupart des systèmes cités ci-dessus :

Définition 90 (Mode d'un symbole de fonction)**.** *Un* mode d'un symbole de fonction f/n *spécifie l'utilisation du symbole de fonction* f/n *en précisant le type des arguments ainsi que les entrées, sorties et paramètres d'un terme de la forme* $f(t_1, \cdots, t_n)$*. Il peut être* simple *ou* structuré*. Un* mode simple d'un terme *est de la forme :*

- $+\mathcal{T}$ *signifie que l'argument est une entrée de type* $\mathcal{T}$*,*
- $-\mathcal{T}$ *signifie que l'argument est une sortie de type* $\mathcal{T}$*,*
- $\#\mathcal{T}$ *signifie que l'argument est un paramètre de type* $\mathcal{T}$*.*

On note $M_{f/m}$ *un mode du symbole de fonction* f/m*. Un* mode structuré d'un symbole de fonction *est de la forme :*

$$f(M_{g_1/m_1}, \cdots, M_{g_n/m_n})$$

où f/n *est un symbole de fonction d'arité* n *et* $M_{g_1/m_1}, \cdots, M_{g_n/m_n}$ *sont des modes simples ou structurés de symboles de fonction.*

La notion de mode peut ensuite être étendue à celle de symbole de prédicat.

Définition 91 (Mode de prédicat et Déclaration de mode de prédicat)**.** *Un* mode de prédicat p/n*, noté* $M_{p/n}$*, est de la forme :*

$$p(M_{f_1/m_1}, \cdots, M_{f_n/m_n})$$

où $M_{f_1/m_1}, \cdots, M_{f_n/m_n}$ *sont des modes de symboles de fonction* $f_1/m_1, \cdots, f_n/m_n$*.*
Une déclaration de mode d'un prédicat p/n *est de la forme :*

$$modeh(Recall, M_{p/n})$$

s'il porte sur un atome de tête d'une clause ou bien de la forme :

$$modeb(Recall, M_{p/n})$$

s'il porte sur un littéral du corps d'une clause où Recall est un entier compris entre 1 *et* ∞ *dont le but est de limiter le non-déterminisme du prédicat* p/n*. Seules les* $Recall^{ièmes}$ *premières instanciations des sorties seront considérées lorsque les entrées et paramètres sont instanciés par des termes fondés.*

Exemple 56. *Reprenons l'exemple 55 et donnons un exemple d'interactions possibles entre plusieurs déclarations de mode. Nous savons que l'opération* $+$ *est fonctionnelle, il en est donc de même pour le prédicat* $plus/3$*. Nous définissons ainsi les modes de prédicat suivants pour le prédicat* $plus/3$ *:*

- $plus(+\mathcal{T}_{\mathbb{Z}}, +\mathcal{T}_{\mathbb{Z}}, -\mathcal{T}_{\mathbb{Z}})$
- $plus(+\mathcal{T}_{\mathbb{Z}}, \#\mathcal{T}_{\mathbb{Z}}, -\mathcal{T}_{\mathbb{Z}})$
- $plus(\#\mathcal{T}_{\mathbb{Z}}, +\mathcal{T}_{\mathbb{Z}}, -\mathcal{T}_{\mathbb{Z}})$
- $plus(-\mathcal{T}_{\mathbb{Z}}, +\mathcal{T}_{\mathbb{Z}}, +\mathcal{T}_{\mathbb{Z}})$

— $plus(-\mathcal{T}_{\mathbb{Z}}, +\mathcal{T}_{\mathbb{Z}}, \#\mathcal{T}_{\mathbb{Z}})$
— $plus(-\mathcal{T}_{\mathbb{Z}}, \#\mathcal{T}_{\mathbb{Z}}, +\mathcal{T}_{\mathbb{Z}})$
— $plus(+\mathcal{T}_{\mathbb{Z}}, -\mathcal{T}_{\mathbb{Z}}, +\mathcal{T}_{\mathbb{Z}})$
— $plus(\#\mathcal{T}_{\mathbb{Z}}, -\mathcal{T}_{\mathbb{Z}}, +\mathcal{T}_{\mathbb{Z}})$
— $plus(+\mathcal{T}_{\mathbb{Z}}, -\mathcal{T}_{\mathbb{Z}}, \#\mathcal{T}_{\mathbb{Z}})$

L'opération de soustraction pour les rationnels s'utilise de la même manière que l'addition précédemment définie. Le prédicat moins/3 associé à l'opération de soustraction dispose de modes de prédicat similaires à ceux de plus/3. Nous sommes maintenant capables de définir récursivement la multiplication à l'aide des opérations d'addition et de soustraction. En supposant que nous bénéficions de définitions pour plus/3 et moins/3, nous obtenons la définition suivante pour le prédicat de la multiplication mult/3 :

$mult(0, X, 0).$
$mult(X, Y, Z) \quad \leftarrow \quad moins(X, 1, X1),\ mult(X1, Y, Z1),\ plus(Y, Z1, Z).$

Ce programme peut être associé aux déclarations de modes suivantes régissant l'usage de ce dernier :

— $modeb(1, plus(+\mathcal{T}_{\mathbb{Z}}, +\mathcal{T}_{\mathbb{Z}}, -\mathcal{T}_{\mathbb{Z}}))$
— $modeb(1, moins(+\mathcal{T}_{\mathbb{Z}}, \#\mathcal{T}_{\mathbb{Z}}, -\mathcal{T}_{\mathbb{Z}}))$
— $modeb(1, mult(+\mathcal{T}_{\mathbb{Z}}, +\mathcal{T}_{\mathbb{Z}}, -\mathcal{T}_{\mathbb{Z}}))$
— $modeh(1, mult(\#\mathcal{T}_{\mathbb{Z}}, \#\mathcal{T}_{\mathbb{Z}}, -\mathcal{T}_{\mathbb{Z}}))$

Il est possible de simplifier la notion de mode de prédicat. Une version a ainsi été proposée dans le système April [Santos 2003] où un mode de prédicat se limite à l'usage de modes simples sans type. Le mode d'un prédicat p/n est alors un mapping des entiers $\{1, \cdots, n\}$ vers $\{+, -, \#\}$.

Dans nos travaux, nous utiliserons principalement des modes simples. Nous appliquons maintenant la définition de déclaration de mode à notre tâche d'apprentissage de transformation. S'en suit l'exemple suivant :

Exemple 57. *Dans le cadre de l'apprentissage de transformation d'un objet $x \in \mathcal{X}$ vers un objet $y \in \mathcal{X}$, le prédicat* transfo/2 *disposera des déclarations de mode suivantes :*

$$modeh(1, \mathsf{transfo}(+\mathcal{T}_{\mathcal{X}}, -\mathcal{T}_{\mathcal{X}}))$$
$$modeh(1, \mathsf{transfo}(\#\mathcal{T}_{\mathcal{X}}, \#\mathcal{T}_{\mathcal{X}}))$$

La deuxième déclaration de mode définit une clause propre à un seul exemple puisque les entrées sont des termes fondés. Ces déclarations peuvent être adaptées aux cas de la transformation de mots et d'arbres comme ce qui suit :

$$modeh(1, \mathsf{transfo}(+\mathcal{T}_{\Sigma^\star}, -\mathcal{T}_{\Sigma^\star}))$$
$$modeh(1, \mathsf{transfo}(\#\mathcal{T}_{\Sigma^\star}, \#\mathcal{T}_{\Sigma^\star}))$$
$$modeh(1, \mathsf{transfo}(+\mathcal{T}_{T_\Sigma}, -\mathcal{T}_{T_\Sigma}))$$
$$modeh(1, \mathsf{transfo}(\#\mathcal{T}_{T_\Sigma}, \#\mathcal{T}_{T_\Sigma}))$$

La valeur de Recall est à 1 puisque les transformations étudiées sont déterministes. Dans le cas où une transformation est définie de manière récursive, il est intéressant d'y ajouter la déclaration de mode suivante :

$$modeb(1, \mathsf{transfo}(+\mathcal{T}_{\mathcal{X}}, -\mathcal{T}_{\mathcal{X}}))$$

Elle rend possible l'apprentissage de transformations récursives si la théorie contient une définition pour ce prédicat ou si les définitions découvertes pour ce prédicat lors d'un apprentissage disjonctif sont ajoutées au fur et à mesure à la théorie du domaine.

Les définitions de mode de fonction, de mode de prédicat et de déclaration de mode permettent de limiter la forme des atomes dans une clause. Cependant, les interactions possibles entre les atomes d'une même clause ne sont pas précisées. Ce point est réglé avec la définition de *respect d'un mode ou de déclaration de mode.*

Définition 92 (Respect d'un mode de fonction, d'un mode de prédicat, d'une déclaration de mode)**.** Un atome respecte un mode d'un prédicat $M_{p/n}$ (resp. un mode de fonction $M_{f/n}$) *si et seulement s'il est de la forme* $p(t_1, \cdots, t_n)$ *(resp.* $f(t_1, \cdots, t_n)$*) où* $t_1, \cdots, t_n$ *sont des termes obtenus en replaçant les arguments d'entrée* $(+)$ *et de sortie* $(-)$ *par des variables et les paramètres* $(\#)$ *par des termes fondés du type correspondant.*

Soit un ensemble de déclarations de mode M*.* Une clause définie $C = h \leftarrow b_1, \cdots, b_n$ respecte M *si et seulement si :*

1. *l'atome* h *respecte un mode de* M *de la forme* $modeh(Recall_h, M_{\mathsf{ps}(h)})$*,*
2. *chaque* b_i *respecte un mode de* M *de la forme* $modeb(Recall_{b_i}, M_{\mathsf{ps}(b_i)})$*,*
3. *chaque terme associé à une entrée* $+\mathcal{T}$ *dans un atome* b_j *est soit une entrée* $+\mathcal{T}$ *dans l'atome* h *soit une sortie* $-\mathcal{T}$ *dans un littéral* b_i *avec avec* $1 \leqslant i < j$*.*

où $\mathsf{ps}(A)$ *est le symbole de prédicat d'un atome* A*.*

Une théorie T respecte M *si chaque clause de* T *respecte cet ensemble de déclarations de mode* M*.*

Le langage de clauses défini à partir d'un ensemble de déclarations de mode M*, noté* $\mathcal{L}_M$*, est l'ensemble de toutes les clauses respectant* M *et basées sur l'utilisation des symboles de prédicat, des symboles de fonction et des constantes des déclarations de mode de* M*.*

La notion de mode de prédicat ainsi que celle de déclaration de mode imposent pour un symbole de prédicat un triplet entrées/paramètres/sorties. Afin d'en disposer facilement pour un atome $A = p(t_1, \cdots, t_n)$ respectant un mode M du prédicat p/n ou une déclaration de mode $M_{p/n}$, nous notons respectivement $\mathsf{input}^M_{\mathcal{T}}(A)$ l'ensemble des entrées, $\mathsf{param}^M_{\mathcal{T}}(A)$ l'ensemble des paramètres et $\mathsf{output}^M_{\mathcal{T}}(A)$ l'ensemble des sorties de type $\mathcal{T}$ de A vis-à-vis de M. Nous omettons l'argument M lorsque le contexte le permet afin de simplifier la notation. Nous supprimons également le type de ces notations pour représenter l'ensemble de toutes les entrées, de tous les paramètres et de toutes les sorties tous types confondus d'un atome. Ces notations peuvent être étendues à celles d'un ensemble d'atomes S par rapport à un ensemble de déclarations de mode M. Nous notons ainsi $\mathsf{input}^M_{\mathcal{T}}(S) = \{t \in \mathsf{input}^m_{\mathcal{T}}(\ell) \mid \ell \in S \text{ et } m \in M\}$, $\mathsf{param}^M_{\mathcal{T}}(S) = \{t \in \mathsf{param}^m_{\mathcal{T}}(\ell) \mid \ell \in S \text{ et } m \in M\}$ et $\mathsf{output}^M_{\mathcal{T}}(S) = \{t \in \mathsf{output}^m_{\mathcal{T}}(\ell) \mid \ell \in S \text{ et } m \in M\}$ l'ensemble des entrées, paramètres et sorties de type $\mathcal{T}$ des atomes de cet ensemble. Comme précédemment, nous omettrons le typage $\mathcal{T}$ pour autoriser n'importe quel type et ne spécifierons pas l'ensemble de déclarations de mode M lorsque le contexte est sans ambiguité.

Ces définitions jouent un rôle important dans la réduction de l'espace de recherche en limitant par le respect de déclarations de mode la forme syntaxique des clauses. En effet, une clause dont chaque littéral contient au moins une entrée et qui respecte un ensemble de déclarations de mode est obligatoirement une clause dont les littéraux sont connectés. La notion de déclaration de mode ne limite cependant pas réellement le nombre d'instanciations des sorties en fonction des entrées. Nous nous y intéressons dans ce qui suit et plus particulièrement à la notion de fonctionnalité le permettant.

Fonctionnalité

Malgré la présence de déclarations de mode, l'espace de recherche peut demeurer grand dans le cas où les modes autorisés ont une valeur de *Recall* fixée à l'infini. Afin de l'éviter, de

nombreux travaux ont étudié la notion de fonctionnalité pour les programmes logiques. Le but est de borner le nombre d'instanciation des sorties en fonction des entrées et paramètres sans avoir à omettre des instanciations de sorties du fait de la valeur de *Recall*.

Nous retrouvons ainsi les travaux de [Paulson 1991, Bergadano 1993b, Bergadano 1993a, Hernández-orallo 1999] avec, entre autres, le système FILP permettant la synthèse de programme logique fonctionnel ou encore FFOIL de [Quinlan 1996]. Selon [Bergadano 1993b], un programme est *fonctionnel* si chaque symbole de prédicat p/n utilisé dans ce programme peut être associé à une fonction totale telle que :

— m arguments sont des entrées (avec $0 < m < n$),
— $n - m$ arguments sont des sorties,
— une instanciation des entrées produit une unique instanciation des arguments de sortie.

La notion de fonctionnalité est proche de celle de mode. Ainsi, nous extrapolons la notion de fonctionnalité à celle de mode de prédicat afin de constituer les définitions de *mode de prédicat fonctionnel* et de *déclaration de mode fonctionnelle* suivantes.

Définition 93 (Mode de prédicat fonctionnel et Déclaration de mode fonctionnelle)**.** *Un mode de prédicat $M_{p/n}$ est fonctionnel par rapport à une théorie T si pour tout atome $A = p(t_1, \cdots, t_n)$ respectant le mode $M_{p/n}$ et une substitution θ liant chaque variable d'entrée de A à un terme fondé du type correspondant, il existe, au plus, une substitution fondée θ' telle que $T \vdash A\theta\theta'$.*

Une déclaration de mode fonctionnelle *par rapport à une théorie T est une déclaration de mode dont le mode de prédicat est fonctionnel par rapport à T.*

Le langage de clauses défini à partir de déclarations de mode fonctionnelles M_T par rapport à une théorie T, noté $\mathcal{L}^{fun}_{M_T}$, est l'ensemble de toutes les clauses respectant M_T et basées sur l'utilisation des symboles de prédicat, des symboles de fonction et des constantes des déclarations de mode fonctionnelles de M_T.

Le respect d'un mode de prédicat fonctionnel par un atome assure que l'atome dont les entrées et paramètres sont instanciés par des termes fondés du type associé ne possède, au plus, qu'une instanciation possible pour ses sorties. La valeur de *Recall* de la déclaration de mode associée est alors obligatoirement de 1 puisqu'au plus une instanciation des sorties est possible.

La notion de fonction est présente naturellement dans de nombreux problèmes. C'est notamment le cas dans notre problème de transformation. Nous la retrouvons plus particulièrement dans la notion d'opération élémentaire définie dans la sous-section suivante.

6.1.3 Opérations élémentaires et Script d'édition

Afin de représenter une transformation d'un élément d'un ensemble $\mathcal{X}$ vers un autre élément de cet ensemble, nous utiliserons des opérations d'édition élémentaires sur $\mathcal{X}$. On appelle *opération d'édition sur un ensemble* $\mathcal{X}$ une fonction $\delta : \mathcal{X} \to \mathcal{X}$. Soit un ensemble d'opérations d'édition $\Delta = \{\delta_1, \cdots, \delta_n\}$. Un *script d'édition* est une séquence d'opérations d'édition $S = (\delta_{i_1}, \cdots, \delta_{i_m})$ avec $\{i_1, \cdots, i_m\} \subseteq \{1, \cdots, n\}$. On note $|S|$ la longueur du script d'édition S. La transformation issue d'un script d'édition $S = (\delta_{i_1}, \cdots, \delta_{i_m})$ est la fonction $S : \mathcal{X} \to \mathcal{X}$ telle que $S = (\delta_{i_1} \circ \cdots \circ \delta_{i_m})$ (composition au sens relationnel). On note $\mathcal{S}_\Delta$ l'ensemble de tous les scripts d'édition construits sur Δ. Deux scripts d'édition $S, S' \in S_\Delta$ sont *équivalents sur un ensemble d'éléments* $\mathcal{X}$, noté $S \equiv_{\mathcal{X}} S'$, si et seulement si $\forall x \in \mathcal{X}.\ S(x) = S'(x)$.

L'application d'une opération sur un objet est plus ou moins facile à réaliser. Un coût lui est alors imputé en fonction de l'importance de la transformation qui en découle. On note

$\mathsf{cost}_\delta : \mathcal{X} \to \mathbb{N}$ la fonction du coût d'application de l'opération δ pour un ensemble d'objets $\mathcal{X}$. Le coût d'un script $S \in \mathcal{S}_\Delta$ sur un objet $x \in \mathcal{X}$ est $\mathsf{cost}_S(x) = \mathsf{cost}_{\delta_{i_1}}(x) + \mathsf{cost}_{\delta_{i_2}}(\delta_{i_1}(x)) + \cdots + \mathsf{cost}_{\delta_{i_m}}(\delta_{i_1} \circ \cdots \circ \delta_{i_{m-1}}(x))$. Il peut exister plusieurs scripts d'édition $S_1, \cdots, S_n$ tels que, pour un élément $x \in \mathcal{X}$, on ait $S_1(x) = \cdots = S_n(x)$. Un script $S = (\delta_{i_1}, \cdots, \delta_{i_m})$ défini pour deux éléments $x, y \in \mathcal{X}$ est *optimal* si $\nexists S' \in S_\Delta$ tel que $S(x) = S'(x) = y$ et $\mathsf{cost}_{S'}(x) < \mathsf{cost}_S(x)$. Il peut exister plusieurs scripts optimaux réalisant la transformation d'un élément x vers un élément y de $\mathcal{X}$. Afin de mieux visualiser l'ensemble des scripts d'édition portant sur un ensemble $\mathcal{X}$ et basés sur l'utilisation d'un ensemble d'opérations d'édition Δ, nous introduisons une représentation graphique de ces derniers.

Représentation graphique

Nous introduisons maintenant une représentation sous forme de graphe dirigé de l'ensemble des scripts d'édition portant sur un ensemble $\mathcal{X}$ et utilisant des opérations Δ.

Définition 94 (Graphe de Transformation)**.** *Le graphe de transformation des éléments d'un ensemble $\mathcal{X}$ par des opérations élémentaires d'édition définies sur $\mathcal{X}$ d'un ensemble d'opérations d'édition Δ sur $\mathcal{X}$ est un tuple $G = (N, E, lab_\Delta)$ où N est l'ensemble des nœuds égal à $\mathcal{X}$, $E \subseteq N \times N$ est l'ensemble des arêtes de ce graphe et $lab_\Delta : E \to 2^\Delta$ est une fonction d'étiquetage des arêtes qui associe une arête à un sous-ensemble de Δ telle que :*

1. $\forall (x, y) \in E \ \forall \delta \in lab_\Delta((x, y)),\ \delta(x) = y$
2. $\forall (x, y), (x', y') \in E$ *si* $x = x'$ *ou* $y = y'$ *et que* $lab_\Delta((x, y)) \cap lab_\Delta((x', y')) \neq \emptyset$*, alors* $x = x'$ *et* $y = y'$*.*

Certains concepts standards aux graphes vont nous être utiles par la suite. Nous les rappelons donc et les adaptons, si nécessaire, à nos graphes de transformation. Un *chemin entre deux points* x_1 et x_k d'un graphe G est une séquence d'arêtes $(x_1, x_2), (x_2, x_3), \cdots, (x_{k-1}, x_k)$ issue de ce graphe avec $x_1, \cdots, x_k \in N$. Les étiquettes $lab_\Delta((x, y))$ associées à une arête $(x, y) \in E$ d'un graphe de transformation G représentent l'ensemble des opérations élémentaires applicables à l'élément x afin d'obtenir l'élément y. Si un chemin lie deux points x et y, alors au moins un script d'édition assure le passage du nœud x au nœud y. Afin de différencier ces scripts d'édition, nous appellerons *chemin de transformation* dans un graphe G une séquence de triplets $p = (x_1, \delta_{i_1}, x_2), (x_2, \delta_{i_2}, x_3), \cdots, (x_{k-1}, \delta_{i_{k-1}}, x_k) \in (N \times \Delta \times N)^\star$ pour laquelle $(x_1, x_2), (x_2, x_3), \cdots, (x_{k-1}, x_k)$ est un chemin de G et $\forall j \in [1..k-1],\ \delta_{i_j} \in lab_\Delta(x_j, x_{j+1})$. Ce chemin de transformation correspond à un script d'édition $S_p = (\delta_{i_1}, \cdots, \delta_{i_{k-1}})$ pour lequel $S(x_1) = x_k$.

Nous définissons maintenant quelques opérations élémentaires classiques portant sur les mots puis d'autres portant sur les arbres.

Opération élémentaire et de mode fonctionnelle

La notion d'opération élémentaire posée, nous pouvons faire le lien avec celle de mode fonctionnelle. Nous avons vu qu'une opération élémentaire est une fonction de $\mathcal{X}$ vers $\mathcal{X}$. Ce genre de fonction dispose d'entrées et de sorties et peut donc être représentée à l'aide de modes de prédicat fonctionnels et d'une théorie. Ainsi, une fonction $p : E_1 \times \cdots \times E_{n-1} \to S$ qui dispose de $n-1$ entrées est représentable à l'aide d'un programme logique $T_{p/n}$ associé au mode de prédicat suivant :

$$M_{p/n} = p(+\mathcal{T}_{E_1}, \cdots, +\mathcal{T}_{E_{n-1}}, -\mathcal{T}_S)$$

Il est possible pour une fonction d'instancier certaines de ses entrées. Ceci a pour effet de réduire son espace de départ. Ce phénomène se traduit dans le mode de prédicat par la

présence de paramètres. L'instanciation partielle des i premiers arguments de la fonction p se traduit par exemple par le mode de prédicat :

$$M_{p/n} = p(\#\mathcal{T}_{E_1}, \cdots, \#\mathcal{T}_{E_i}, +\mathcal{T}_{E_{i+1}}, \cdots, +\mathcal{T}_{E_{n-1}}, -\mathcal{T}_S)$$

que nous représenterons sans perte de généralité par la fonction :

$$p_{t_1,\cdots,t_{i-1}} : \mathcal{T}_{E_{i+1}} \times \cdots \times \mathcal{T}_{E_{n-1}} \to \mathcal{T}_S$$

où $t_1 \in E_1, \cdots, t_i \in E_i$.

Nous appelons ainsi *prédicat de transformation* un symbole de prédicat qui représente une opération élémentaire d'édition définie sur un ensemble $\mathcal{X}$ et *atome de transformation* un atome dont le symbole de prédicat est un prédicat de transformation. Ce dernier doit respecter une déclaration de mode fonctionnelle associée à une théorie. Remarquons que la déclaration de mode d'un tel prédicat a une sortie de type $\mathcal{T}_\mathcal{X}$ et a exactement une de ses entrées dont le type est $\mathcal{T}_\mathcal{X}$. Nous notons T_Δ l'ensemble des définitions de prédicats de transformation issu d'un ensemble d'opérations élémentaires Δ et M_Δ l'ensemble des déclarations de mode fonctionnelles associé à ces prédicats de transformation.

Pour les mots

Avant de lister les différentes transformations élémentaires envisagées pour les mots ainsi que leurs déclarations de mode fonctionnelles associées, rappelons quelques concepts de bases portant sur les mots. Un alphabet Σ est un ensemble fini de symboles. Un mot $w = w_0 w_1 w_2 \cdots w_n$ est une séquence finie de symboles de Σ telle que $\forall k \in [0..n],\ w_k \in \Sigma$. Le symbole w_k est le k^{e} symbole du mot w. Le mot vide est symbolisé par ε. On note $|w|$ la taille du mot w, c'est-à-dire son nombre de lettres, et $|w|_{l\in\Sigma}$ le nombre d'occurences du symbole l dans le mot w. L'ensemble des mots, y compris le mot vide, construit à partir de l'alphabet Σ est noté Σ^*. On note w_i^j le sous-mot issu de w contenant les symboles de w compris entre i et j inclus avec $0 \leqslant i < j < |w|$.

Dans la suite, nous considérons pour l'ensemble des mots Σ^* les opérations d'édition élémentaires suivantes :

— l'*insertion* d'un symbole a à la position i :

$$\mathsf{ins}_{i,a}(w) = w_0 \cdots w_{i-1}\ a\ w_i \cdots w_n$$

— la *suppression* du symbole à la position j :

$$\mathsf{del}_j(w) = w_0 \cdots w_{j-1}\ w_{j+1} \cdots w_n$$

— la *substitution* du symbole à la position j par un symbole a :

$$\mathsf{sub}_{j,a}(w) = w_0 \cdots w_{j-1}\ a\ w_{j+1} \cdots w_n$$

où $w = w_0 w_1 \cdots w_n \in \Sigma^*,\ i \in [0..|w|],\ j \in [0..|w|[$ et $a \in \Sigma$. Ces opérations sont standards dans la littérature et permettent, par composition, la construction de n'importe quel mot de $\Sigma^\star$ à partir d'un autre mot de $\Sigma^\star$. Chacune de ces opérations a une complexité linéaire en temps dans la taille du mot d'entrée. Les atomes issus de ces fonctions sont associés respectivement aux modes de prédicat fonctionnels suivants :

— $\mathsf{ins}(+\mathcal{T}_{\Sigma^\star}, \#\mathcal{T}_{|w|-1}, \#\mathcal{T}_\Sigma, -\mathcal{T}_{\Sigma^\star})$
— $\mathsf{del}(+\mathcal{T}_{\Sigma^\star}, \#\mathcal{T}_{|w|}, -\mathcal{T}_{\Sigma^\star})$
— $\mathsf{sub}(+\mathcal{T}_{\Sigma^\star}, \#\mathcal{T}_{|w|-1}, \#\mathcal{T}_\Sigma, -\mathcal{T}_{\Sigma^\star})$

avec $\mathcal{T}_{\Sigma^\star}$ l'ensemble des termes fondés construit à partir des mots de $\Sigma^\star$, $\mathcal{T}_i = \{0/0, \cdots, i/0 \mid i \in \mathbb{N}\}$ l'ensemble des constantes représentant les entiers de 1 à i et $\mathcal{T}_\Sigma = \{a/0 \mid a \in \Sigma\}$ l'ensemble des constantes représentant les symboles de Σ. D'autres modes fonctionnels sont définissables à partir de ces fonctions. Nous retrouvons notamment :

- $\mathsf{ins}(\#\mathcal{T}_{\Sigma^\star}, \#\mathcal{T}_{|w|-1}, \#\mathcal{T}_\Sigma, -\mathcal{T}_{\Sigma^\star})$
- $\mathsf{ins}(+\mathcal{T}_{\Sigma^\star}, \#\mathcal{T}_{|w|-1}, \#\mathcal{T}_\Sigma, \#\mathcal{T}_{\Sigma^\star})$
- $\mathsf{ins}(\#\mathcal{T}_{\Sigma^\star}, \#\mathcal{T}_{|w|-1}, \#\mathcal{T}_\Sigma, \#\mathcal{T}_{\Sigma^\star})$
- ...
- $\mathsf{del}(\#\mathcal{T}_{\Sigma^\star}, \#\mathcal{T}_{|w|}, -\mathcal{T}_{\Sigma^\star})$
- $\mathsf{del}(+\mathcal{T}_{\Sigma^\star}, \#\mathcal{T}_{|w|}, \#\mathcal{T}_{\Sigma^\star})$
- ...
- $\mathsf{sub}(\#\mathcal{T}_{\Sigma^\star}, \#\mathcal{T}_{|w|-1}, \#\mathcal{T}_\Sigma, -\mathcal{T}_{\Sigma^\star})$
- $\mathsf{sub}(+\mathcal{T}_{\Sigma^\star}, \#\mathcal{T}_{|w|-1}, \#\mathcal{T}_\Sigma, \#\mathcal{T}_{\Sigma^\star})$
- ...

Ils peuvent être utilisés à la fois pour définir des déclarations de mode fonctionnelles portant sur des atomes de têtes et des atomes du corps d'une clause.

Nous considérons que le coût des opérations d'édition que sont la substitution, la suppression et l'insertion est indépendant de la position et de la lettre. Nous leur imposons ainsi un même coût. On note respectivement $\mathsf{cost}_{\mathsf{sub}}$, $\mathsf{cost}_{\mathsf{del}}$ et $\mathsf{cost}_{\mathsf{ins}}$ le coût d'édition d'une opération de substitution, d'insertion et de suppression. On a alors : $\forall w \in \Sigma^* \ \forall i \in [0..|w|] \ \forall j, k \in [0..|w|[\ \forall a, b \in \Sigma, \mathsf{cost}_{\mathsf{ins}_{i,a}}(w) = \mathsf{cost}_{\mathsf{del}_j}(w) = \mathsf{cost}_{\mathsf{sub}_{k,b}}(w) = 1$.

Il est possible d'envisager une multitude d'opérations élémentaires portant sur les mots. Nous pouvons ainsi définir des méta-opérations basées sur les opérations élémentaires standards. Nous envisageons par exemple les méta-opérations suivantes :

- $\mathsf{sub\_beg}_{a,b} : \Sigma^\star \to \Sigma^\star$, $\mathsf{sub\_end}_{a,b} : \Sigma^\star \to \Sigma^\star$ et $\mathsf{sub\_all}_{a,b} : \Sigma^\star \to \Sigma^\star$ avec $a, b \in \Sigma$. Ces opérations consistent à renommer respectivement la première lettre a, la dernière lettre a et chaque lettre a du mot fourni en entrée par une lettre b,
- $\mathsf{del\_beg}_a : \Sigma^\star \to \Sigma^\star$, $\mathsf{del\_end}_a : \Sigma^\star \to \Sigma^\star$ et $\mathsf{del\_all}_a : \Sigma^\star \to \Sigma^\star$ avec $a, b \in \Sigma$. Ces opérations consistent à supprimer respectivement la première lettre a, la dernière lettre a et chaque lettre a du mot fourni en entrée
- $\mathsf{ins\_beg}_{a,b} : \Sigma^\star \to \Sigma^\star$, $\mathsf{ins\_end}_{a,b} : \Sigma^\star \to \Sigma^\star$ et $\mathsf{ins\_all}_{a,b} : \Sigma^\star \to \Sigma^\star$ avec $a, b \in \Sigma$. Ces opérations consistent à insérer respectivement après la première lettre a, la dernière lettre a et chaque lettre a du mot fourni en entrée une lettre b,
- ...

où $a, b \in \Sigma$.

Ces méta-opérations disposent également de déclarations de mode fonctionnelles construites comme précédemment. Les opérations d'insertion, de suppression et de substitution permettent par composition la construction de n'importe quel mot de $\Sigma^\star$ à partir d'un de ses mots. Il existe ainsi, pour n'importe quelle méta-opération avec en entrée un mot donné $x \in \Sigma^\star$ et retournant un mot $y \in \Sigma^\star$, un script d'édition optimal S tel que $S(x) = y$. Ce script est une composition des opérations d'insertion, de suppression et de substitution. Pour cette raison, nous faisons le choix d'égaler le coût d'utilisation d'une méta-opération sur un arbre à celui du script d'édition optimal permettant la même transformation pour cette entrée. Nous continuons avec la présentation des opérations élémentaires portant sur les arbres.

Pour les arbres

Rappelons qu'un arbre $t \in T_\Sigma$ est un tuple $(N_t, root_t, child_t, ns_t, lab^t_\Sigma)$ où N_t est l'ensemble des nœuds, $root_t$ un nœud appelé racine de l'arbre, $child_t$ l'ensemble des couples de nœuds définissant la relation père-fils, ns_t l'ensemble de couples de nœuds définissant la relation de fraternité et lab^t_Σ la fonction d'étiquetage des nœuds de t. Une définition complète se trouve au chapitre 3 définition 42.

Nous utilisons pour les arbres les opérations standards d'édition définies et utilisées notamment dans [Zhang 1989, Bille 2005, Demaine 2009, Pawlik 2011] :

— La *suppression* d'un nœud $n \in N_t$, différent de la racine, d'un arbre $t \in T_\Sigma$. On la note $t' = \mathsf{del}_n(t)$ où $t' \in T_\Sigma$ est l'arbre résultant de cette opération. Les fils du nœud n deviennent alors des fils du nœud parent de n et sont insérés entre les fils du nœud parent de n à la place occupée précédemment par le nœud supprimé. L'ordre des fils de n est conservé. Nous utiliserons la fonction suivante pour représenter cette opération :

$$\mathsf{del} : T_\Sigma \times N_t \to T_\Sigma$$

— L'*insertion* d'un nœud dans l'arbre t. On la note $t' = \mathsf{ins}_{a,n,i,j}(t)$ avec $t, t' \in T_\Sigma$, $a \in \Sigma$, $n \in N_t$ et $i, j \in \mathbb{N}$ où t' est l'arbre résultant de cette opération. Il s'agit de l'opération inverse de la suppression. Elle consiste à ajouter un nœud étiqueté par un symbole $a \in \Sigma$ dans un arbre $t \in T_\Sigma$. Le nœud ajouté sera un fils d'un nœud $n \in N_t$ de t et aura comme fils une sous-séquence des fils de n dont il aura pris la place. Plus précisément, le nœud ajouté prendra la place de la sous-séquence $ni,\ n(i+1),\ \cdots,\ n(j-1)$ de fils de n, avec $0 \leq i \leq j \leq k$ où k est le nombre de fils de n, devenant ainsi le $i^{\text{ième}}$ fils de n. Les fils de la séquence de nœuds ne sont alors plus des fils de n mais des fils du nœud inséré. Nous utiliserons la fonction suivante pour exprimer cette opération :

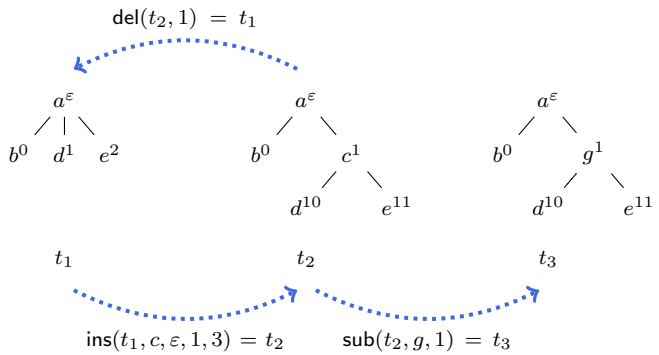
$$\mathsf{ins} : T_\Sigma \times \Sigma \times N_t \times \mathbb{N} \times \mathbb{N} \to T_\Sigma$$

— La *substitution* du label d'un nœud $n \in N_t$ d'un arbre $t \in T_\Sigma$ par un autre label $a \in \Sigma$ est notée $t' = \mathsf{sub}_{a,n}(t)$ où t' est l'arbre résultant. Cette opération ne change pas la structure de l'arbre t. Nous utiliserons la fonction suivante pour exprimer cette opération :

$$\mathsf{sub} : T_\Sigma \times \Sigma \times N_t \to T_\Sigma$$

Dans les opérations de délétion et d'insertion, il est important de noter que l'ordre établi entre les fils avant l'opération d'édition reste inchangé après la transformation. Chacune de ces opérations a une complexité linéaire en temps dans la taille de l'arbre d'entrée. Nous illustrons ces opérations élémentaires de transformation que sont l'insertion (ins), la suppression (del) et la substitution (sub) avec l'exemple suivant :

Exemple 58. *Soit trois arbres* $t_1, t_2, t_3 \in T_\Sigma$ *avec* $\Sigma = \{a, b, c, d, e, f, g\}$.



Le passage d'un arbre à un autre s'effectue par application de l'opération se trouvant sur les flèches bleues en pointillés.

Ces opérations sont standards dans la littérature et permettent, par composition, la construction à partir d'un arbre de T_Σ de n'importe quel autre arbre de T_Σ. Comme précédemment, nous pouvons associer ces opérations élémentaires à des modes de prédicat fonctionnels. Nous obtenons ainsi la liste non exhaustive des modes :

— $\mathsf{ins}(+\mathcal{T}_{T_\Sigma}, \#\mathcal{T}_\Sigma, \#\mathcal{T}_{N_t}, \#\mathcal{T}_{width(t)}, \#\mathcal{T}_{width(t)}, -\mathcal{T}_{T_\Sigma})$
— $\mathsf{del}(+\mathcal{T}_{T_\Sigma}, \#\mathcal{T}_{N_t}, -\mathcal{T}_{T_\Sigma})$
— $\mathsf{sub}(+\mathcal{T}_{T_\Sigma}, \#\mathcal{T}_{N_t}, \#\mathcal{T}_\Sigma, -\mathcal{T}_{T_\Sigma})$
— ...

avec $\mathcal{T}_{T_\Sigma}$ l'ensemble des termes fondés construit à partir des arbres de T_Σ et $\mathcal{T}_\Sigma = \{a/0 \mid a \in \Sigma\}$ l'ensemble des constantes représentant les symboles de Σ. Les deux autres types nécessitent le choix d'un arbre $t \in T_\Sigma$ allant être le premier paramètre de ces fonctions. On a ainsi $\mathcal{T}_{N_t}$ l'ensemble des constantes représentant les positions de l'arbre t et $\mathcal{T}_{width(t)} = \{i/0 \mid 0 \leq i \leq width(t)\}$ l'ensemble des constantes représentant les entiers compris entre 0 et $width(t)$ où $width(t)$ est la largeur de l'arbre t. Ces modes de prédicats peuvent être utilisés à la fois pour définir des déclarations de mode fonctionnelles portant sur l'atome de tête et les atomes du corps d'une clause.

Nous considérons que le coût des opérations d'édition que sont la substitution, la suppression et l'insertion est indépendant des entrées de ces fonctions. On les note respectivement $\mathsf{cost}_{\mathsf{sub}}$, $\mathsf{cost}_{\mathsf{del}}$ et $\mathsf{cost}_{\mathsf{ins}}$. Ces coûts sont ramenés à 1 pour nos tâches comme pour les mots. Il est possible de définir des méta-opérations basées sur l'usage des opérations élémentaires précédentes. Nous présentons quelques méta-opérations possibles :

— $\mathsf{sub\_beg}_{a,b} : T_\Sigma \to T_\Sigma$, $\mathsf{sub\_end}_{a,b} : T_\Sigma \to T_\Sigma$ et $\mathsf{sub\_all}_{a,b} : T_\Sigma \to T_\Sigma$ avec $a, b \in \Sigma$. Ces opérations consistent à renommer par un label b fgoçàprespectivement le premier nœud et le dernier nœud suivant un parcours transversal ainsi que tous les nœuds disposant d'un label a de l'arbre fourni en entrée,
— $\mathsf{del\_beg}_a : T_\Sigma \to T_\Sigma$, $\mathsf{del\_end}_a : T_\Sigma \to T_\Sigma$ et $\mathsf{del\_all}_a : T_\Sigma \to T_\Sigma$ avec $a, b \in \Sigma$. Ces opérations consistent à supprimer respectivement le premier nœud et le dernier nœud suivant un parcours transversal ainsi que chaque nœud disposant d'un label a de l'arbre fourni en entrée,
— ...

avec $a, b \in \Sigma$. Il existe, pour chaque méta-opération prenant en entrée un arbre donné $x \in T_\Sigma$ et sortant un arbre $y \in T_\Sigma$, un script optimal S tel que $S(x) = y$. Ce script est composé des opérations d'insertion, de suppression et de substitution portant sur les arbres.

Ainsi, pour une entrée donnée, nous égalons le coût d'utilisation d'une méta-opération sur un arbre à celui du script d'édition optimal assurant la même transformation.

6.1.4 Contextualisation des opérations élémentaires

Dans le cadre de notre tâche d'apprentissage de transformation, nous est fourni en entrée un ensemble d'opérations élémentaires de la forme $\delta : \mathcal{X} \to \mathcal{X}$ ainsi qu'un ensemble d'exemples. Ces exemples sont des couples d'éléments de $\mathcal{X}$ pour lesquels il existe un script d'édition optimal S tel que $\forall (x, y) \in E,\ S(x) = y$. La plupart des opérations élémentaires présentées pour les mots ainsi que pour les arbres sont paramétrables et paramétrées par des arguments de natures variées. Ceux-ci peuvent appartenir à des ensembles de tailles infinies. Or, cette présentation du problème suppose que les fonctions utilisées comme opérations ne possèdent pas de paramètre ou bien des paramètres instanciés et statiques. Il est donc nécessaire d'instancier les paramètres de ces fonctions avant utilisation. Ceci peut entraîner un manque d'expressivité pour certains problèmes et limiter les transformations apprenables. Nous illustrons cela avec l'exemple suivant :

Exemple 59. *Nous considérons dans cet exemple le problème de transformation visant à apprendre la conjugaison des verbes du* 1^{er} *groupe à la première personne du présent de l'indicatif. Nous disposons pour cela des couples de mots suivants :*

$$\begin{array}{lcl} (e_1, s_1) & = & (\ caver\ ,\ cave\ ) \\ (e_2, s_2) & = & (\ cirer\ ,\ cire\ ) \\ (e_3, s_3) & = & (\ parler\ ,\ parle\ ) \end{array}$$

que l'on transforme en atomes grâce à la fonction word-to-term *:*

$$\begin{array}{lccl} \mathsf{transfo}(& cons(c, cons(a, cons(v, cons(e, cons(r, nil))))) & , & cons(c, cons(a, cons(v, cons(e, nil)))) \quad) \\ \mathsf{transfo}(& cons(c, cons(i, cons(r, cons(e, cons(r, nil))))) & , & cons(c, cons(i, cons(r, cons(e, nil)))) \quad) \\ \mathsf{transfo}(& cons(p, cons(a, cons(r, cons(l, cons(e, cons(r, nil)))))) & , & cons(p, cons(a, cons(r, cons(l, cons(e, nil))))) \quad) \end{array}$$

ainsi que des opérations d'édition élémentaires propres aux mots que sont $\mathsf{ins}_{a,i}$, del_j *et* $\mathsf{sub}_{j,a}$. *La transformation commune à ces couples de mots consiste en la suppression de la dernière lettre du mot d'entrée. Cette transformation dépend donc de la taille des mots. Or, la position de la lettre supprimée par l'opération de délétion n'est pas instanciée dynamiquement. Il n'existe donc pas, dans ce cas, et avec les opérations d'édition spécifiées, de script d'édition représentant cette transformation.*

Afin de remédier à ce manque, nous définissons un nouveau mode de prédicat pour l'opération del_j *:*

$$\mathsf{del}(+\mathcal{T}_{\Sigma^\star}, +\mathcal{T}_{\mathbb{N}}, -\mathcal{T}_{\Sigma^\star})$$

Cette déclaration précise que le premier paramètre de la fonction del *doit être fourni par une fonction tiers. Ainsi, nous devons définir une fonction qui aura cette tâche. Nous proposons pour cela un prédicat* last/3 *dont le mode de prédicat fonctionnel est le suivant :*

$$\mathsf{last}(+\mathcal{T}_{\Sigma^\star}, \#\mathcal{T}_{\Sigma}, -\mathcal{T}_{|w|-1})$$

Le prédicat last/3 *a pour objectif de renvoyer la dernière position qu'occupe une lettre dans un mot. On a ainsi :* $\mathsf{last}(e_1, r) = \mathsf{last}(e_2, r) = 3$ *et* $\mathsf{last}(e_3, r) = 4$. *Grâce à ce symbole de prédicat, nous trouvons la définition suivante :*

$$\mathsf{transfo}(E, S) \leftarrow \mathsf{last}(E, r, P),\ \mathsf{del}(E, P, S).$$

Cette définition fonctionne pour chaque couple de mots donné en entrée de l'apprentissage.

Afin de combler le manque mis en évidence par l'exemple 59, il nous est nécessaire de définir de nouveaux modes de prédicat pour les opérations d'édition élémentaires vues précédemment. Ces modes de prédicat ont pour objectif de permettre une instanciation dynamique de certains paramètres auparavant statiques. Ces paramètres sont alors instanciés grâce aux sorties de fonctions tiers destinées à proposer des informations contextuelles sur les objets manipulés. Ces fonctions tiers nécessitent la définition de nouveaux prédicats appelés *prédicats de contexte*. L'objectif de ces prédicats est de fournir des sorties destinées à être utilisées comme paramètres des opérations de transformation présentes dans Δ.

Définition 95 (Mode de prédicat contextuel et Déclaration de mode contextuelle)**.** *Un mode de prédicat $M_{p/n}$ est* contextuel pour le type $\mathcal{T}$ par rapport à une théorie T *si :*

1. *il existe une unique entrée de $M_{p/n}$ de type $\mathcal{T}$;*
2. *les types d'entrées et de paramètres de $M_{p/n}$ sont de taille finie, à l'exception de celles de l'entrée de type $\mathcal{T}$ qui peut être infinie ;*
3. *le mode de prédicat $M_{p/n}$ est fonctionnel par rapport à T ;*

Une déclaration de mode $M_{p/n}$ est contextuelle pour le type $\mathcal{T}$ par rapport à une théorie T *si cette déclaration de mode est de la forme modeb$(1, M_{p/n})$ où $M_{p/n}$ est un mode de prédicat contextuel pour le type $\mathcal{T}$ par rapport à la théorie T.*

Un ensemble de déclarations de mode M_T est contextuel pour le type $\mathcal{T}$ par rapport à une théorie T *si chaque déclaration de mode de M_T est contextuelle pour le type $\mathcal{T}$ par rapport à la théorie T.*

La définition de mode de prédicat contextuel ne permet pour un atome qu'un nombre borné d'instanciations lorsque le terme d'entrée de type $\mathcal{T}_\mathcal{X}$ est fixé. Ainsi, seul le type $\mathcal{T}_\mathcal{X}$ est un ensemble infini de termes fondés a contrario des autres types d'entrées et de paramètres, tous de taille finie. Le nombre d'instanciations est le produit des tailles des types d'entrées et de paramètres. Dans le cas où les paramètres sont connus, seuls les types d'entrées sont considérés. Nous appelons *prédicat de contexte* un prédicat utilisé pour définir un mode de prédicat contextuel et *atome de contexte* un atome dont le symbole de prédicat est un prédicat contextuel.

Notre théorie du domaine ne contient pour le moment que les définitions de prédicats de transformation issues des opérations élémentaires définies sur $\mathcal{X}$ de Δ. Celles-ci forment l'ensemble T_Δ et sont associées à un ensemble de déclarations de mode fonctionnelles M_Δ. Nous incorporons à la théorie du domaine les définitions de prédicats de contexte définis pour le type $\mathcal{T}_\mathcal{X}$. Ces définitions forment l'ensemble T_C et sont associées à un ensemble de déclarations de mode contextuelles M_C. Nous appelons *théorie de transformation* une théorie du domaine T associée à des déclarations de modes M_T telle que T contient les théories T_Δ et T_C, c'est-à-dire $T = T_\Delta \cup T_C$, respectivement associées aux déclarations de mode M_Δ et M_C, c'est-à-dire $M_T = M_\Delta \cup M_C$. Chacune de ces théories est fonctionnelle par rapport à l'ensemble des modes M_T et ne partage aucun symbole de prédicat avec l'autre théorie afin de ne pas perturber son fonctionnement.

Les clauses qui respectent une théorie de transformation T associée à un ensemble de déclarations de mode M_T possèdent des restrictions au niveau de leurs syntaxes. En effet, la sortie d'un atome de transformation de type $\mathcal{T}_\mathcal{X}$ ne peut être utilisée que comme unique entrée du même type d'un atome de contexte ou de transformation. Les entrées d'un prédicat de transformation dont le type diffère de $\mathcal{T}_\mathcal{X}$ sont quant-à-elles obligatoirement des sorties d'atome de contexte. De cette manière, l'instanciation dynamique d'un atome de transformation nécessite la présence d'un ou plusieurs atomes de contexte aux sorties adéquates.

Nous illustrons maintenant ces différents points et proposons pour les opérations élémentaires des mots, ainsi que celles des arbres, différents modes et prédicats de contexte.

Prédicats de contexte pour les mots

Afin de définir les prédicats de contexte pour les mots, nous commençons par étoffer l'ensemble des déclarations de modes de ces derniers. Ainsi, nous définissons pour la fonction d'insertion ins les modes :

$modeh(1, \mathsf{ins}(+\mathcal{T}_{\Sigma^\star}, +\mathcal{T}_{|w|-1}, \#\mathcal{T}_\Sigma, -\mathcal{T}_{\Sigma^\star}))$
$modeh(1, \mathsf{ins}(+\mathcal{T}_{\Sigma^\star}, \#\mathcal{T}_{|w|-1}, +\mathcal{T}_\Sigma, -\mathcal{T}_{\Sigma^\star}))$
$modeh(1, \mathsf{ins}(+\mathcal{T}_{\Sigma^\star}, +\mathcal{T}_{|w|-1}, +\mathcal{T}_\Sigma, -\mathcal{T}_{\Sigma^\star}))$

pour la fonction de suppression del le mode :

$modeh(1, \mathsf{del}(+\mathcal{T}_{\Sigma^\star}, +\mathcal{T}_{|w|}, -\mathcal{T}_{\Sigma^\star}))$

et pour la fonction de substitution sub les modes :

$modeh(1, \mathsf{sub}(+\mathcal{T}_{\Sigma^\star}, +\mathcal{T}_{|w|-1}, \#\mathcal{T}_\Sigma, -\mathcal{T}_{\Sigma^\star}))$
$modeh(1, \mathsf{sub}(+\mathcal{T}_{\Sigma^\star}, \#\mathcal{T}_{|w|-1}, +\mathcal{T}_\Sigma - \mathcal{T}_{\Sigma^\star}))$
$modeh(1, \mathsf{sub}(+\mathcal{T}_{\Sigma^\star}, +\mathcal{T}_{|w|-1}, +\mathcal{T}_\Sigma, -\mathcal{T}_{\Sigma^\star}))$

avec $\mathcal{T}_{\Sigma^\star}$ l'ensemble des termes fondés construit à partir de $\Sigma^\star$, $\mathcal{T}_i = \{0/0, \cdots i/0 \mid i \in \mathbb{N}\}$ l'ensemble des constantes représentant les entiers de 1 à i et $\mathcal{T}_\Sigma = \{a/0 \mid a \in \Sigma\}$ l'ensemble des constantes représentant les symboles de Σ. Ce même travail peut être fait pour les méta-opérations.

Ces déclarations de mode nous impose les définitions de prédicat de contexte dont la sortie est de type $\mathcal{T}_{|w|-1}$ et de type $\mathcal{T}_\Sigma$. Ces définitions de prédicats dépendent bien évidemment de la tâche visée. Dans le cas de la tâche d'apprentissage visée dans l'exemple 59, nous pouvons par exemple définir les prédicats de contexte suivants :

— $firstNthLetter(w, n, l, p)$ indique la position p de la $n^{\text{ième}}$ lettre l dans le mot w et a pour modes :

$firstNthLetter(+\mathcal{T}_{\Sigma^\star}, \#\mathcal{T}_\Sigma, \#\mathcal{T}_{|w|-1}, \#\mathcal{T}_{|w|-1})$
$firstNthLetter(\#\mathcal{T}_{\Sigma^\star}, \#\mathcal{T}_\Sigma, \#\mathcal{T}_{|w|-1}, \#\mathcal{T}_{|w|-1})$
$firstNthLetter(+\mathcal{T}_{\Sigma^\star}, \#\mathcal{T}_\Sigma, \#\mathcal{T}_{|w|-1}, -\mathcal{T}_{|w|-1})$

L'opération $lastNthLetter(w, n, l, p)$ exécute la même tâche en partant de la fin du mot.

— $next(w, n, p)$ indique que dans le mot w la lettre à la position n est suivie d'une lettre à la position p et a pour modes :

$next(+\mathcal{T}_{\Sigma^\star}, \#\mathcal{T}_{|w|-1}, \#\mathcal{T}_{|w|-1})$
$next(\#\mathcal{T}_{\Sigma^\star}, \#\mathcal{T}_{|w|-1}, \#\mathcal{T}_{|w|-1})$
$next(+\mathcal{T}_{\Sigma^\star}, \#\mathcal{T}_{|w|-1}, -\mathcal{T}_{|w|-1})$

L'opération inverse de $next(w, n, p)$ est $prev(w, p, n)$.

— $posBeg(w, n, l)$ indique la lettre l à la position n dans w et a pour modes :

$posBeg(+\mathcal{T}_{\Sigma^\star}, \#\mathcal{T}_{|w|-1}, -\mathcal{T}_\Sigma)$
$posBeg(\#\mathcal{T}_{\Sigma^\star}, \#\mathcal{T}_{|w|-1}, \#\mathcal{T}_\Sigma)$
$posBeg(+\mathcal{T}_{\Sigma^\star}, \#\mathcal{T}_{|w|-1}, \#\mathcal{T}_\Sigma)$

L'opération $posEnd(w, n, l, p)$ exécute la même tâche en partant de la fin du mot.

Les programmes logiques des prédicats de contexte vont être fortement utilisés lors de l'apprentissage. Pour cette raison, il est préférable de ne considérer que des opérations de faible complexité. Remarquons que les programmes logiques présentés ont une complexité linéaire dans la taille du mot donné en entrée.

Prédicats de contexte pour les arbres

Nous proposons également pour les arbres de nouvelles déclarations de mode pour ses opérations élémentaires standards. On ainsi les déclarations de mode suivantes pour la fonction ins :

$$
\begin{array}{l}
modeh(1, \mathsf{ins}(+\mathcal{T}_{T_\Sigma}, +\mathcal{T}_\Sigma, \#\mathcal{T}_{N_t}, \#\mathcal{T}_{width(t)}, \#\mathcal{T}_{width(t)}, -\mathcal{T}_{T_\Sigma})) \\
modeh(1, \mathsf{ins}(+\mathcal{T}_{T_\Sigma}, \#\mathcal{T}_\Sigma, \#\mathcal{T}_{N_t}, \#\mathcal{T}_{width(t)}, \#\mathcal{T}_{width(t)}, -\mathcal{T}_{T_\Sigma})) \\
modeh(1, \mathsf{ins}(+\mathcal{T}_{T_\Sigma}, \#\mathcal{T}_\Sigma, \#\mathcal{T}_{N_t}, \#\mathcal{T}_{width(t)}, \#\mathcal{T}_{width(t)}, -\mathcal{T}_{T_\Sigma})) \\
\ldots \\
modeh(1, \mathsf{ins}(+\mathcal{T}_{T_\Sigma}, +\mathcal{T}_\Sigma, +\mathcal{T}_{N_t}, +\mathcal{T}_{width(t)}, \#\mathcal{T}_{width(t)}, -\mathcal{T}_{T_\Sigma})) \\
modeh(1, \mathsf{ins}(+\mathcal{T}_{T_\Sigma}, +\mathcal{T}_\Sigma, +\mathcal{T}_{N_t}, +\mathcal{T}_{width(t)}, +\mathcal{T}_{width(t)}, -\mathcal{T}_{T_\Sigma}))
\end{array}
$$

pour la fonction de suppression del le mode :

$$modeh(1, \mathsf{del}(+\mathcal{T}_{T_\Sigma}, +\mathcal{T}_{N_t}, -\mathcal{T}_{T_\Sigma}))$$

et pour la fonction de substitution sub les modes :

$$
\begin{array}{l}
modeh(1, \mathsf{sub}(+\mathcal{T}_{T_\Sigma}, +\mathcal{T}_{N_t}, \#\mathcal{T}_\Sigma, -\mathcal{T}_{T_\Sigma})) \\
modeh(1, \mathsf{sub}(+\mathcal{T}_{T_\Sigma}, \#\mathcal{T}_{N_t}, +\mathcal{T}_\Sigma, -\mathcal{T}_{T_\Sigma})) \\
modeh(1, \mathsf{sub}(+\mathcal{T}_{T_\Sigma}, +\mathcal{T}_{N_t}, +\mathcal{T}_\Sigma, -\mathcal{T}_{T_\Sigma}))
\end{array}
$$

avec $\mathcal{T}_{T_\Sigma}$ l'ensemble des termes fondés construit à partir de T_Σ, $\mathcal{T}_{width(t)} = \{i/0 \mid 0 \leq i \leq width(t)\}$ l'ensemble des constantes représentant les entiers où $width(t)$ est la largeur de l'arbre t et $\mathcal{T}_\Sigma = \{a/0 \mid a \in \Sigma\}$ l'ensemble des constantes représentant les symboles de Σ.

Ces déclarations de mode nous imposent l'usage de définitions de prédicat de contexte dont la sortie est de type $\mathcal{T}_{N_t}$, $\mathcal{T}_\Sigma$ ou $\mathcal{T}_{width(t)}$. Elles dépendent bien évidemment de la tâche visée. Dans notre cas, il s'agit de la transformation de documents XML. Nous pouvons par exemple définir les prédicats de contexte suivants :

— $subTree(t_1, n, t_2)$ spécifie que l'arbre t_1 a un sous-arbre t_2 dont la racine est le nœud n de t_1. Il a pour modes :

$$
\begin{array}{l}
subTree(\#\mathcal{T}_{T_\Sigma}, \#\mathcal{T}_{N_t}, \#\mathcal{T}_{T_\Sigma}) \\
subTree(+\mathcal{T}_{T_\Sigma}, \#\mathcal{T}_{N_t}, \#\mathcal{T}_{T_\Sigma}) \\
subTree(+\mathcal{T}_{T_\Sigma}, +\mathcal{T}_{N_t}, \#\mathcal{T}_{T_\Sigma}) \\
subTree(+\mathcal{T}_{T_\Sigma}, \#\mathcal{T}_{N_t}, -\mathcal{T}_{T_\Sigma}) \\
subTree(+\mathcal{T}_{T_\Sigma}, +\mathcal{T}_{N_t}, -\mathcal{T}_{T_\Sigma})
\end{array}
$$

— $isLeaf(t, n, b)$ indique si le nœud à la position n de l'arbre t est une feuille en prenant pour valeur de b *true* ou *false*. Il a pour modes :

$$
\begin{array}{l}
isLeaf(\#\mathcal{T}_{T_\Sigma}, \#\mathcal{T}_{N_t}, \#\mathcal{T}_\mathbb{B}) \\
isLeaf(+\mathcal{T}_{T_\Sigma}, \#\mathcal{T}_{N_t}, \#\mathcal{T}_\mathbb{B}) \\
isLeaf(+\mathcal{T}_{T_\Sigma}, -\mathcal{T}_{N_t}, \#\mathcal{T}_\mathbb{B}) \quad \text{où } \mathcal{T}_\mathbb{B} = \{true, false\}. \\
isLeaf(+\mathcal{T}_{T_\Sigma}, \#\mathcal{T}_{N_t}, +\mathcal{T}_\mathbb{B}) \\
isLeaf(+\mathcal{T}_{T_\Sigma}, -\mathcal{T}_{N_t}, -\mathcal{T}_\mathbb{B})
\end{array}
$$

Nous disposons à ce stade de représentations logiques des opérations élémentaires d'édition ainsi que des opérations de contexte. Nous ne connaissons cependant pas la difficulté de l'apprentissage d'une transformation basée sur ce genre de programmes logiques. Dans ce but, nous étudions dans la section qui suit un problème de transformation de mots basé sur l'utilisation d'opérations élémentaires simples et sans paramètre.

6.2 Transformation simple sur les mots

Les opérations élémentaires d'édition retenues dans cette section sont les opérations de renommage de chaque occurrence d'une lettre d'un mot par une autre lettre. Ces opérations d'édition sont fonctionnelles et non paramétrables. Le problème de transformation consiste alors à apprendre un script d'édition usant de ces opérations. Cette étude a pour but de montrer la taille de l'espace de recherche dans le cas d'une transformation basique. Ce dernier risque d'être plus conséquent avec l'ajout d'autres opérations élémentaires. Pour ce faire, nous commençons par quelques préliminaires sur les graphes, puis présentons un type d'homomorphismes destiné à représenter nos transformations. Enfin, nous établissons le lien entre ces transformations sous forme de scripts d'édition et ces homomorphismes. Cette dernière étape nous permettra d'analyser la taille de l'espace de recherche pour cette transformation.

6.2.1 Préliminaires

Les notions rappelées ci-dessous sont établies dans le chapitre 3 section 3.1. Elles portent sur les graphes ainsi que sur les homomorphismes.

Un *digraphe* G est un couple $G = (N, E)$ où N est un ensemble de nœuds et $E \subset N \times N$ un ensemble de couples de nœuds appelés *arêtes*. Un *cycle* C d'un digraphe $G = (N, E)$ est une séquence de nœuds $C = (x_0, \cdots, x_k)$ avec $k > 0$ telle que $x_0 = x_k$ et $\forall i \in [0..k-1]$, $(x_i, x_{i+1}) \in E$. On note $nodes(C)$ l'ensemble des nœuds d'un cycle C. Dans un digraphe $G = (N, E)$, $\forall x, y \in N$, on dit que x *atteint* y, noté $x \to y$ ou $y \leftarrow x$, si $(x, y) \in E$ ou bien si $\exists z \in V$ tel que $(x, z) \in E$ où z atteint y. On note $\to^*$ la clôture transitive de $\to$. Le nœud y est appelé un *descendant* du nœud x, noté $y \in desc(x)$, où $desc(x)$ est l'ensemble de tous les descendants de x. On appelle *feuille d'un nœud* $x \in N$ tout nœud de l'ensemble $\mathsf{leaf}(x) = \{y \in N \mid y \to^* x \wedge \nexists z \in N.\ z \to y\}$.

Un graphe G est décomposable en une union de composantes G_i telle que $G = G_1 \cup \cdots \cup G_n$ avec $n \geqslant 1$ et $\forall i, j \in [1..n]$ avec $i \neq j$, G_i et G_j ont des nœuds et arêtes disjointes. Une composante G_i est alors appelée *composante maximalement connectée*.

Un *homomorphisme* h *défini sur un alphabet* Σ est une fonction de Σ^* vers Σ^* telle que $\forall w, v \in \Sigma^*, h(wv) = h(w)h(v)$ et $h(\varepsilon) = \varepsilon$ où ε est le symbole du mot vide.

6.2.2 1-Homomorphisme et Représentation graphique

Nous introduisons ci-dessous une sous-classe d'homomorphismes : les 1-homomorphismes. Ils ont pour objectif la représentation de nos transformations simples sous forme d'homomorphismes. Plus précisement, la classe de transformations visée permet le remplacement de chaque lettre $x \in \Sigma$ d'un mot $w \in \Sigma^*$ par une autre lettre $y \in \Sigma \cup \{\varepsilon\}$ ce que permet également un 1-homomorphisme.

Définition 96 (1-Homomorphisme)**.** *Un* 1-homomorphisme *est un homomorphisme* h *tel que* $\forall x \in \Sigma.\ h(x) \in \Sigma \cup \{\varepsilon\}$.

Lemme 19. *La composée de plusieurs* 1*-homomorphismes sur* Σ *est un* 1*-homomorphisme sur* Σ*.*

Preuve. On sait que la composée d'homomorphismes est un homomorphisme. De ce fait, la composée de 1-homomorphismes est aussi un homomorphisme. Il reste donc à vérifier que cette homomorphisme est un 1-homomorphisme. Soit la composition de 1-homomorphismes suivante : $h = h_1 \circ h_2$ t.q. h_1 et h_2 sont des 1-homomorphismes ($\circ$ étant la composition au sens relationnel). On veut montrer que $\forall w \in \Sigma,\ h(w) \in \Sigma \cup \{\varepsilon\}$ et $h(\varepsilon) = \varepsilon$.

On commence par montrer que $h(\varepsilon) = \varepsilon$. On pose $w = \varepsilon$. On a alors $h(\varepsilon) = h_1 \circ h_2(\varepsilon) = h_2(h_1(\varepsilon)) = h_2(\varepsilon) = \varepsilon$. On a donc bien $h(\varepsilon) = \varepsilon$.

On montre ensuite que $\forall w \in \Sigma,\ h(w) \in \Sigma \cup \{\varepsilon\}$. On pose $w = w_1 w_2 \cdots w_k$ avec $w_{1 \leqslant i \leqslant k} \in \Sigma$. On a alors :

$$\begin{aligned} h(w) &= h(w_1 \cdots w_k) \\ &= h_1 \circ h_2(w_1 \cdots w_k) \\ &= h_2(h_1(w_1 \cdots w_k) \\ &= h_2(h_1(w_1) \cdots h_1(w_k)) \\ &= h_2(h_1(w_1)) \cdots h_2(h_1(w_k)) \end{aligned}$$

Si $h_1(w_i) = \varepsilon$ alors $h_2(h_1(w_i)) = \varepsilon$ d'où $h_1 \circ h_2(w_i) = \varepsilon$. Sinon $h_2(h_1(w_i)) \in \Sigma \cup \{\varepsilon\}$ puisque $h_1(w_i) \in \Sigma$. On a donc bien $\forall w \in \Sigma,\ h_1 \circ h_2(w) \in \Sigma \cup \{\varepsilon\}$ et $h(\varepsilon) = \varepsilon$. La composition de 1-homomorphismes est un 1-homomorphisme. □

Nous établissons maintenant une représentation graphique des homomorphismes : les graphes d'homomorphismes. Nous caractérisons ensuite les 1-homomorphismes ainsi que leurs représentations graphiques associées.

Définition 97 (Graphe d'homomorphisme)**.** *Un* graphe d'un homomorphisme h *est un graphe* $G_h = (\Sigma \cup \{\varepsilon\}, E_h)$ *où* $E_h = \{(x, y) \in (\Sigma \cup \{\varepsilon\})^2 \mid h(x) = y\}$.

Similairement à un graphe, un graphe d'homomorphisme est décomposable en une ou plusieurs composantes maximalement connectées.

Définition 98 (Graphe k-Cyclique)**.** *Un* graphe k-cyclique *est un graphe contenant au plus k cycles.*

Dans le reste de cette sous-section, nous utilisons le terme de **graphe** pour faire référence à celui de **digraphe** et d'**homomorsphisme** pour 1**-homomorphisme**.

Lemme 20. *Soit $G_i = (N_i, E_i)$ une composante d'un graphe d'homomorphisme contenant un cycle C, $\forall x \in nodes(C).\ desc(x) = nodes(C)$.*

Preuve. Soit une composante $G_i = (N_i, E_i)$ d'un graphe d'homomorphisme contenant un cycle C. $\forall x, y \in nodes(C)$, on a $x \to y$ d'où $y \in desc(x)$ et donc $desc(x) = nodes(C)$. □

Lemme 21. *Soit $G_i = (N_i, E_i)$ une composante d'un graphe d'homomorphisme contenant un cycle C, $\forall x \in N_i.\ nodes(C) \subseteq desc(x)$.*

Preuve. Soit une composante $G_i = (N_i, E_i)$ d'un graphe d'homomorphisme contenant un cycle $C = (x_0, \cdots, x_n)$. On suppose que $\exists x \in N_i$ t.q. $nodes(C) \nsubseteq desc(x)$. On sait que $nodes(C) \nsubseteq desc(x)$ donc, d'après le lemme 20, $x \notin C$.

On s'intéresse alors au cas où $x \in N_i \backslash nodes(C)$. On sait que G_i est une composante d'un graphe d'homomorphisme, de ce fait G_i est maximalement connectée. De même, on sait que si $y \in nodes(C)$ alors $y \notin desc(x)$ puisque $nodes(C) \nsubseteq desc(x)$. On a donc $\exists z \in N_i$ t.q. $z \in desc(x) \cap desc(y)$ et $y \notin desc(x)$, c'est à dire $x \to z \leftarrow y$. Or, C est un cycle appartenant à un graphe d'homomorphisme et $y \in C$, de ce fait $z \in C$. D'après le lemme 20, si $z \in nodes(C)$ alors $desc(z) = nodes(C)$. On a ainsi $desc(z) \subseteq desc(x)$ et donc $nodes(C) \subseteq desc(x)$ ce qui contredit l'hypothèse de départ. Donc $\forall x \in N_i\ .\ nodes(C) \subseteq desc(x)$.

□

Lemme 22. *Chaque composante d'un graphe d'homomorphisme est* 1*-cyclique.*

Preuve. Soit une composante G_i d'un graphe d'homomorphisme contenant deux cycles C_1 et C_2. Nous savons qu'une seule arête au plus peut sortir d'un nœud. Le cas où un nœud est commun à C_1 et C_2 implique donc systématiquement que ces cycles soient confondus, on suppose donc que $nodes(C_1) \neq nodes(C_2)$. Soit $x \in nodes(C_1)$, on a alors d'après le lemme 21 $nodes(C_2) \subseteq desc(x)$. Or, d'après le lemme 20, si $x \in nodes(C_1)$ alors $desc(x) = nodes(C_1)$ donc $nodes(C_2) \subseteq nodes(C_1)$. De même, soit $y \in nodes(C_2)$, on a alors $nodes(C_1) \subseteq desc(y)$ d'après le lemme 21. Or, d'après le lemme 20, si $y \in nodes(C_2)$ alors $desc(y) = nodes(C_2)$ donc $nodes(C_1) \subseteq nodes(C_2)$. On a ainsi $nodes(C_1) \subseteq nodes(C_2)$ et $nodes(C_2) \subseteq nodes(C_1)$ et donc $nodes(C_1) = nodes(C_2)$. Ceci contredit l'hypothèse de départ selon laquelle $nodes(C_1) \neq nodes(C_2)$. On a donc au plus un cycle par composante. □

Lemme 23. *Chaque graphe d'homomorphisme a exactement une composante $G_i = (N_i, E_i)$ telle que $\varepsilon \in N_i$ et G_i contient un cycle qui est un self-loop sur ε.*

Preuve. Soit un graphe d'homomorphisme $G = (N, E)$. On sait que $\varepsilon \in N$ et $(\varepsilon, \varepsilon) \in E$, il existe donc une composante G_i t.q. $\varepsilon \in N_i$ et $(\varepsilon, \varepsilon) \in E_i$ puisque G_i est maximalement connectée et $h(\varepsilon) = \varepsilon$. De ce fait, G_i contient un self-loop sur ε. Or, G_i est 1-cyclique donc G_i ne peut contenir d'autre cycle. Un graphe d'homomorphisme contient donc une composante G_i t.q. $\varepsilon \in N_i$ et G_i a exactement un cycle. Ce cycle est un self-loop sur ε. □

Lemme 24. *Soit un graphe $G = (N, E)$ étant un cycle, $\forall x \in N,\ \mathsf{leaf}(x) = \emptyset$.*

Preuve. Soit un graphe $G = (N, E)$ étant un cycle. Puisque G est un cycle, $\forall x \in N\ \exists y \in N.\ x \to y$ donc $\forall x \in N.\ \mathsf{leaf}(x) = \emptyset$. □

Définition 99 (Composante redeem)**.** *Une composante $G_i = (N_i, E_i)$ est* redeem *si $\exists x \in N$ tel que $\mathsf{leaf}(x) \neq \emptyset$.*

Il est facile de voir qu'une composante redeem n'est pas un cycle.

Définition 100 (Redeem homomorphisme)**.** *Un homomorphisme h est un* homomorphisme redeem *si et seulement si son graphe d'homomorphisme contient au moins une composante redeem ou bien uniquement des self-loops. On note $\mathcal{C}_{rdm}$ la classe des homomorphismes redeem.*

Nous venons de définir une représentation sous forme d'homomorphisme ainsi que de graphe pour les transformations basiques désirées. Nous proposons maintenant une représentation sous forme de script d'édition pour celles-ci.

6.2.3 Représentation d'un script simple

Afin d'établir cette relation, nous définissons en premier lieu le type d'opérations élémentaires utilisées pour nos transformations basiques et les nommerons *instructions*. Nous présentons ensuite la notion de *script de transformation*. Il s'agit d'un script d'édition basé sur l'usage d'instructions. Enfin, nous faisons le lien entre les 1-homomorphismes et les scripts de transformation.

Définition 101 (Instruction)**.** *Une* instruction *est un élément de $\Sigma \times (\Sigma \cup \{\varepsilon\})$.*

Définition 102 (Script)**.** *Un* script *est une succession d'instructions.*

Définition 103 (Transformation $T_{(x,y)}$)**.** *Une* transformation $T_{(x,y)}$ d'une instruction (x, y) *avec $x \in \Sigma$ et $y \in \Sigma \cup \{\varepsilon\}$ est un homomorphisme tel que $\forall z \in (\Sigma \cup \{\varepsilon\}) \setminus \{x\}\ T_{(x,y)}(z) = z$ et $T_{(x,y)}(x) = y$.*

Définition 104 (Transformation)**.** *Une* transformation *est une fonction de* Σ^* *vers* Σ^**.*

Définition 105 (Transformation d'un script)**.** *Une* transformation T_S d'un script $S = ((x_1, y_1), \cdots, (x_n, y_n))$ *est égale à* $T_S = T_{x_1y_1} \circ \cdots \circ T_{x_ny_n}$ *(où* $\circ$ *est la composition relationelle). On note* $\mathcal{C}_{st}$ *la classe des transformations scriptables.*

Une transformation d'un script est ainsi une transformation d'un script d'édition telle que les opérations élémentaires autorisées dans ce script d'édition sont limitées à des transformations $T_{(x,y)}$ d'instructions (x, y) et donc à des substitutions totales d'une lettre en une autre appliquées à un mot. Ce type de transformation peut se ramener à 1-homomorphisme comme nous l'indiquent les lemmes suivants.

Lemme 25. *Une transformation* $T_{(x,y)}$ *est* 1*-homomorphisme.*

Preuve. Chaque transformation $T_{(x,y)}$ avec $x \in \Sigma$ et $y \in \Sigma \cup \{\varepsilon\}$ est un 1-homomorphisme puisque $T_{(x,y)}$ est un homomorphisme tel que $\forall z \in (\Sigma \cup \{\varepsilon\}) \setminus \{x\}\ T_{(x,y)}(z) = z$ et $T_{(x,y)}(x) = y$. On a ainsi $T_{(x,y)}(z), T_{(x,y)}(y) \in \Sigma \cup \{\varepsilon\}$, et $T_{(x,y)}(\varepsilon) = \varepsilon$ ce qui vérifie bien la définition de 1-homomorphisme. □

Lemme 26. *Chaque transformation* T_S *d'un script* S *est un* 1*-homomorphisme.*

Preuve. La transformation $T_S = T_{x_1y_1} \circ \cdots \circ T_{x_n,y_n}$ d'un script $S = ((x_1, y_1), \cdots, (x_n, y_n))$ est la composition de transformations T_{x_i,y_i} avec $1 \leqslant i \leqslant n$, $x_i \in \Sigma$ et $y_i \in \Sigma \cup \{\varepsilon\}$. Or, une transformation T_{x_i,y_i} est un 1-homomorphismes et la composée de 1-homomorphismes est également un 1-homomorphisme. T_S est donc un 1-homomorphisme. □

Le lien entre les scripts de transformation et les 1-homomorphismes nous permet de représenter un script de transformation sous la forme d'un graphe d'homomorphisme. Le graphe d'une transformation scriptable est alors le graphe d'homomorphisme du 1-homomorphisme issu de ce script. Ce graphe possède la propriété suivante :

Lemme 27. *Soit* G_{T_S} *un graphe d'une transformation scriptable* T_S*, si une composante de* G_{T_S} *contient un cycle différent d'un self-loop alors* G_{T_S} *a une composante redeem.*

Preuve. Soit G_{T_S} un graphe d'une transformation scriptable T_S. On suppose que G_{T_S} ne possède pas de composante redeem et qu'il contient au moins une composante cyclique $C = (x_0, \cdots, x_k)$ différente d'un self-loop. Si le graphe G_{T_S} ne contient pas de composante redeem, alors il ne contient que des cycles. Dans ce cas, la composante contenant le symbole ε ne peut appartenir qu'à une composante self-loop. Il n'existe ainsi pas de symbole supprimé dans ce script. Or, selon le graphe de transformation, chaque symbole est transformé en un autre symbole et la transformation de ces symboles se déroule de manière séquentielle. De ce fait, s'il n'existe pas de symbole absent des symboles du cycle, alors la première transformation lettre à lettre présente dans un cycle supprime obligatoirement un symbole. On a donc une contradiction et il n'existe pas de script de transformation T_S pour le graphe de transformation G_{T_S}. Le graphe G_{T_S} contient donc une composante redeem s'il contient un cycle.

□

Nous prouvons maintenant l'équivalence de classes entre celles des transformations scriptables et celles des homomorphismes redeem. Nous démontrons ensuite qu'il est possible de tester l'égalité de deux transformations scriptables à l'aide de leurs représentations graphiques.

Théorème 25. *Une transformation est scriptable si et seulement si elle est un homomorphisme redeem.*

Preuve. On veut prouver que $\mathcal{C}_{st} = \mathcal{C}_{rdm}$.

Commençons par montrer que $\nexists T_S \in \mathcal{C}_{st}$ t.q. $T_S \notin \mathcal{C}_{rdm}$. Soit $T_S \in \mathcal{C}_{st}$. On suppose que $T_S \notin \mathcal{C}_{rdm}$. Si $T_S \notin \mathcal{C}_{rdm}$ alors G_{T_S} est un graphe ne contenant que des cycles. Or si G_{T_S} est un graphe ne contenant que des cycles alors $T_S \notin \mathcal{C}_{st}$ de par le lemme 27. Ceci contredit l'hypothèse de départ. On a donc $\mathcal{C}_{st} \subset \mathcal{C}_{rdm}$.

On veut ensuite démontrer que $\nexists T_S \in \mathcal{C}_{rdm}$ t.q. $T_S \notin \mathcal{C}_{st}$. Soit $T_S \in \mathcal{C}_{rdm}$. On suppose que $T_S \notin \mathcal{C}_{st}$. On sait que si G_{T_S} se résume à un ou plusieurs cycles alors $T_S \notin \mathcal{C}_{st}$. Or si G_{T_S} ne contient que des cycles alors $T_S \notin \mathcal{C}_{rdm}$ par la définition d'homomorphisme redeem. Ceci contredit l'hypothèse de départ. On a donc $\mathcal{C}_{rdm} \subset \mathcal{C}_{st}$.

Puisque $\mathcal{C}_{rdm} \subset \mathcal{C}_{st}$ et que $\mathcal{C}_{st} \subset \mathcal{C}_{rdm}$ alors $\mathcal{C}_{rdm} = \mathcal{C}_{st}$.

□

Théorème 26. $\forall T_1, T_2 \in \mathcal{C}_{st},\ T_1 = T_2$ *si et seulement si* $G_{T_1} = G_{T_2}$.

Preuve. On veut montrer que si $T_1 = T_2$, alors $G_{T1} = G_{T_2}$.

Soit $T_1, T_2 \in C_{st}$, on suppose que $T_1 = T_2$ alors $\forall x \in \Sigma, T_1(x) = T_2(x)$. Or $G_{T_1} = (\Sigma \cup \{\varepsilon\}, E_{T_1})$ et $G_{T_2} = (\Sigma \cup \{\varepsilon\}, E_{T_2})$ tel que $\forall x \in \Sigma, (x, T_1(x)) \in E_{T_1}$ et $(x, T_2(x)) \in E_{T_2}$. Donc $\forall x \in \Sigma, (x, T_1(x)) \in E_{T_1}$ et $(x, T_1(x)) \in E_{T_2}$
On a donc $E_1 = E_2$ et donc $G_{T_1} = G_{T_2}$.

On veut montrer que si $G_{T1} = G_{T2}$, alors $T_1 = T_2$.

Soit $G_{T_1}, G_{T_2} \in C_{dh}$, on suppose que $G_{T_1} = G_{T_2}$.
On pose $w = (w_0, \cdots, w_n) \in \Sigma^*$, on a alors $T_1(w) = T_1(w_0) \cdots T_1(w_n)$ et $T_2(w) = T_2(w_0) \cdots T_2(w_n)$. Or $G_{T_1} = G_{T_2}$ donc $\forall x \in \Sigma, T_1(x) = T_2(x)$. On a donc bien $T_1(w) = T_1(w_0) \cdots T_1(w_n) = T_2(w_0) \cdots T_2(w_n) = T_2(w)$ donc $T_1 = T_2$.

On a montré que : $\forall G_{T_1}, G_{T_2} \in C_{dh}, G_{T_1} = G_{T_2} \rightarrow T_1 = T_2$ et que $\forall T_1, T_2 \in C_{st}, T_1 = T_2 \rightarrow G_{T_1} = G_{T_2}$. Donc $\forall T_1, T_2 \in C_{st}, T_1 = T_2$ ssi $G_{T_1} = G_{T_2}$. □

Il est ainsi possible de tester l'égalité entre deux scripts de transformation grâce à leurs graphes de transformation. Nous terminons l'étude des propriétés de cette classe en spécifiant le nombre de transformations différentes et possibles pour un alphabet Σ donné.

Théorème 27. *Soit un alphabet* Σ*, il existe* $(|\Sigma| + 1)^{|\Sigma|} - |\Sigma|!$ *transformations non-équivalentes sur* Σ*, c'est-à-dire en* $|\Sigma|^{\mathcal{O}(|\Sigma|)}$.

Preuve. Soit un alphabet Σ de taille $|\Sigma|$, il existe donc $(|\Sigma| + 1)^{|\Sigma|}$ 1-homomorphismes différents.

Parmi eux, certains ne sont pas des *redeem* homomorphismes et donc des transformations. On considère un graphe d'homomorphisme où les composantes sont des cycles. Ce graphe n'est donc pas un *redeem* homomorphisme. Ce cas correspond à une permutation sans répétition sur Σ.

On aura donc parmi les $(|\Sigma| + 1)^{|\Sigma|}$ 1-homomorphismes, $|\Sigma|!$ 1-homomorphismes qui ne sont pas des *redeem* homomorphismes. On a donc $(|\Sigma|+1)^{|\Sigma|} - |\Sigma|!$ *redeem* homomorphismes sur Σ et donc le même nombre de transformations non équivalentes possibles.

On veut maintenant montrer que ce nombre de transformations est bien exponentiel. On sait que :

$$n! = n \times (n-1) \times \cdots \times (\frac{n}{2}) \times (\frac{n}{2} - 1) \times \cdots \times 1$$

$$n! \leqslant n^{\frac{n}{2}} \times (\frac{n}{2})^{\frac{n}{2}}$$

$$n! \leqslant (\frac{n}{2})^n$$

donc :

$$n^n - n! + 1 \geqslant n^n - (\frac{n}{2})^n + 1$$

$$n^n - n! + 1 \geqslant n^n(1 - \frac{1}{2^n}) + 1$$

or $\frac{1}{2^n} \underset{n \to +\infty}{\to} 0$ donc :

$$n^n - n! + 1 \geqslant n^n + 1$$

On a donc bien

$$(|\Sigma| + 1)^{|\Sigma|} - |\Sigma|! + 1 \geqslant |\Sigma|^{|\Sigma|}$$

□

L'espace de recherche constitué des opérations de substitution totale est donc exponentiellement grand par rapport à la taille des alphabets des mots regardés. Ces opérations demeurent basiques et réduisent la taille de l'espace de recherche. L'ajout de nouvelles opérations comme l'insertion d'une lettre à une position pour un mot ainsi que l'augmentation de la taille de l'alphabet risque donc d'augmenter de manière conséquente la taille de cet espace de recherche. Une énumération de ce dernier n'est donc pas envisageable afin de retrouver une transformation donnée. Nous nous posons maintenant la question de la difficulté d'un apprentissage d'une transformation pour ce dernier.

6.2.4 Apprentissage

Nous nous intéressons à l'apprentissage d'1-homomorphisme h à partir d'un ensemble d'exemples E_φ. Cet ensemble est constitué de paires de mots $(x, y) \in E_\varphi$ telles que $h(x) = y$.

Nous désirons montrer que cette tâche n'est pas triviale. Pour ce faire, nous nous intéressons au problème décisionnel suivant. Soit un ensemble d'exemples E_φ, existe-t-il un 1-homomorphisme h tel que $\forall (x, y) \in E_\varphi,\ h(x) = y$. En d'autres mots, soit un ensemble d'exemples $CONS = \{E \subseteq \Sigma^* \times \Sigma^* \mid \exists h : \Sigma \to \Sigma \cup \{\varepsilon\}\ \forall (x, y) \in E,\ h(x) = y\}$ et un ensemble d'exemples E_φ, l'ensemble E_φ est-il un sous-ensemble de $CONS$? Si la réponse est vraie alors une fonction h consistante avec E_φ est apprenable. De l'existence d'une telle fonction peut être déduit sa construction. Ce problème nous donne donc un bon indicatif de la complexité de la construction d'une telle fonction comme nous le montre l'algorithme 13. Nous illustrons avec l'exemple suivant l'existence d'ensembles de couples de mots ne figurant pas dans $CONS$. Soit l'alphabet $\Sigma = \{a, b\}$ et l'ensemble $E_\varphi = \{(aa, ab)\}$. Il n'existe pas pour cet ensemble de 1-homomorphisme h tel que $h(aa) = ab$. On a ainsi $E_\varphi \notin CONS$.

Nous sommes maintenant intéressés à prouver que notre problème décisionnel est *NP*-Complet. Cela est faisable par la réduction de $3SAT_{1in3}$ en $CONS$. Avant de le prouver, nous rappelons quelques notions de base portant sur les formules $3SAT$ et $3SAT_{1in3}$.

Notons t (respectivement f) la valeur de vérité *true* (respectivement *false*). Une formule $3CNF$ est une conjonction de clauses telle que chaque clause est une disjonction de trois littéraux où un littéral et son complémentaire ne peuvent apparaître dans la même clause. Un littéral est une variable ou sa négation. Nous notons $3CNF$ l'ensemble de toutes les formules $3CNF$. Soit une formule $3CNF$, $\varphi = (l_{11} \vee l_{12} \vee l_{13}) \wedge \cdots \wedge (l_{m1} \vee l_{m2} \vee l_{m3})$ et $\{x_1, \cdots, x_n\}$ les variables apparaissant dans φ i.e. $l_{11}, l_{12}, l_{13}, l_{21}, \cdots, l_{m1}, l_{m2}, l_{m3} \in \{x_1, \bar{x}_1, \cdots, x_n, \bar{x}_n, \}$. Une formule $3CNF$ φ est $3SAT$ si et seulement si chaque clause de φ contient au moins un littéral vrai i.e. $\exists v : \{l_{11}, \cdots, l_{m3}\} \to \{t, f\}$ t.q. $\forall i \in [1..m]\ \exists j \in [1..3]\ v(l_{ij}) = t$. Une formule $3CNF$ φ est $3SAT_{1in3}$ si et seulement si chaque clause de φ contient exactement un littéral vrai i.e. $\exists v : \{l_{11}, \cdots, l_{m3}\} \to \{t, f\}$ s.t. $\forall i \in [1..m]\ \exists! j \in [1..3]\ v(l_{ij}) = t \wedge \forall k \in [1..m] \setminus \{j\}\ v(l_{ik}) = f$.

Le problème d'appartenance d'une formule à l'ensemble $3SAT$ est NP-Complet [Karp 1972a]. Il en est de même pour le problème d'appartenance d'une formule à l'ensemble $3SAT_{1in3}$ [Schaefer 1978]. Nous présentons notre réduction de $3SAT_{1in3}$ à $CONS$ nécessaire à la démonstration du lemme suivant :

Lemme 28. $\forall \varphi \in 3CNF.\ E_\varphi \in CONS \iff \varphi \in 3SAT_{1in3}$.

Soit une formule $3CNF$ $\varphi = (l_{11} \vee l_{12} \vee l_{13}) \wedge \cdots \wedge (l_{m1} \vee l_{m2} \vee l_{m3})$ tel que $l_{11}, l_{12}, l_{13}, l_{21}, \cdots, l_{m1}, l_{m2}, l_{m3} \in \{x_1, \bar{x}_1, \cdots, x_n, \bar{x}_n, \}$. Soit un alphabet $\Sigma = \{x_1, \bar{x}_1, \cdots, x_n, \bar{x}_n, t\}$, nous pouvons construire à partir de la formule φ un ensemble de paires de mots $E_\varphi = \{(x_1\bar{x}_1, t), \cdots, (x_n\bar{x}_n, t), (l_{11}l_{12}l_{13}, t), \cdots, (l_{m1}l_{m2}l_{m3}, t)\}$. L'alphabet Σ contient toutes les variables apparaissant dans φ ainsi que leurs négations.

Nous illustrons cette construction avec l'exemple suivant :
Soit la formule $\varphi = (x_1 \vee \bar{x}_2 \vee \bar{x}_3) \wedge (\bar{x}_2 \vee \bar{x}_3 \vee x_4) \in 3SAT_{1in3}$ et v la fonction d'assignation aux valeurs de vérité telle que $v(x_1) = v(x_4) = t$ et $v(x_2) = v(x_3) = f$. À partir de la formule φ, nous pouvons construire l'ensemble des exemples :

$$E_\varphi = \{(x_1\bar{x}_1, t), \cdots, (x_4\bar{x}_4, t), (x_1\bar{x}_2\bar{x}_3, t), (\bar{x}_2, \bar{x}_3, x_4, t)\}$$

tel que $h(x_1) = h(x_4) = h(\bar{x}_2) = h(\bar{x}_3) = t$ et $h(\bar{x}_1) = h(\bar{x}_4) = h(x_2) = h(x_3) = \varepsilon$.

Preuve. Soit $\varphi = (l_{11} \vee l_{12} \vee l_{13}) \wedge \cdots \wedge (l_{m1} \vee l_{m2} \vee l_{m3}) \in 3CNF$ telle que $l_{11}, \cdots, lm3 \in \{x_1, \bar{x}_1, \cdots, x_n, \bar{x}_n\}$ et $E_\varphi = \{(x_1\bar{x}_1, t), \cdots, (x_n\bar{x}_n, t), (l_{11}l_{12}l_{13}, t), \cdots, (l_{m1}l_{m2}l_{m3}, t)\}$.

($\Leftarrow$) Nous commençons par prouver que si $E_\varphi \in CONS$, alors $\varphi \in 3SAT_{1in3}$.

On suppose que $E_\varphi \in CONS$. Il existe donc une fonction h décrite comme ci-dessus. À partir des exemples $(x_1\bar{x}_1, t), \cdots, (x_n\bar{x}_n) \in E_\varphi$, nous construisons la fonction d'évaluation v telle que :
$v(x_i) = \begin{cases} t & \text{if} \quad h(x_i) = t \\ f & \quad\ \ h(\bar{x}_i) = t \end{cases}$ car $h(x_i\bar{x}_i) = h(x_i).h(\bar{x}_i) = t$ donc $h(x_i) = t \iff h(\bar{x}_i) = \varepsilon$.
Soit $C_j = l_{j1} \vee l_{j2} \vee l_{j3}$, nous savons que $h(l_{j1}l_{j2}l_{j3}) = t = h(l_{j1}).h(l_{j2}).h(l_{j3})$. Sans perte de généralité, on a $h(l_{j1}) = t$ et $h(l_{j2}) = h(l_{j3}) = \varepsilon$. Ainsi, $v(l_{j1}) = t$ et $v(l_{j2}) = v(l_{j3}) = f$. On a donc $\varphi \in 3SAT_{1in3}$.

($\Rightarrow$) Nous prouvons que si $\varphi \in 3SAT_{1in3}$, alors $E_\varphi \in CONS$.

On suppose que $\varphi \in 3SAT_{1in3}$. Il existe donc une fonction d'évaluation v. Nous construisons h tel que :
$h(x_i) = \begin{cases} t & \text{si} \quad v(x_i) = t \\ \varepsilon & \quad\ \ v(x_i) = f \end{cases}$ et $h(\bar{x}_i) = \begin{cases} t & \text{if} \quad v(x_i) = f \\ \varepsilon & \quad\ \ v(x_i) = t \end{cases}$
On sait que $h(x_i\bar{x}_i) = h(x_i).h(\bar{x}_i)$ donc :

1. si $v(x_i) = t$, alors $h(x_i\bar{x}_i) = t.\varepsilon = t$;
2. ou si $v(x_i) = f$, alors $h(x_i\bar{x}_i) = \varepsilon.t = t$.

On a donc $h(x_i\bar{x}_i) = h(x_i).h(\bar{x}_i) = t$.

Nous devons maintenant prouver que $h(l_{j1}l_{j2}l_{j3}) = t$.

On sait que $h(l_{j1}l_{j2}l_{j3}) = h(l_{j1}).h(l_{j2}).h(l_{j3})$ et $\forall j \in [1..m]$ $C_j = l_{j1} \vee l_{j2} \vee l_{j3}$ tel que (et sans perte de généralité) $v(l_{j1}) = t$ et $v(l_{j2}) = v(l_{j3}) = f$. On a donc $h(l_{j1}) = t$ et $h(l_{j2}) = h(l_{j3}) = \varepsilon$. Ainsi, $h(l_{j1}l_{j2}l_{j3}) = t$ et on a donc $E_\varphi \in CONS$.

□

À l'aide du lemme 28, nous explicitons la dureté du problème $CONS$.

Théorème 28. *$CONS$ est NP-Complet.*

Preuve. Soit un ensemble d'exemples $E_\varphi \subseteq CONS$, il est facile de voir que $\forall (x, y) \in E_\varphi,\ h(x) = y$ peut être testé en temps polynomial. De plus, $\forall \varphi \in 3CNF.\ E_\varphi \in CONS \iff \varphi \in 3SAT_{1in3}$ (voir lemme 28). Donc *CONS* est *NP*-Complet.

□

Cette preuve démontre que *CONS* est un problème NP-Complet. Cependant, elle repose sur l'usage d'un alphabet de taille non fixée. Or, il est fréquent dans le domaine de l'inférence grammaticale de considérer la taille de l'alphabet comme constante. Ainsi, nous savons que l'espace de recherche représenté par l'ensemble des opérations élémentaires autorisées dans un problème de transformation pourra être très grand. Nous nous attaquons dans la section suivante à ce problème d'apprentissage.

6.3 Apprentissage d'une transformation

Nous nous intéressons dans cette section à l'apprentissage d'une transformation commune à un ensemble d'exemples $E_\mathcal{X}$. Rappelons que chaque exemple de $E_\mathcal{X}$ décrit la transformation d'un objet, appartenant à un ensemble d'objets $\mathcal{X}$, vers un autre objet de ce même ensemble. On a ainsi $E_\mathcal{X} \subseteq \mathcal{X}^2$. Le choix de représenter une transformation d'un élément d'un ensemble $\mathcal{X}$ vers un autre élément de ce même ensemble par un script d'édition a été effectué au début de ce chapitre. Un script d'édition est une séquence d'opérations élémentaires définies sur $\mathcal{X}$. Ces opérations élémentaires se présentent sous la forme de fonctions de $\mathcal{X}$ vers $\mathcal{X}$. Elles forment un ensemble Δ d'opérations élémentaires définies sur $\mathcal{X}$. Celui-ci est donné en entrée de notre problème de transformation. La tâche d'apprentissage consiste alors à inférer à partir de ces exemples une fonction $\mathsf{transfo} : \mathcal{X} \to \mathcal{X}$, appelée *fonction de transformation*, telle que $\forall (x, y) \in E_\mathcal{X}, \mathsf{transfo}(x) = y$.

Dans le cadre de la PLI et comme cela a été expliqué au début de ce chapitre, il nous a été nécessaire d'adapter ces différents éléments à la PLI. Nous travaillons ainsi sur un ensemble d'exemples E obtenu à partir de l'ensemble $E_\mathcal{X}$ par la transformation de chaque couple $(e, s) \in E_\mathcal{X}$ en un atome :

$$\mathsf{transfo}(\mathsf{object\text{-}to\text{-}term}(e_i), \mathsf{object\text{-}to\text{-}term}(s_i))$$

Rappelons que la fonction object-to-term assure la transformation sans perte d'un élément de $\mathcal{X}$ en un terme. L'ensemble E issu de $E_\mathcal{X}$ est ainsi égal à :

$$\{\mathsf{transfo}(\mathsf{object\text{-}to\text{-}term}(e_1), \mathsf{object\text{-}to\text{-}term}(s_1)),\ \cdots,\ \mathsf{transfo}(\mathsf{object\text{-}to\text{-}term}(e_n), \mathsf{object\text{-}to\text{-}term}(s_n))\}$$

L'objectif de l'apprentissage est alors d'apprendre une définition de la fonction transfo/2 et donc un programme logique. Celui-ci représentera un script d'édition.

À la manière des exemples, les opérations d'édition élémentaires de l'ensemble Δ ont été adaptées. Pour cela, nous avons introduit la notion de déclaration de mode fonctionnelle. Celle-ci permet la représentation d'une fonction sous la forme d'une définition d'un symbole de prédicat. Elle régit également l'instanciation des entrées, paramètres et sorties ainsi que les types d'arguments d'un atome basé sur le prédicat d'une de ces définitions.

Afin d'élargir l'expressivité des opérations élémentaires, le choix de remplacer dans les déclarations de mode initiales certains paramètres par des entrées en conservant le type a été fait. Ces déclarations de mode ne proposent comme sortie que des éléments du type $\mathcal{T}_\mathcal{X}$. Afin de respecter ces déclarations de mode, il a été nécessaire d'en proposer d'autres ainsi que des définitions de prédicat offrant des sorties de même type que celui de ces nouvelles entrées. La notion de déclaration de mode contextuelle ainsi que celle de prédicat de contexte ont alors été introduites. Elles sont utiles à la définition de fonctions tiers dont

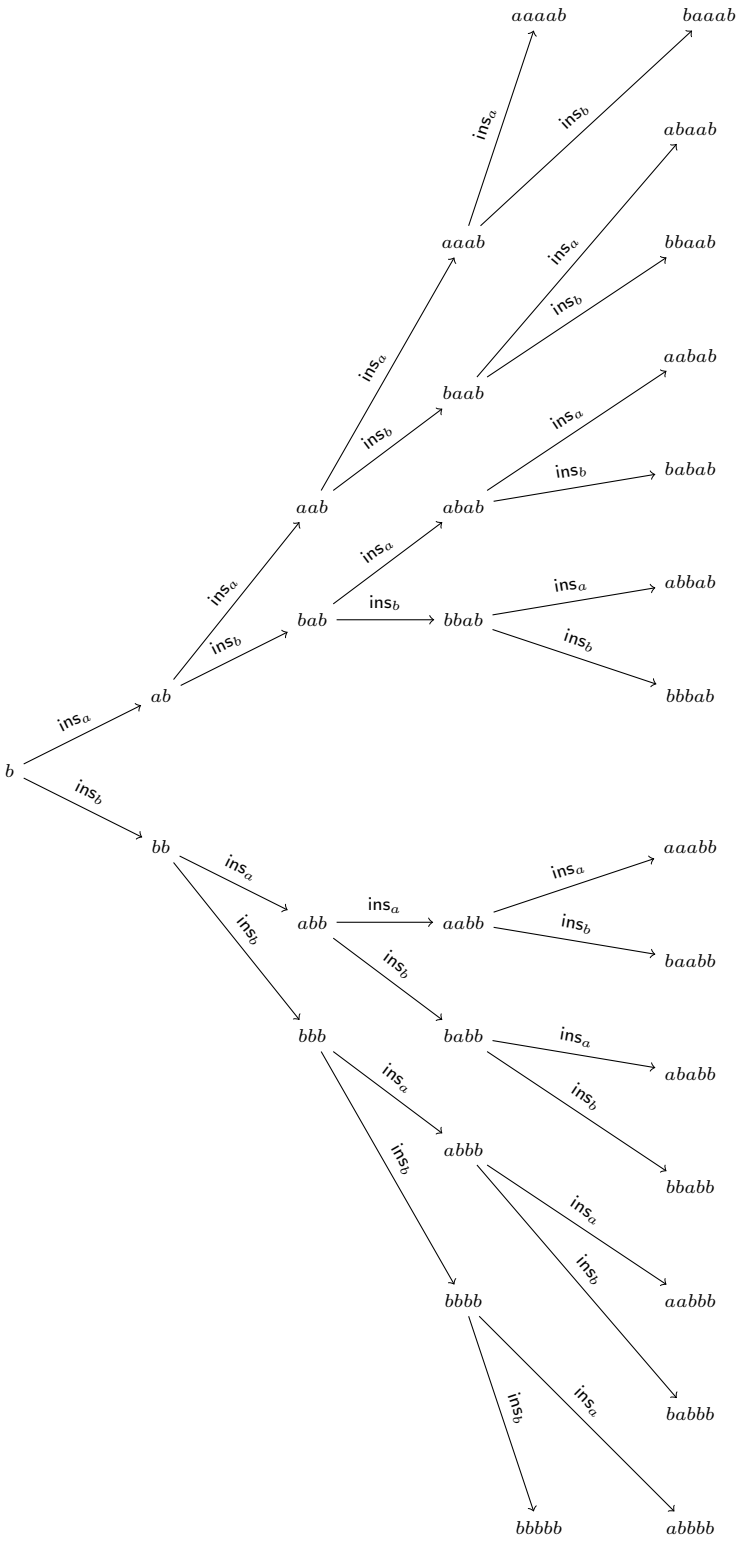


FIGURE 6.1 – Ensemble de transformations distinctes de taille inférieure à 5 partant d'un mot b construit à partir d'un alphabet $\Sigma = \{a, b\}$ et n'utilisant que l'opération d'insertion $\mathsf{ins\_beg} : \Sigma^\star \times \Sigma \to \Sigma^\star$.

l'intérêt est d'apporter des informations sur les éléments manipulés. Elles proposent des sorties compatibles avec les entrées des déclarations de mode des opérations d'édition et permettent un paramétrage dynamique.

Les différents points présentés restreignent la forme des programmes logiques apprenables. Cependant, nous avons vu au cours de la section précédente que l'espace de recherche pour des opérations élémentaires simples sur des mots, c'est-à-dire sans paramètre, peut être de taille exponentielle dans la taille de l'alphabet de ces mots. Ainsi, l'énumération des opérations élémentaires n'est pas appropriée. Ce problème s'intensifie avec l'ajout d'opérations paramétrables comme, par exemple, l'insertion d'une lettre dans un mot. En effet, cette opération complexifie l'espace de recherche comme dans la figure 6.1.

Une partie de l'espace de recherche pour le cas particulier de l'insertion en début de mot d'une lettre d'un alphabet fixé y est représentée. Dans ce cas, l'espace de recherche devient infini et le nombre de transformations distinctes utilisant n opérations élémentaires pour un alphabet Σ est $\mathcal{O}(|\Sigma|^n)$. Ainsi, la construction totale de l'espace de recherche ou partielle de toutes les transformations de taille bornée par un entier n n'est également pas adaptée.

Pour cette raison, nous optons pour la construction d'un unique script de transformation. La construction de cette transformation sera guidée par les exemples et se traduira par une recherche dans l'espace de recherche. L'apprentissage d'une transformation se décompose alors en plusieurs étapes :

1. la construction d'une clause à partir d'un atome $\mathsf{transfo}(\mathsf{object\text{-}to\text{-}term}(e_i), \mathsf{object\text{-}to\text{-}term}(s_i))$ issu d'un couple d'éléments entrée/sortie (e_i, s_i). Cette clause doit représenter la transformation de ce couple ;
2. la généralisation de plusieurs clauses précédemment apprises ;
3. l'élagage de la généralisation obtenue ;
4. la suppression des exemples de E couverts par la clause apprise ;

Ces étapes sont répétées jusqu'à épuisement de l'ensemble d'exemples. Nous obtenons ainsi l'algorithme 6.3.

La fonction enhance de cet algorithme assure la construction du corps d'une clause dont l'atome de tête est de la forme :

$$h = \mathsf{transfo}(\mathsf{object\text{-}to\text{-}term}(e_i), \mathsf{object\text{-}to\text{-}term}(s_i))$$

Une clause construite par la fonction enhance devra à la fois contenir des atomes de transformation ainsi que des atomes de contexte. La généralisation des clauses construites par la fonction enhance est réalisée par la fonction generalize. Elle consiste à généraliser les contextes entre eux, puis les atomes de transformation. La fonction prune réalise ensuite la suppression des littéraux dispensables à la transformation. Enfin, lorsqu'une hypothèse a été apprise, on supprime de E les exemple couverts par celle-ci. Cette suppression utilise une règle d'inférence $\vdash$. Cette règle sera utilisée à plusieurs reprises au cours de cette section. Nous favorisons pour notre tâche la SLD-résolution. Ce choix est justifié par l'article de Santos et Muggleton [Santos 2011]. Ils ont montré que la SLD-résolution de Prolog [Kowalski 1971] est la règle d'inférence la plus efficace dans le cas d'une base de données déterministe.

Nous donnons maintenant plus de détails sur ces différentes fonctions et précisons la forme des clauses représentant une transformation.

function learn(E, T, M_T)
1: **let** $H := \emptyset$;
2: **while** $E \neq \emptyset$ **do**
3: $E' := E \setminus \{e_i\}$;
4: $h :=$ enhance(e_i, T, M_T) ;
5: **if** $h = null$ **then return** no transformation found ;
6: **while** $E' \neq \emptyset$ **do**
7: $E' := E' \setminus \{e_j\}$;
8: $e :=$ enhance(e_j, T, M_T) ;
9: **if** $e = null$ **then return** no transformation found ;
10: $h' :=$ generalize(h, e, M_T) ;
11: **if** $h' \neq null$ **then**
12: $h := h'$;
13: **end while**
14: $h :=$ prune(h, E, T, M_T) ;
15: **for each** $e \in E$ **do**
16: **if** $h \wedge T \vdash e$ **then** $E := E \setminus \{e\}$;
17: **end for**
18: $H := H \cup \{h\}$;
19: **end while**
20: **return** H ;
end function

6.3.1 Clause de transformation

Rappelons que l'utilisation de déclarations de mode a pour principal objet la construction de clauses respectant ces modes. Nous considérons que nous disposons de déclarations de mode M_T associées à une théorie de transformation T. Les clauses hypothèses apprises par notre algorithme se doivent de respecter ces modes. Elles appartiennent ainsi à $\mathcal{L}^{fun}_{M_T}$. Par notre choix de ne représenter qu'un seul script d'édition par hypothèse apprise, nous pouvons contraindre davantage la forme des clauses manipulées. Pour cette raison, nous définissons la notion de *clause de transformation*.

Définition 106 (Clause de transformation)**.** *Soit les entrées d'un problème de transformation donné que sont un ensemble de déclarations de mode $M_T = M_\Delta \cup M_C$ associées à une théorie de transformation $T = T_\Delta \cup T_C$ défini pour le type $\mathcal{T}_\mathcal{X}$ et un ensemble d'exemples $E = \{\mathsf{transfo}(e_1, s_1), \cdots, \mathsf{transfo}(e_n, s_n)\}$ issu d'un ensemble d'exemples $E_\mathcal{X}$. Une* clause de transformation *de ce problème est une clause de $\mathcal{L}^{fun}_{M_T}$ de la forme :*

$$\mathsf{transfo}(t_0, t_n) \leftarrow C_{t_0}, \ell^\delta_{t_0}, \cdots, C_{t_j}, \ell^\delta_{t_j}, \cdots, C_{t_{n-1}}, \ell^\delta_{t_{n-1}}.$$

où :
- *$C_{t_j} = \ell^0_{t_j}, \cdots, \ell^{i_j}_{t_j}$, avec $0 \leqslant j < n$ et $0, \cdots, i_j$ des entiers consécutifs, est une séquence d'atomes de contexte dont le paramètre d'entrée de type $\mathcal{T}_\mathcal{X}$ est le terme t_j ;*
- *$\ell^\delta_{t_j}$ avec $0 \leqslant j < n$ est l'atome de transformation dont le paramètre d'entrée de type $\mathcal{T}_\mathcal{X}$ est le terme t_j et dont les atomes de contexte sont les atomes de $C_{t_0}, \cdots, C_{t_j}$;*
- *$\ell^\delta_{t_{n-1}}$ est l'atome de transformation dont le paramètre de sortie de type $\mathcal{T}_\mathcal{X}$ est le terme t_n.*

Une clause de transformation fondée ne possède pas obligatoirement d'atome de contexte. Dans ce cas, seuls les termes de type $\mathcal{T}_\mathcal{X}$ peuvent être des variables. L'ordre imposé par la

notion de respect des déclarations de mode a pour objectif de limiter l'origine des paramètres utiles à l'instanciation d'un atome de transformation. Le respect des déclarations de mode fonctionnelles et contextuelles contraint un peu plus la forme d'une clause et oblige les paramètres d'un atome de transformation à être issus d'atomes de contexte. Ainsi, dans une clause de transformation, les atomes de contexte nécessaires à un atome de transformation se situent avant lui. De cette manière, une opération de transformation paramétrée dynamiquement et appliquée à un objet ne dépend que des propriétés déduites de cet objet et des précédents.

Une clause de transformation représente un script de transformation dont les opérations d'édition peuvent être paramétrées par un contexte. Ainsi, pour un entier $0 \leqslant j < n$, l'ensemble des atomes $\ell^0_{t_0}, \cdots, \ell^{i_0}_{t_0}, \cdots, \ell^0_{t_j}, \cdots, \ell^{i_j}_{t_j}, \ell^{\delta}_{t_j}$ correspond à une opération d'édition dont l'entrée de type $\mathcal{T}_{\mathcal{X}}$ est t_j et les autres entrées sont des sorties des atomes de contexte issus des termes $t_0, \cdots, t_j$. L'instanciation des variables d'un atome de transformation par des atomes de contexte est déterministe en fonction du terme d'entrée de type $\mathcal{T}_{\mathcal{X}}$. En effet, les déclarations de mode des atomes de contexte sont fonctionnelles. Ainsi, leur respect impose que leurs entrées soient instanciées ce qui fixe les variables de sortie. Nous disposons ainsi d'un ensemble d'opérations d'édition paramétrées définies sur $\mathcal{T}_{\mathcal{X}}$.

La forme des clauses de transformation fixée, nous pouvons maintenant nous intéresser à leur enrichissement. Cette tâche consiste pour notre problème en l'ajout de littéraux de transformation et/ou de contexte dans le corps de la clause afin de construire une clause de transformation.

6.3.2 Recherche du contexte

Le principe d'enrichissement d'une clause consiste à ajouter à une clause des littéraux déduits d'une théorie du domaine ainsi que de cette clause. Plusieurs méthodes visent à réaliser cette tâche et ont été proposées, notamment la *saturation* de Rouveirol [Rouveirol 1992, Rouveirol 1994]. Saturer une clause consiste à lui ajouter tous les littéraux impliqués par une théorie donnée afin de construire la clause la plus spécifique possible appelée *bottom clause*. La *clause la plus spécifique (ou bottom clause) d'une clause C par rapport à une théorie T* est une clause $\bot(C)$, la plus spécifique, telle que $\bot(C) \cup T \models C$. Nous définissons maintenant de manière formelle l'opération de saturation :

Définition 107 (Saturation élémentaire et Saturation)**.** *Soit une clause sans constante, ni terme complexe $C_e = \{h_e, \neg b^e_1, \cdots, \neg b^e_n\}$, appelée clause exemple, et une clause $C_s = \{h_s, \neg b^s_1, \cdots, \neg b^s_n\}$ également sans constante ni terme complexe, appelée clause saturante, telles qu'il existe une substitution θ pour laquelle $\{\neg b^s_1, \cdots, \neg b^s_n\}\theta \subseteq \{\neg b^e_1, \cdots, \neg b^e_n\}$ (c'est-à-dire θ-subsume). On appelle* saturation élémentaire de C_e par C_s *la clause :*

$$C_h = \{h_e, \neg b^e_1, \cdots, \neg b^e_n\} \cup \{\neg h_s\theta\}$$

Soit un programme défini T et une clause définie $C_e = \{h_e, \neg b^e_1, \cdots, \neg b^e_n\}$ tous les deux sans constante, ni terme complexe, la saturation de C par T, notée $\mathsf{saturation}^*(C,T)$*, est la clôture transitive de C par toutes les clauses de T à l'aide de la saturation élémentaire. Elle est définie récursivement comme suit :*

1. $\mathsf{saturation}^0(C,T) = \{C\}$,
2. $\mathsf{saturation}^{n>0}(C,T) = \mathsf{saturation}^{n-1}(C,T) \cup \{h\theta \mid h\theta$ *est obtenue par saturation élémentaire de* $\{h, \neg b_1, \cdots, \neg b_m\} \in T$ *avec* $\mathsf{saturation}^{n-1}(C,T)\}$
3. $\mathsf{saturation}^\star(C,T) = \mathsf{saturation}^0(C,T) \cup \mathsf{saturation}^1(C,T) \cup \mathsf{saturation}^2(C,T) \cup \cdots$

La transformation d'une clause définie en clause définie sans constante ni terme complexe est réalisable grâce à l'aplatissement. La clause obtenue par saturation peut ensuite être désaplatie. Plus de détails sur ces points se trouvent dans le chapitre 2 section 2.5 ainsi que dans [Rouveirol 1992, Rouveirol 1994]. Nous illustrons le principe de saturation avec l'exemple suivant :

Exemple 60. *Soit la théorie T suivante :*

$$\begin{array}{lcl} list(nil). & & \\ list(cons(X,Y)) & \leftarrow & list(Y). \\ member(X,cons(X,Y)) & \leftarrow & list(Y). \end{array}$$

et la clause exemple C :

$$member(4,cons(3,cons(4,nil))).$$

La bottom clause de $\bot(C)$ est :

$$\begin{array}{lcl} member(4,cons(3,cons(4,nil))) & \leftarrow & member(4,cons(3,cons(4,nil))),\ list(cons(4,nil)), \\ & & member(3,cons(3,cons(4,nil))), \\ & & list(cons(3,cons(4,nil))). \end{array}$$

Une clause saturée par une théorie T est ainsi enrichie de littéraux déduits de la clause et de la théorie. Celle-ci peut boucler à l'infini dans le cas où une clause de la théorie du domaine n'est pas *range-restricted*, c'est-à-dire qu'une variable de la tête n'est pas présente dans son corps.

Les principes de saturation et de bottom clause sont repris dans de nombreux systèmes de PLI comme Progol [Muggleton 1995], ProGolem [Muggleton 2010], Beth [Tang 2003] ou ALEPH [Srinivasan 2004]. Tout littéral présent dans une clause la plus spécifique n'est pas toujours pertinent pour le problème étudié. Des modifications sur la saturation ont été apportées afin de contraindre la forme d'une clause saturée et ainsi limiter la présence de littéraux non pertinents. Parmi ces contraintes se trouvent l'obligation qu'à une clause saturée de respecter un ensemble de déclarations de mode [Muggleton 1995, Duboc 2009] et celle de n'utiliser que certains symboles de prédicat appelés *prédicat de détermination* [Srinivasan 2004].

Au vu de la forme des clauses que nous considérons, la saturation d'un de nos exemples générerait de nombreux littéraux non pertinents pour la construction d'une clause de transformation. Ainsi, nous optons pour un enrichissement simplifié du contexte. Celui-ci est présenté à l'algorithme 16 . Il prend en entrée un terme fondé t de type $\mathcal{T}_\mathcal{X}$ et un ensemble de déclarations de mode contextuelles M pour le type $\mathcal{T}_\mathcal{X}$ associé à une théorie T. Il construit alors l'ensemble des instanciations fondées d'atomes dont :

— le symbole de prédicat est présent dans une déclaration de mode de M,
— l'entrée de type $\mathcal{T}_\mathcal{X}$ est t
— et les autres arguments sont des variables.

Le nombre d'atomes de l'ensemble construit est le résultat de la somme de toutes les instances fondées d'atomes respectant une déclaration de mode de M pour une entrée de type $\mathcal{T}_\mathcal{X}$ donnée. Il est de taille finie mais peut être exponentiel en fonction de l'arité des prédicats. Il faut donc utiliser des prédicats de faible arité comme généralement en PLI.

Algorithme 16 Construction de l'ensemble des atomes respectant un ensemble de déclarations de mode M associé à une théorie T et ayant comme entrée un terme t.

function $\mathsf{enhance}_{\mathsf{atom}}(t, T, M)$
1: **let** $S := \emptyset$;
2: **for each** mode declaration $M_{p/n}$ of M **do**
3: **for each** atom $A = p(t_1, \cdots, t_n)$ s.t. A is constructed from $M_{p/n}$ by replacing term of type $\mathcal{T}_{\mathcal{X}}$ with t, parameters and inputs with ground terms from the corresponding types and the output by a variable **do**
4: $S := S \cup \{A\theta \mid A \wedge T_C \vdash A\theta$ and θ is ground$\}$;
5: **end for**
6: **end for**
7: **return** S ;
end function

Pour générer le contexte, nous utilisons cet algorithme avec, comme déclarations de mode, les déclarations de mode contextuelles M_C et, comme théorie, celle propre à M_C c'est-à-dire T_C. Les prédicats de contexte utilisés pour la construction de ces atomes fondés sont assimilables à des prédicats de détermination. L'exemple 61 illustre l'application de cet algorithme.

Exemple 61. *Soit le mot "gérer" représenté par le terme :*

$$cons(g, cons(é, cons(r, cons(e, cons(r, nil)))))$$

et l'opération de contexte $p = firstNthLetter(w, n, l)$ *qui renvoie la position* p *de la* $n^{ième}$ *lettre* l *dans le mot* w*. Nous utilisons les déclarations de mode suivantes :*

$$firstNthLetter(+\mathcal{T}_{\Sigma^\star}, +\mathcal{T}_{\Sigma}, +\mathcal{T}_{|w|-1}, -\mathcal{T}_{|w|-1})$$
$$firstNthLetter(\#\mathcal{T}_{\Sigma^\star}, \#\mathcal{T}_{\Sigma}, \#\mathcal{T}_{|w|-1}, \#\mathcal{T}_{|w|-1})$$

et obtenons les atomes de contexte suivants :

firstNthLetter(cons(g,cons(é,cons(r,cons(e,cons(r,nil))))),e,1,3)
firstNthLetter(cons(g,cons(é,cons(r,cons(e,cons(r,nil))))),é,1,1)
firstNthLetter(cons(g,cons(é,cons(r,cons(e,cons(r,nil))))),g,1,0)
firstNthLetter(cons(g,cons(é,cons(r,cons(e,cons(r,nil))))),r,1,2)
firstNthLetter(cons(g,cons(é,cons(r,cons(e,cons(r,nil))))),r,2,4)

6.3.3 Recherche des atomes de transformation guidée par un exemple

Nous avons vu, lors de la sous-section 6.1.3, la possibilité de représenter l'ensemble des scripts d'édition sous la forme d'un graphe de transformation. Nous exploitons cette représentation afin de trouver un chemin de transformation liant deux éléments de l'ensemble $\mathcal{X}$. Rappelons que ce chemin correspond alors à un script d'édition. Ainsi, la construction des atomes de transformation sera déduite de ce chemin. Pour ce faire, notre approche se base sur l'utilisation de l'algorithme A^* et d'heuristiques, les distances d'édition, afin de permettre une recherche efficace dans ce graphe. Nous présentons ainsi l'algorithme A^* de recherche d'un plus court chemin dans un graphe et étudions l'utilisation de distances d'édition comme heuristique de cet algorithme.

Le problème du plus court chemin entre un nœud source s et une nœud cible t d'un graphe consiste à trouver un plus court chemin partant de s et allant à t. La taille d'un chemin est ici le nombre d'arêtes qui le composent. Cette définition ne prend pas en compte

les étiquettes des arêtes que peut contenir un graphe de transformation ainsi que le coût induit par ces étiquettes. En effet, chaque triplet (x, δ, y) d'un chemin de transformation est associable au coût $\mathsf{cost}_\delta(x)$ de l'application de la transformation élémentaire δ à l'élément x afin d'obtenir l'élément y. Il en est de même pour un chemin de transformation dont le coût sera celui de son script d'édition. Ainsi, nous définissons dans le cas d'un graphe de transformation G, un plus court chemin partant d'un nœud x_1 et allant au nœud x_k de G comme un chemin de transformation $p = (x_1, \delta_{i_1}, x_2), \cdots, (x_{k-1}, \delta_{i_{k-1}}, x_k)$ partant de x_1 jusqu'à x_k dont le script d'édition $S_p = (\delta_{i_1}, \cdots, \delta_{i_{k-1}})$ est optimal de x_1 à x_k. Le problème du plus court chemin entre un nœud source s et une nœud cible t dans un graphe de transformation G consiste alors à trouver un plus court chemin de transformation partant de s jusqu'à t dans G.

L'algorithme A^*

Afin de résoudre ce problème du plus court chemin, nous allons utiliser l'algorithme A^* [Hart 1968, Koenig 2004]. Il s'agit d'un algorithme de recherche du plus court chemin entre deux points dans un graphe. Il a la particularité d'utiliser une fonction heuristique pour guider la recherche et donc l'ordre de visite des nœuds du graphe. Cette heuristique dépend, non seulement, du chemin déjà parcouru mais également d'une estimation du chemin qui reste à parcourir. Elle est définie par une fonction $f : N \to \mathbb{R}^+$:

$$f(x) = g(x) + h(x)$$

où :

- N est l'ensemble des nœuds du graphe.
- $h : N \to \mathbb{R}^+$ est une estimation de la distance entre le nœud x et le nœud t. Elle permet de guider l'algorithme pour qu'il choisisse le chemin le plus prometteur.
- $g : N \to \mathbb{R}^+$ est le distance réelle et connue la plus courte entre le nœud x et le nœud source s. Elle donne la distance parcourue depuis le nœud s pour atteindre x.

L'algorithme 17 est une adaptation de l'algorithme A^* au cas des graphes de transformation.

L'exploration d'un graphe a comme point de départ le nœud s. Ce nœud est l'unique nœud de l'ensemble des nœuds ouverts O et sera le premier à être visité. On appelle nœud courant le nœud de O ayant la plus petite valeur de f. Une fois choisi, le nœud courant est supprimé de l'ensemble des nœuds ouverts O et ajouté à l'ensemble des nœuds visités C. Ce dernier est vide au début de l'exploration. Les voisins du nœud courant, i.e. les nœuds liés au nœud courant par une arête, sont ensuite ajoutés à l'ensemble O s'ils n'ont jamais été visités. On recommence ensuite avec un nouveau nœud courant et ainsi de suite jusqu'à trouver le nœud t ou explorer l'ensemble du graphe. Ce dernier point correspond à obtenir l'ensemble O vide. Les nœuds dont la valeur de f est identique sont explorés en fonction de leur ordre d'ajout dans l'ensemble O.

Afin d'être *admissible*, c'est-à-dire être sûr de trouver, s'il existe, un chemin optimal, la fonction h doit être *minorante* sur N, c'est-à-dire :

$$\forall x, t \in N \ \forall S \in S_\Delta \text{ tel que } S(x) = t,\ \mathsf{cost}_S(x) \geqslant h(x)$$

où S est un chemin de transformation de x à t. De plus, si h est *monotone* sur N, c'est-à-dire :

$$\forall x, y, t \in N \ \forall S \in S_\Delta \text{ tel que } S(x) = y,\ h(x) - h(y) \leqslant \mathsf{cost}_S(x)$$

Algorithme 17 Algorithme A^* basé sur une heuristique h monotone et minorante par rapport à g sur N. Cet algorithme retourne un plus court chemin liant le nœud s au nœud t pour un graphe $G = (N, E)$ dirigé et étiqueté.

```
function A*_transfo(s, t)
 1: O := {s};
 2: C := ∅;
 3: g(s) := 0;
 4: f(s) := h(s);
 5: while O ≠ ∅ do
 6:     let x ∈ O
            s.t. f(x) = min({f(x) | x ∈ O});
 7:     O := O \ {x};
 8:     if x = t then
            return get_path(father, s, t);
 9:     C := C ∪ {x};
10:     for each (x, δ, y) ∈ succ_Δ(y) do
11:         let score := g(x) + cost_δ(x);
12:         if y ∉ O ∪ C or score < g(y) then
13:             father(y) := (x, δ);
14:             O := O ∪ {y};
15:             g(y) := score;
16:             f(y) := g(y) + h(y);
17:         end if
18:     end for
19: end while
20: return ∅;
end function

function get_path(father, s, y)
21: if s = y then
22:     return ε
23: else
24:     let (x, δ) = f(y);
25:     return get_path(father, s, x). (x, δ, y)
26: end if end function
```

où S est un chemin de transformation de x à t, alors chaque nœud visité ne pourra être revisité. Le premier chemin reliant la source à un nœud est donc un plus court chemin. L'algorithme A^* peut alors être implémenté plus efficacement.

L'utilisation de l'algorithme A^* se résume ainsi à la définition d'une fonction h. Celle-ci doit exprimer correctement le poids du chemin jusqu'au nœud final. Dans cet algorithme, nous définissons l'ensemble des fils d'un élément $x \in \mathcal{X}$ de la manière suivante :

$$\forall x \in \mathcal{X},\ succ_\Delta(x) = \{(x, \delta, y) \mid \delta(x) = y \text{ avec } \delta \in \Delta \text{ et } x, y \in \mathcal{X}\}$$

Nous réalisons sa construction à l'aide de la fonction $\mathsf{enhance}_{\mathsf{atom}}$ de l'algorithme 16. Celui-ci prend en entrée un terme de $\mathcal{T}_\mathcal{X}$ ainsi qu'un ensemble de déclarations de mode fonctionnelles M_Δ associées à une théorie T_Δ. Cet ensemble sera ordonné en fonction des opérations de transformation et des paramètres de ces fonctions. L'ordre joue un rôle important et permet d'assurer que, pour chaque clause de transformation, l'ordre des arêtes a été identique. Le choix de placer en premier les prédicats des concepts à apprendre permet de favoriser l'utilisation des clauses de transformation apprises lors de la construction de nouvelles clauses. Ceci permet l'apprentissage de programme récursif. Les prédicats de méta-opérations sont ensuite placés après ceux des opérations standards puisque qu'une méta-opération utilise les définitions d'opérations standards.

Distance d'édition

Afin de diriger notre recherche dans un graphe de transformation, nous désirons utiliser la notion de *distance d'édition*. Parfois appelée *métrique*, elle est une fonction définie sur un ensemble d'éléments $\mathcal{X}$ de même nature. Elle formalise l'idée intuitive de distance entre deux objets de cet ensemble. Plus formellement, une distance sur un ensemble $\mathcal{X}$ est une fonction $\mathsf{d} : \mathcal{X} \times \mathcal{X} \to \mathbb{R}^+$ où $\mathbb{R}^+$ est l'ensemble des réels positifs disposant des propriétés suivantes :

1. Séparation : $\mathsf{d}(x, x) = 0$,
2. Non-négativité : $0 \leqslant \mathsf{d}(x, y)$,
3. Symmetrie : $\mathsf{d}(x, y) = \mathsf{d}(y, x)$,
4. Inégalité triangulaire : $\mathsf{d}(x, z) \leqslant \mathsf{d}(x, y) + \mathsf{d}(y, z)$.

avec $x, y, z \in \mathcal{X}$.

Pour les mots

La distance d'édition [Levenshtein 1966] ou distance de Levenshtein, entre deux mots w et v, notée $\mathsf{cost}(w, v)$, est le nombre minimale d'opérations élémentaires pour passer d'un mot w à un mot v. Rappelons que, dans le cas des mots, nous bénéficions d'opérations élémentaires que sont la substitution, l'insertion et la délétion. Le coût de ces opérations est respectivement représenté par $\mathsf{cost}_{\mathsf{sub}}$, $\mathsf{cost}_{\mathsf{ins}}$ et $\mathsf{cost}_{\mathsf{del}}$. Le pseudo-code de l'algorithme du calcul de la distance d'édition est présenté ci-dessous.

— $\mathsf{cost}(\varepsilon, v) = |v| \times \mathsf{cost}_{\mathsf{ins}} = \mathsf{cost}_S(\varepsilon)$ t.q. $S(\varepsilon) = v$ avec $S = \mathsf{ins}_{0,v_0} \circ \cdots \circ \mathsf{ins}_{n,v_n}$,
— $\mathsf{cost}(w, \varepsilon) = |w| \times \mathsf{cost}_{\mathsf{del}} = \mathsf{cost}_S(w)$ t.q. $S(w) = \varepsilon$ avec $S = \mathsf{del}_m \circ \cdots \circ \mathsf{del}_0$,
— $\mathsf{cost}(wa, va) = \mathsf{cost}(w, v)$,
— $\mathsf{cost}(wa, vb) = \min \begin{cases} \mathsf{cost}(w, v) + \mathsf{cost}_{\mathsf{sub}} \\ \mathsf{cost}(wa, v) + \mathsf{cost}_{\mathsf{ins}} \\ \mathsf{cost}(w, vb) + \mathsf{cost}_{\mathsf{del}} \end{cases}$

où $v = v_0 \cdots v_n, w = w_0 \cdots w_m \in \Sigma^*$, $a, b \in \Sigma$.

Par des techniques de programmation dynamique, nous pouvons calculer cette distance d'édition entre deux mots w et v en temps quadratique $\mathcal{O}(|w| \times |v|)$ et linéaire $\mathcal{O}(\max(|w|, |v|))$ en espace comme présenté dans [Hirschberg 1975].

Notons que les coûts des opérations d'édition peuvent être n'importe quel entier positif. La distance de Hamming [Hamming 1950] peut être vue comme un cas particulier de la distance d'édition pour laquelle le coût des opérations d'insertion et de délétion est infini et le coût des opérations de substitution est égal à 1.

Pour les arbres

Nous présentons maintenant la distance d'édition définie pour les arbres proposée par [Zhang 1989]. La distance d'édition entre deux arbres est le nombre d'opérations élémentaires pour passer d'un arbre $t_1 \in T_\Sigma$ à un arbre $t_2 \in T_\Sigma$. Elle utilise comme opérations élémentaires les opérations d'insertion d'un nœud entre un nœud et un sous-ensemble de ses fils (ins), la suppression d'un nœud avec remontée de ses enfants (del) et le renommage d'un nœud (sub) vus lors de la sous-section 6.1.3.

La distance d'édition de [Zhang 1989] utilise, dans sa définition, la notion de forêt. Rappelons qu'une forêt est une séquence d'arbres. Pour une forêt F et la racine $root_t$ d'un arbre $t \in F$, nous notons $F - t$ la forêt obtenue par suppression de l'arbre t de F, $F - root_t$ celle obtenue par suppression de l'arbre t de F. Les sous-arbres des fils du nœud $root_t$

deviennent dans ce cas une séquence d'arbres de la forêt résultante $F - root_t$. Enfin, nous notons $F(root_t)$ la forêt d'arbres constituée des sous-arbres des fils de $root_t$.

Soit deux forêt d'arbres F_1 et F_2, les arbres t_1 et t_2 sont les arbres les plus à droite de F_1 et F_2 respectivement. Le pseudo-code de l'algorithme du calcul de la distance d'édition est présenté ci-dessous.

- $\mathsf{cost}(\varnothing, \varnothing) = 0$
- $\mathsf{cost}(F_1, \varnothing) = \mathsf{cost}(F_1 - root_{t_1}, \varnothing) + \mathsf{cost}_{\mathsf{del}}$
- $\mathsf{cost}(\varnothing, F_2) = \mathsf{cost}(\varnothing, F_2 - root_{t_2}) + \mathsf{cost}_{\mathsf{ins}}$
- $\mathsf{cost}(F_1, F_2) = \min \begin{cases} \mathsf{cost}(F_1 - root_{t_1}, F_2) + \mathsf{cost}_{\mathsf{del}} \\ \mathsf{cost}(F_1, F_2 - root_{t_2}) + \mathsf{cost}_{\mathsf{ins}} \\ \mathsf{cost}(F_1 - t_1, F_2 - t_2) + \mathsf{cost}(F_1(t_1), F_2(t_2)) + \mathsf{cost}_{\mathsf{sub}} \end{cases}$

où $\varnothing$ représente une forêt vide.

Plusieurs travaux [Demaine 2009, Pawlik 2011] portent sur l'amélioration de la complexité de l'algorithme calculant cette distance d'édition. Celle entre deux arbres t_1 et t_2 de T_Σ est calculable en temps cubique $\mathcal{O}(|N_{t_1}|^3)$ et est quadratique $\mathcal{O}(|N_{t_1}|^2)$ en espace, comme dans [Pawlik 2011].

Script optimal et distance d'édition

La notion de distance est étroitement liée à celle de coût lorsqu'on considère des scripts d'édition définis sur des objets structurés. Dans le cas simplifié où chaque opération élémentaire a un même coût, la distance d'édition entre deux objets correspond au nombre minimum d'opérations élémentaires d'édition nécessaires pour passer du premier objet au second. La distance d'édition entre deux objets $x, y \in \mathcal{X}$ est alors $\mathsf{d}(x, y) = \min(\{|S| \mid S(x) = y$ où $S \in S_\Delta\})$. On peut généraliser cette définition en prenant en compte le coût des opérations d'édition. Rappelons que le coût d'un script d'édition $S \in \mathcal{S}_\Delta$ est $\mathsf{cost}_S(x) = \mathsf{cost}_{\delta_{i_1}}(w) + \mathsf{cost}_{\delta_{i_2}}(\delta_{i_1}(x)) + \cdots + \mathsf{cost}_{\delta_{i_m}}(\delta_{i_1} \circ \cdots \circ \delta_{i_{m-1}}(x))$ avec $x \in \mathcal{X}$ et $S = (\delta_{i_1}, \cdots, \delta_{i_m})$. La *distance d'édition* devient alors : $\mathsf{d}(x, y) = \min(\{\mathsf{cost}_S(x) \mid S(x) = y$ où $S \in S_\Delta\})$. Elle est ainsi équivalente au coût des *scripts d'édition optimaux* de x à y. Nous utiliserons cette dernière.

Nous établissons maintenant quelques propriétés intéressantes pour les scripts d'édition. Ces propriétés vont être utiles lors de l'utilisation d'une distance d'édition comme heuristique de l'algorithme A^*. Elles permettront d'établir son admissibilité ainsi que sa monotonicité.

Nous supposons que nous disposons d'une distance d'édition $\mathsf{d} : \mathcal{X} \times \mathcal{X} \to \mathbb{R}^+$ ainsi que d'un ensemble d'opérations élémentaires Δ définies sur $\mathcal{X}$. Nous fixons le coût de chaque opération d'édition de Δ tel que $\forall \delta \in \Delta\ \forall x \in \mathcal{X}, \mathsf{cost}_\delta(x) \geqslant \mathsf{d}(x, \delta(x))$. Le coût d'application d'une opération d'édition δ sur un élément x est alors supérieur ou égal à la distance d'édition entre x et $\delta(x)$. Nous étudions maintenant l'apprentissage d'une transformation entre deux éléments d'un ensemble $\mathcal{X}$ à l'aide de l'algorithme A^* guidé par cette distance d'édition. Elle se base sur l'usage d'opérations élémentaires définies sur $\mathcal{X}$ formant l'ensemble des opérations élémentaires Δ_d. Afin de former le graphe de transformation entre deux éléments de $\mathcal{X}$, nous utilisons l'ensemble d'opérations élémentaires Δ. Il est intéressant d'inclure dans Δ les opérations de Δ_d. Ceci permet d'assurer qu'un chemin existe entre deux éléments si la distance d'édition entre eux est calculable. Dans le cas contraire, il convient de borner la taille et/ou le coût des chemins de transformation afin de limiter le nombre de nœuds explorés. Cela ne garantie plus la découverte du plus court chemin liant deux éléments.

Nous prouvons, dans ce qui suit, qu'une distance d'édition d est une heuristique admissible et monotique. Elle peut donc être utilisée avec succès par l'algorithme A^*. Ceci

permettra une recherche efficace d'un plus court chemin entre deux éléments de $\mathcal{X}$. Nous supposons, pour cela, que $\forall x, t \in \mathcal{X}, h(x) = \mathsf{d}(x,t)$ et $\forall x, s \in \mathcal{X}, g(x) = \mathsf{cost}_S(s)$ t.q. $S(s) = x$.

Lemme 29. *$\forall x, y \in \mathcal{X}$ $\forall S \in S_\Delta$ tel que $S(x) = y$, $\mathsf{cost}_S(x) \geqslant \mathsf{d}(x,y)$*

Preuve. On sait que $\forall \delta \in \Delta\ \forall x \in \mathcal{X}, \mathsf{cost}_\delta(x) \geqslant \mathsf{d}(x, \delta(x))$. On suppose que $S = (\delta_{i_1}, \delta_{i_2}, \cdots, \delta_{i_m})$ Or :

$$\mathsf{cost}_S(x) = \mathsf{cost}_{\delta_{i_1}}(x) + \mathsf{cost}_{\delta_{i_2}}(\delta_{i_1}(x)) + \cdots + \mathsf{cost}_{\delta_{i_m}}(\delta_{i_1} \circ \cdots \circ \delta_{i_{m-1}}(x))$$

$$\mathsf{cost}_S(x) \geqslant \mathsf{d}(x, \delta_{i_1}(x)) + \mathsf{d}(\delta_{i_1}(x), \delta_{i_1} \circ \delta_{i_2}(x)) + \cdots + \mathsf{d}(\delta_{i_1} \circ \cdots \circ \delta_{i_{m-1}}(x), S(x))$$

Grâce à l'inégalité triangulaire, on a donc :

$$\mathsf{cost}_S(x) \geqslant \mathsf{d}(x, \delta_{i_1}(x)) + \mathsf{d}(\delta_{i_1}(x), \delta_{i_1} \circ \delta_{i_2}(x)) + \cdots + \mathsf{d}(\delta_{i_1} \circ \cdots \circ \delta_{i_{m-1}}(x), S(x)) \geqslant \mathsf{d}(x, S(x))$$

$$\mathsf{cost}_S(x) \geqslant \mathsf{d}(x, S(x))$$

□

Lemme 30. *$\forall x, y, z \in \mathcal{X}$ $\forall S \in S_\Delta$ tel que $S(x) = y$, $\mathsf{d}(x,z) - \mathsf{d}(y,z) \leqslant \mathsf{cost}_S(x)$*

Preuve. On veut montrer que $\forall x, y, z \in \mathcal{X}$ $\forall S \in S_\Delta$ t.q. $S(x) = y$, $\mathsf{d}(x,z) - \mathsf{d}(y,z) \leqslant \mathsf{cost}_S(x)$. On suppose que l'on a $x, y, z \in \mathcal{X}$ $\forall S \in S_\Delta$ t.q. $S(x) = y$. On part de l'inégalité triangulaire :

$$\mathsf{d}(x,z) \leqslant \mathsf{d}(x,y) + \mathsf{d}(y,z)$$

$$\mathsf{d}(x,z) - \mathsf{d}(y,z) \leqslant \mathsf{d}(x,y)$$

Or on sait que $\forall w, z \in \mathcal{X}$ $\forall S \in S_\Delta$ t.q. $S(w) = z$, $\mathsf{cost}_S(w) \geqslant \mathsf{d}(w,z)$ d'où :

$$\mathsf{d}(x,z) - \mathsf{d}(y,z) \leqslant \mathsf{d}(x,y) \leqslant \mathsf{cost}_S(w)$$

□

Une distance d'édition $\mathsf{d} : \mathcal{X} \times \mathcal{X} \to \mathbb{R}^+$ est ainsi une heuristique admissible et monotique. Elle peut être utilisée pour la recherche d'un plus court chemin entre deux éléments de $\mathcal{X}$. Le chemin de transformation trouvé par l'algorithme A^* peut alors être retranscrit sous la forme d'une séquence d'atomes de transformation fondés. Cette séquence sert de base à la construction d'une clause de transformation. Avant de continuer, nous illustrons ce procédé de recherche.

Exemple 62. *Soit les mots d'entrée "gérer" et le mot de sortie "nous gérons" représentés par les termes :*

$t_0 = cons(g, cons(é, cons(r, cons(e, cons(r, nil)))))$
$t_8 = cons(n, cons(o, cons(u, cons(s, cons(' ', cons(g, cons(é, cons(r, cons(o, cons(n, cons(s, nil))))))))))$

et une théorie du domaine contenant les opérations élémentaires définies par les modes :

$\mathsf{ins}(+\mathcal{T}_{\Sigma^\star}, +\mathcal{T}_\Sigma, +\mathcal{T}_{\lvert w\rvert-1}, +\mathcal{T}_{\Sigma^\star})$ $\mathsf{ins}(\#\mathcal{T}_{\Sigma^\star}, \#\mathcal{T}_\Sigma, \#\mathcal{T}_{\lvert w\rvert-1}, \#\mathcal{T}_{\Sigma^\star})$	*insertion d'une lettre à une position dans un mot*
$\mathsf{sub}(+\mathcal{T}_{\Sigma^\star}, +\mathcal{T}_\Sigma, +\mathcal{T}_{\lvert w\rvert-1}, +\mathcal{T}_{\Sigma^\star})$ $\mathsf{sub}(\#\mathcal{T}_{\Sigma^\star}, \#\mathcal{T}_\Sigma, \#\mathcal{T}_{\lvert w\rvert-1}, \#\mathcal{T}_{\Sigma^\star})$	*substitution d'une lettre à une position donnée par une autre lettre dans un mot*

Le chemin de transformation appris est le suivant :

$$\mathsf{transfo}(t_0, t_8) \leftarrow \mathsf{ins}(t_0, ' ', 0, t_1),\ \mathsf{ins}(t_1, n, 0, t_2),\ \mathsf{ins}(t_2, n, 6, t_3),\ \mathsf{ins}(t_3, o, 1, t_4),\ \mathsf{ins}(t_4, s, 2, t_5),\ \mathsf{ins}(t_5, u, 2, t_6),\ \mathsf{sub}(t_6, o, 8, t_7),\ \mathsf{sub}(t_7, s, 10, t_8).$$

où :

$t_1 = cons(' ', cons(g, cons(é, cons(r, cons(e, cons(r, nil))))))$
$t_2 = cons(n, cons(' ', cons(g, cons(é, cons(r, cons(e, cons(r, nil)))))))$
$t_3 = cons(n, cons(' ', cons(g, cons(é, cons(r, cons(e, cons(n, cons(r, nil))))))))$
$t_4 = cons(n, cons(o, cons(' ', cons(g, cons(é, cons(r, cons(e, cons(n, cons(r, nil)))))))))$
$t_5 = cons(n, cons(o, cons(s, cons(' ', cons(g, cons(é, cons(r, cons(e, cons(n, cons(r, nil))))))))))$
$t_6 = cons(n, cons(o, cons(u, cons(s, cons(' ', cons(g, cons(é, cons(r, cons(e, cons(n, cons(r, nil)))))))))))$
$t_7 = cons(n, cons(o, cons(u, cons(s, cons(' ', cons(g, cons(é, cons(r, cons(o, cons(n, cons(r, nil)))))))))))$

6.3.4 Construction d'une clause de transformation à partir d'un exemple

Nous disposons, à ce stade, d'une méthode de construction des atomes de contexte pour un terme donné ainsi que d'une autre pour la construction des atomes de transformation. Nous les regroupons au sein de l'algorithme 18. Ce regroupement forme la fonction enhance nécessaire à l'algorithme 6.3. Celle-ci retourne une clause de transformation.

Algorithme 18 Construction d'une clause de transformation dont la tête est $\mathsf{transfo}(t_0, t_n)$ à partir d'une théorie de transformation T associée à un ensemble de déclarations de mode M_T.

function enhance(h, T, M_T)
1: **let** $T = T_\Delta \cup T_C$ **and** $M_T = M_\Delta \cup M_C$;
2: **let** $h = \mathsf{transfo}(t_0, t_n)$;
3: **let** $\ell^\delta_{t_0}, \ell^\delta_{t_1}, \ell^\delta_{t_2}, \cdots, \ell^\delta_{t_{n-1}} = A^*_{\mathsf{transfo}}(t_0, t_n)$;
4: $E := \mathsf{transfo}(t_0, t_n) \leftarrow$;
5: **for** $j = 0$ **to** $n - 1$ **do**
6: **let** $E = h \leftarrow b_1, \cdots, b_k$.
7: **let** $C_{t_j} = \ell^0_{t_j}, \cdots, \ell^{i_j}_{t_j} = \mathsf{enhance}_{\mathsf{atom}}(t_j, T_C, M_C)$;
8: $E := h \leftarrow b_1, \cdots, b_k, C_{t_j}, \ell^\delta_{t_j}$.
9: **end for**
10: **return** E ;
end function

L'algorithme 18 réalise la construction d'une clause de transformation à partir d'un atome $\mathsf{transfo}(t_0, t_n)$ en deux étapes. La première correspond à la construction d'un chemin liant le terme d'entrée t_0 au terme de sortie t_n. Ce chemin est découvert grâce à l'algorithme 17. Il est représenté par une séquence d'atomes de transformation. La seconde correspond à l'ajout d'un contexte pour chaque terme d'entrée de type $\mathcal{T}_\mathcal{X}$ présent dans les atomes de transformation. Le contexte d'un terme se situe dans la clause avant le premier atome de transformation qui l'utilise comme entrée. Ce point permet d'assurer le respect des déclarations de mode de M_T. La clause de transformation ainsi produite est fondée.

6.3.5 Généralisation des clauses de transformation

Nous nous intéressons maintenant à la généralisation de clauses de transformation obtenues par la fonction enhance. Une clause de transformation contient un chemin entre un terme d'entrée et un terme de sortie. Elle contient également les contextes de chaque terme de type $\mathcal{T}_\mathcal{X}$ de ce chemin.

Algorithme 19 Généralisation de deux clauses de transformation P et Q respectant une théorie de transformation T associée à un ensemble de déclarations de mode M_T

function generalize(P, Q, M_T)
1: **let** $M_T = M_\Delta \cup M_C$;
2: **let** $P = \mathsf{transfo}(p_0, p_n) \leftarrow C_{p_0}, \ell^\delta_{p_0}, \cdots, C_{p_{n-1}}, \ell^\delta_{p_{n-1}}.$;
3: **let** $Q = \mathsf{transfo}(q_0, q_m) \leftarrow C_{q_0}, \ell^\delta_{q_0}, \cdots, C_{q_{m-1}}, \ell^\delta_{q_{m-1}}.$;
4: **if** $n \neq m$ **then return** $null$;
5: $\Theta := \varnothing$;
6: $\mathsf{transfo}(r_0, r_n) := \mathsf{gen\_atom}(\mathsf{transfo}(p_0, p_n), \mathsf{transfo}(q_0, q_n), \Theta)$;
7: $V_{r_{-1}} := \emptyset$;
8: **for** $i = 0$ **to** $n - 1$ **do**
9: $\quad \ell^\delta_{r_i} := \mathsf{gen\_atom}(\ell^\delta_{p_i}, \ell^\delta_{q_i}, \Theta)$;
10: $\quad$ **if** $\ell^\delta_{r_i} = \varnothing$
$\quad$ **or** $\ell^\delta_{r_i}$ does not respect a mode in M_Δ
$\quad$ **or** ($i \neq 0$ **and** $\mathsf{output}^{M_\Delta}_{\mathcal{T}_\mathcal{X}}(\ell^\delta_{r_{i-1}}) \neq r_i$) **then**
11: $\quad\quad$ **return** $null$;
12: $\quad C_{r_i} := \mathsf{gen\_atoms\_set}(C_{p_i}, C_{q_i}, \Theta)$;
13: $\quad V_{r_i} := V_{r_{i-1}} \cup \{r_i\}$;
14: $\quad C_{r_i} := \mathsf{get\_consistent\_context}(C_{r_i}, M_C, V_{r_i})$;
15: $\quad V_{r_i} := V_{r_i} \cup \mathsf{input}^{M_C}(C_{r_i}) \cup \mathsf{output}^{M_C}(C_{r_i})$;
16: $\quad$ **if** $\mathsf{input}(\ell^\delta_{r_i}) \not\subseteq V_{r_i}$ **then**
$\quad\quad$ **return** $null$;
17: **end for**
18: **return** $\mathsf{transfo}(r_0, r_n) \leftarrow C_{r_0}, \ell^\delta_{r_0}, \cdots, C_{r_{n-1}}, \ell^\delta_{r_{n-1}}.$;
end

Algorithme 20 Généralisation pour la transformation.

function gen_atom$(\ell, \ell', \mathbf{var}\ \Theta)$
1: **let** $\ell = R(t_1, \ldots, t_n)$;
2: **let** $\ell' = P(s_1, \ldots, s_m)$;
3: **if** $R \neq P$ **or** $n \neq m$ **then**
4: $\quad$ **return** $\varnothing$;
5: **for** $i \in \{1, \ldots, n\}$ **do**
6: $\quad$ **if** $t_i = s_i$ **then**
7: $\quad\quad t^\star_i = t_i$;
8: $\quad$ **else if** ($t_i = X$ **or** $s_i = Y$)
$\quad$ **and** $\exists t_i/s_i/t^\star \in \Theta$ **then**
9: $\quad\quad t^\star_i = t^\star$;
10: $\quad$ **else**
11: $\quad\quad t^\star_i :=$ **new** variable ;
12: $\quad\quad \Theta := \Theta \cup \{t_i/s_i/t^\star_i\}$;
13: **end for**
14: **return** $\{R(t^\star_1, \ldots, t^\star_n)\}$;
end function

function gen_atoms_set(C, D, Θ)
15: $G := \varnothing$;
16: **for** $\ell \in C$ **do**
17: $\quad$ **for** $\ell' \in D$ **do**
18: $\quad\quad G := G \cup \mathsf{gen\_atom}(\ell, \ell', \Theta)$;
19: $\quad$ **end for**
20: **end for**
21: **return** G ;
end function

La généralisation de deux clauses de transformation P et Q suit ce découpage. Elle com-

mence par la généralisation du chemin de transformation comme présenté à l'algorithme 19. Le nombre d'atomes de transformation présents dans ces clauses doit être le même. Chaque atome de la séquence $(\ell^{\delta}_{p_0}, \cdots, \ell^{\delta}_{p_{n-1}})$ issue de P est généralisé avec l'atome à cette même position de la séquence $(\ell^{\delta}_{q_0}, \cdots, \ell^{\delta}_{q_{n-1}})$ issue de Q. On construit ainsi une nouvelle séquence d'atomes généralisés de même taille. La généralisation atome par atome est réalisée par la fonction gen_atom de l'algorithme 20. Celle-ci conserve les termes en l'état s'ils sont identiques ou les variabilise dans le cas contraire. Si la séquence obtenue après généralisation n'est pas de taille similaire à celles issues de P et Q alors l'algorithme 19 retourne *null* et la généralisation des contextes n'est pas nécessaire. La généralisation des contextes consiste à généraliser pour tout entier $i \in \{1, \cdots, n-1\}$ l'ensemble d'atomes de contexte C_{p_i} de C avec l'ensemble C_{q_i} de Q. Cette généralisation est réalisée par la fonction gen_atoms_set présente à l'algorithme 19. Elle opère à la manière du moindre généralisée en généralisant chaque littéral du premier ensemble avec chaque littéral du second ensemble à l'aide de la fonction gen_atom. La généralisation de ces contextes ne respecte plus forcément les déclarations de mode, il est donc nécessaire de vérifier cela. La fonction get_consistent_context de l'algorithme 22 s'en charge. Elle recherche, en premier lieu, les atomes de contexte respectant une déclaration de mode dont les paramètres sont fondés, puis ceux respectant une déclaration de mode et utilisant les sorties de ces atomes et ainsi de suite. Cela permet d'établir la dépendance des atomes de contexte les uns avec les autres.

L'algorithme 19 retourne ensuite la clause de transformation. Cette clause contient les atomes de transformation généralisés ainsi que les atomes de contexte associés. Contrairement aux atomes de transformation obtenus par généralisation sélective, les atomes de contexte ont, quant-à-eux, été générés par un produit de littéraux. Seuls les atomes de contexte connectés aux atomes de transformation et respectueux d'une déclaration de mode ont été conservés. Parmi ces atomes, certains ne sont peut-être pas pertinents par rapport à la transformation. Leur suppression permettrait alors d'étendre la couverture de la clause de transformation. Cette étape correspond à l'élagage d'une clause de transformation.

6.3.6 Elagage de la généralisation

L'élagage d'une clause de transformation est réalisé par la fonction prune de l'algorithme 21. Il a pour but de supprimer dans une clause de transformation tout atome de contexte dispensable aux atomes de transformation.

Pour ce faire, l'opération d'élagage traite les ensembles d'atomes de contexte de chaque atome de transformation. Ce traitement est effectué pour chaque contexte en fonction de leur ordre d'apparition dans la clause. Pour un contexte C_{t_j}, on commence par la suppression d'un atome de contexte $\ell^{k}_{t_j}$. Cette suppression peut causer la perte de respect des déclarations de mode. L'algorithme get_consistent_context est alors appliqué sur la clause afin de ne conserver que les littéraux respectueux des déclarations de mode contextuelles.

Algorithme 21 Algorithme d'élagage : $\mathsf{prune}(C, E, M_T)$. Élague une clause de transformation en fonction d'un ensemble d'exemples E et de déclarations de mode M_T. Cet algorithme retourne une clause de transformation.

function $\mathsf{prune}(C, E, M_T)$
1: **let** $M_T = M_\Delta \cup M_C$;
2: **let** $C = \mathsf{transfo}(t_0, t_n) \leftarrow C_{t_0}, \ell^\delta_{t_0}, \cdots, C_{t_{n-1}}, \ell^\delta_{t_{n-1}}.$;
3: $cover := \{e \in E \mid C \wedge T \vdash e\}$;
4: **for** $j = 0$ **to** $n - 1$ **do**
5: **let** $C_{t_j} = \ell^0_{t_j}, \cdots, \ell^{i_j}_{t_j}$;
6: **for** $k = 0$ **to** i_j **do**
7: $C' := C$;
8: $C_{t_j} := C_{t_j} \setminus \{\ell^k_{t_j}\}$
9: $C := \mathsf{get\_consistent\_clause}(C, M_T)$;
10: $C := \mathsf{remove\_useless\_outputs}(C, M_T)$;
11: **if** $C = null$ **then**
12: $C := C'$;
13: **else**
14: $cover' := \{\mathsf{transfo}(in, out') \mid C \wedge T \vdash \mathsf{transfo}(in, out') \text{ with } \mathsf{transfo}(in, out) \in E\}$;
15: **if** $cover' \setminus E \neq \emptyset$ **or** $cover \not\subseteq cover'$ **then**
16: $C := C'$;
17: **else**
18: $cover := cover'$;
19: **end if**
20: **end if**
21: **end for**
22: **end for**
23: **return** $\mathsf{transfo}(t_0, t_n) \leftarrow C_{t_0}, \ell^\delta_{t_0}, \cdots, C_{t_{n-1}}, \ell^\delta_{t_{n-1}}.$;
end function

Algorithme 22 Suppression d'atomes ne respectant pas un ensemble de modes M. Ces algorithmes prennent en entrée une clause de transformation (get_consistent_clause) ou un ensemble d'atomes de contexte et un ensemble de variables d'entrée V (get_consistent_context).

function get_consistent_context(G_C, M_C, V)
1: $R := \emptyset$;
2: $G' := \emptyset$;
3: **while** $G' \neq G_C$ **do**
4: $\quad V := V \cup \mathsf{output}^{M_C}(R)$;
5: $\quad G' := G_C$;
6: $\quad R := R \cup \{\ell \in G_C \mid \ell$ respects a mode in M_C s.t. $\mathsf{input}(\ell) \subseteq V\}$;
7: $\quad G_C := G_C \setminus R$;
8: **end while**
9: **return** R;
end function

function get_consistent_clause(C, M_T)
10: **let** $M_T = M_\Delta \cup M_C$;
11: **let** $C = \mathsf{transfo}(t_0, t_n) \leftarrow C_{t_0}, \ell^\delta_{t_0}, \cdots, C_{t_{n-1}}, \ell^\delta_{t_{n-1}}.$;
12: $V := \mathsf{input}^{M_\Delta}_{\mathcal{T}_\mathcal{X}}(\mathsf{transfo}(t_0, t_n))$;
13: **for** $i = 0$ to $n-1$ **do**
14: $\quad C_{t_i} :=$ get_consistent_context(C_{t_i}, M_T, V);
15: $\quad V := V \cup \mathsf{input}^{M_C}(C_{t_i}) \cup \mathsf{output}^{M_C}(C_{t_i})$;
16: $\quad$ **if** $\mathsf{input}(\ell^\delta_{t_i}) \not\subseteq V$ **then**
17: $\quad\quad$ **return** *null*;
18: $\quad V := V \cup \mathsf{output}^{M_\Delta}(\ell^\delta_{t_i})$;
19: **end for**
20: **if** $\mathsf{output}^{M_\Delta}(\mathsf{transfo}(t_0, t_n)) \not\subseteq V$ **then**
21: $\quad\quad$ **return** *null*;
22: **return** C;
end function

Algorithme 23 Suppression d'atomes de contexte dont les sorties sont inutilisées par les atomes de transformation. Cet algorithme prend en entrée une clause de transformation h et des déclarations de mode M_T et retourne une clause de transformation.

function remove_useless_outputs(C, M_T)
1: **let** $M_T = M_\Delta \cup M_C$;
2: **let** $C = \mathsf{transfo}(t_0, t_n) \leftarrow C_{t_0}, \ell^\delta_{t_0}, \cdots, C_{t_{n-1}}, \ell^\delta_{t_{n-1}}.$;
3: **for** $i = 0$ **to** $n-1$ **do**
4: $\quad C'_{t_i} = \{\ell \in C_{t_i} \mid \mathsf{output}^{M_C}(\ell) = \emptyset\}$;
5: $\quad V := \mathsf{input}^{M_\Delta}(\ell^\delta_{t_i}) \cup \{\mathsf{input}^{M_C}(\ell) \mid \ell \in C'_{t_i}\}$;
6: $\quad$ **while** $V \neq \emptyset$ **do**
7: $\quad\quad V := V \setminus \{X\}$;
8: $\quad\quad$ **for** $\ell \in C_{t_i}$ **s.t.** $\mathsf{output}^{M_C}(\ell) = X$ **do**
9: $\quad\quad\quad C_{t_i} := C_{t_i} \setminus \{\ell\}$;
10: $\quad\quad\quad C'_{t_i} := C'_{t_i} \cup \{\ell\}$;
11: $\quad\quad\quad V := V \cup \mathsf{input}^{M_C}(\ell)$;
12: $\quad\quad$ **end for**
13: $\quad$ **end while**
14: **end for**
15: **return** $\mathsf{transfo}(t_0, t_n) \leftarrow C'_{t_0}, \ell^\delta_{t_0}, \cdots, C'_{t_{n-1}}, \ell^\delta_{t_{n-1}}.$;
end function

Suite à cela, les littéraux dont les sorties ne sont plus utiles à la clause de transformation sont supprimés par la fonction remove_useless_outputs de l'algorithme 23. La clause de transformation ainsi obtenue est testée sur les exemples afin de vérifier qu'elle couvre toujours les exemples précédemment couverts ainsi que, si possible, de nouveaux exemples. Une fois tous les littéraux de la clause traités, l'algorithme 21 retourne une clause de transformation.

Cette clause respecte l'ensemble des déclarations de mode M_T.

Nous illustrons la généralisation de deux clauses de transformation enrichies avec l'exemple suivant :

Exemple 63. *Soit deux exemples de $E_{(\Sigma^\star)^2}$ avec $\Sigma = \{g, é, r, e, o, u, s, n\}$:*

$$\begin{array}{rcl} e_1 & = & (\ gérer\ ,\ nous\ gérons\ ) \\ e_2 & = & (\ régner\ ,\ nous\ régnons\ ) \end{array}$$

Les déclarations de mode des opérations élémentaires définies de la théorie de transformation sont :

$\mathsf{ins}(+\mathcal{T}_{\Sigma^\star}, +\mathcal{T}_\Sigma, +\mathcal{T}_{|w|-1}, +\mathcal{T}_{\Sigma^\star})$ *insertion d'une lettre à une position dans un mot*
$\mathsf{ins}(\#\mathcal{T}_{\Sigma^\star}, \#\mathcal{T}_\Sigma, \#\mathcal{T}_{|w|-1}, \#\mathcal{T}_{\Sigma^\star})$

$\mathsf{sub}(+\mathcal{T}_{\Sigma^\star}, +\mathcal{T}_\Sigma, +\mathcal{T}_{|w|-1}, +\mathcal{T}_{\Sigma^\star})$ *substitution d'une lettre à une position donnée par*
$\mathsf{sub}(\#\mathcal{T}_{\Sigma^\star}, \#\mathcal{T}_\Sigma, \#\mathcal{T}_{|w|-1}, \#\mathcal{T}_{\Sigma^\star})$ *une autre lettre dans un mot*

et celles des opérations de contexte sont :

— *$firstNthLetter(w, n, l, p)$ indique la position p de la $n^{ième}$ lettre l dans le mot w et a pour modes :*

$firstNthLetter(+\mathcal{T}_{\Sigma^\star}, \#\mathcal{T}_\Sigma, \#\mathcal{T}_{|w|-1}, \#\mathcal{T}_{|w|-1})$
$firstNthLetter(\#\mathcal{T}_{\Sigma^\star}, \#\mathcal{T}_\Sigma, \#\mathcal{T}_{|w|-1}, \#\mathcal{T}_{|w|-1})$
$firstNthLetter(+\mathcal{T}_{\Sigma^\star}, \#\mathcal{T}_\Sigma, \#\mathcal{T}_{|w|-1}, -\mathcal{T}_{|w|-1})$

L'opération $lastNthLetter(w, n, l, p)$ exécute la même tâche en partant de la fin du mot.

— *$next(w, n, p)$ indique que dans le mot w la lettre à la position n est suivie d'une lettre à la position p et a pour modes :*

$next(+\mathcal{T}_{\Sigma^\star}, \#\mathcal{T}_{|w|-1}, \#\mathcal{T}_{|w|-1})$
$next(\#\mathcal{T}_{\Sigma^\star}, \#\mathcal{T}_{|w|-1}, \#\mathcal{T}_{|w|-1})$
$next(+\mathcal{T}_{\Sigma^\star}, \#\mathcal{T}_{|w|-1}, -\mathcal{T}_{|w|-1})$

L'opération inverse de $next(w, n, p)$ est $prev(w, p, n)$.

— *$posBeg(w, n, l)$ indique la lettre l à la position n dans w et a pour modes :*

$posEnd(+\mathcal{T}_{\Sigma^\star}, \#\mathcal{T}_{|w|-1}, -\mathcal{T}_\Sigma)$
$posEnd(\#\mathcal{T}_{\Sigma^\star}, \#\mathcal{T}_{|w|-1}, \#\mathcal{T}_\Sigma)$
$posEnd(+\mathcal{T}_{\Sigma^\star}, \#\mathcal{T}_{|w|-1}, \#\mathcal{T}_\Sigma)$

L'opération $posEnd(w, n, l, p)$ exécute la même tâche en partant de la fin du mot.

La clause de transformation obtenue par généralisation et élagage est :

$$\begin{array}{rcl} \mathsf{transfo}(X0, X1) & \leftarrow & C_{X_0},\ \mathsf{ins}(X0,'\ ', 0, X2),\ C_{X_2}, \mathsf{ins}(X2, n, 0, X3),\ C_{X_3},\ \mathsf{ins}(X3, n, X4, X5), \\ & & C_{X_5},\ \mathsf{ins}(X5, o, 1, X6),\ C_{X_6},\ \mathsf{ins}(X6, s, 2, X7),\ C_{X_7},\ \mathsf{ins}(X7, u, 2, X8), \\ & & C_{X_8},\ \mathsf{sub}(X8, o, X9, X10),\ C_{X_{10}},\ \mathsf{sub}(X10, s, X11, X1). \end{array}$$

où :

$$\begin{array}{rcl} C_{X_0} & = & next(X0, 1, 2),\ next(X0, 0, 1),\ prev(X0, 2, 1),\ prev(X0, 1, 0) \\ C_{X_2} & = & firstNthLetter(X2,'\ ', 1, 0),\ lastNthLetter(X2,'\ ', 1, 0),\ lengthCons(X2, X4), \\ & & next(X2, 1, 2),\ next(X2, 0, 1),\ posBeg(X2, 0,'\ '),\ prev(X2, 2, 1),\ prev(X2, 1, 0) \\ C_{X_3} & = & firstNthLetter(X3, n, 1, 0),\ firstNthLetter(X3,'\ ', 1, 1),\ lastNthLetter(X3,'\ ', 1, 1), \\ & & next(X3, 1, 2),\ next(X3, 0, 1),\ posBeg(X3, 0, n),\ posBeg(X3, 1,'\ '),\ prev(X3, 2, 1), \\ & & prev(X3, 1, 0) \\ C_{X_5} & = & firstNthLetter(X5, n, 1, 0),\ firstNthLetter(X5,'\ ', 1, 1),\ lastNthLetter(X5, n, 1, X4), \\ & & lastNthLetter(X5,'\ ', 1, 1),\ lengthCons(X5, X9),\ next(X5, 1, 2),\ next(X5, 0, 1), \\ & & posBeg(X5, 0, n),\ posBeg(X5, 1,'\ '),\ posEnd(X5, 1, n),\ prev(X5, 2, 1),\ prev(X5, 1, 0) \end{array}$$

C_{X_6} = $firstNthLetter(X6, n, 1, 0)$, $firstNthLetter(X6, o, 1, 1)$, $firstNthLetter(X6,' ', 1, 2)$, $lastNthLetter(X6, o, 1, 1)$, $lastNthLetter(X6,' ', 1, 2)$, $next(X6, 1, 2)$, $next(X6, 0, 1)$, $posBeg(X6, 0, n)$, $posBeg(X6, 1, o)$, $posBeg(X6, 2,' ')$, $posEnd(X6, 1, n)$, $prev(X6, 2, 1)$, $prev(X6, 1, 0)$

C_{X_7} = $firstNthLetter(X7, n, 1, 0)$, $firstNthLetter(X7, o, 1, 1)$, $firstNthLetter(X7, s, 1, 2)$, $lastNthLetter(X7, n, 1, X9)$, $lastNthLetter(X7, o, 1, 1)$, $lastNthLetter(X7, s, 1, 2)$, $lengthCons(X7, X11)$, $next(X7, 1, 2)$, $next(X7, 0, 1)$, $posBeg(X7, 0, n)$, $posBeg(X7, 1, o)$, $posBeg(X7, 2, s)$, $posEnd(X7, 1, n)$, $prev(X7, 2, 1)$, $prev(X7, 1, 0)$

C_{X_8} = $firstNthLetter(X8, n, 1, 0)$, $firstNthLetter(X8, o, 1, 1)$, $firstNthLetter(X8, u, 1, 2)$, $lastNthLetter(X8, o, 1, 1)$, $lastNthLetter(X8, u, 1, 2)$, $next(X8, 1, 2)$, $next(X8, 0, 1)$, $posBeg(X8, 0, n)$, $posBeg(X8, 1, o)$, $posBeg(X8, 2, u)$, $posEnd(X8, 1, n)$, $prev(X8, 2, 1)$, $prev(X8, 1, 0)$,

$C_{X_{10}}$ = $firstNthLetter(X10, n, 1, 0)$, $firstNthLetter(X10, o, 1, 1)$, $firstNthLetter(X10, o, 2, X9)$, $firstNthLetter(X10, u, 1, 2)$, $lastNthLetter(X10, o, 1, X9)$, $lastNthLetter(X10, o, 2, 1)$, $lastNthLetter(X10, u, 1, 2)$, $next(X10, 1, 2)$, $next(X10, 0, 1)$, $posBeg(X10, 0, n)$, $posBeg(X10, 1, o)$, $posBeg(X10, 2, u)$, $posEnd(X10, 2, o)$, $posEnd(X10, 1, n)$, $prev(X10, 2, 1)$, $prev(X10, 1, 0)$

Il est également possible d'apprendre des programmes récursifs comme suit.

Exemple 64. *Cet exemple illustre l'apprentissage d'une méta-opération sous la forme d'un programme récursif simple. Cette méta-opération consiste en la substitution totale d'une lettre en une autre dans un mot. Soit les exemples suivants :*

$$\begin{array}{lcc} e_1 & = & (\ ab\ ,\ bb\ ) \\ e_2 & = & (\ bba\ ,\ bbb\ ) \\ e_3 & = & (\ bab\ ,\ bbb\ ) \\ e_4 & = & (\ bbaa\ ,\ bbbb\ ) \\ e_5 & = & (\ aab\ ,\ bbb\ ) \\ e_6 & = & (\ aba\ ,\ bbb\ ) \\ e_7 & = & (\ baa\ ,\ bbb\ ) \\ e_8 & = & (\ bbbaaa\ ,\ bbbbbb\ ) \end{array}$$

Nous utilisons la même théorie du domaine que l'exemple 63 à laquelle est ajoutée l'opération de contexte :

$$nbLetter : \mathcal{T}_{\Sigma^\star} \times \mathcal{T}_{\Sigma} \rightarrow \mathcal{T}_{|w|}$$

Cette opération retourne le nombre d'occurrences d'une lettre donnée dans un mot et a pour modes :

$nbLetter(+\mathcal{T}_{\Sigma^\star}, +\mathcal{T}_{\Sigma}, -\mathcal{T}_{|w|})$
$nbLetter(+\mathcal{T}_{\Sigma^\star}, +\mathcal{T}_{\Sigma}, \#\mathcal{T}_{|w|})$
$nbLetter(+\mathcal{T}_{\Sigma^\star}, \#\mathcal{T}_{\Sigma}, -\mathcal{T}_{|w|})$
$nbLetter(+\mathcal{T}_{\Sigma^\star}, \#\mathcal{T}_{\Sigma}, \#\mathcal{T}_{|w|})$
$nbLetter(\#\mathcal{T}_{\Sigma^\star}, \#\mathcal{T}_{\Sigma}, \#\mathcal{T}_{|w|})$

Les clauses de transformation apprises sont :

$\mathsf{transfo}(X0, X1) \leftarrow firstNthLetter(X0, a, 1, X2),\ nbLetter(X0, a, 1),\ \mathsf{sub}(X0, b, X2, X1).$
$\mathsf{transfo}(X0, X1) \leftarrow firstNthLetter(X0, a, 1, X2),\ \mathsf{sub}(X0, b, X2, X3),\ \mathsf{transfo}(X3, X1).$

Les exemples $e_1, \cdots, e_3$ permettent l'apprentissage de la première clause tandis que les autres concernent la deuxième clause.

Notre première méthode d'apprentissage posée, nous nous intéressons à ses limites.

6.3.7 Limites de l'approche guidée par un exemple

La première limite de notre approche provient de l'enrichissement d'une clause. Lors de l'enrichissement d'un exemple, la construction de la séquence d'atomes d'opération n'est réalisée qu'à partir d'un seul exemple. Ainsi, la séquence d'opération de transformation ne prend en compte que l'exemple en cours sans rechercher d'éventuel lien avec les autres exemples de l'ensemble. Une autre limite de cette approche provient de la dépendance à l'ordre des exemples. Ainsi, en fonction de l'exemple de départ, la généralisation apprise différera. Cette limitation peut être vue comme une conséquence de la première limitation. Nous illustrons ces points avec l'exemple suivant :

Exemple 65. *Reprenons l'ensemble des exemples de l'exemple 63 :*

$$\begin{array}{rcl} e_1 & = & (\ gérer\ ,\ nous\ gérons\ ) \\ e_2 & = & (\ régner\ ,\ nous\ régnons\ ) \end{array}$$

et ajoutons l'exemple :

$$e_3 \quad = \quad (\ jouer\ ,\ nous\ jouons\ )$$

Nous reprenons également les opérations de transformation et de contexte de l'exemple 63 et ajoutons une opération de transformation copy définie par :

$$copy(w, i, j, k) = w_0 \cdots w_k w_i \cdots w_j w_{k+1} \cdots w_n$$

avec $w = w_0 \cdots w_n \in \Sigma^\star$, $i \in \{0, \cdots, n\}$, $j \in \{i, \cdots, n\}$ *et* $k \in \{0, \cdots, n+1\}$ *et a pour signature :*

$$copy : \Sigma^\star \times \mathbb{N} \times \mathbb{N} \times \mathbb{N} \to \Sigma^\star$$

Cette opération copie un sous-mot $w_i, \cdots, w_j$ *de* w *et l'ajoute à la position* k *dans* w. *Remarquons que dans le cas où* $i = j$, *l'opération de copie est similaire à celle d'insertion dont l'alphabet serait restreint aux lettres présentes dans le mot* w. *Ces opérations sont donc parfois concurrentes.*

En fonction de l'ordre des exemples, les clauses de transformation apprises par l'algorithme 6.3 différeront. Dans le cas où l'exemple e_1 *est présenté en premier puis* e_2 *et* e_3, *l'unique clause de transformation apprise sera celle de l'exemple 63 :*

$$\begin{array}{rcl} \mathsf{transfo}(X0, X1) & \leftarrow & C_{X_0},\ \mathsf{ins}(X0,'\ ', 0, X2),\ C_{X_2}, \mathsf{ins}(X2, n, 0, X3),\ C_{X_3},\ \mathsf{ins}(X3, n, X4, X5), \\ & & C_{X_5},\ \mathsf{ins}(X5, o, 1, X6),\ C_{X_6},\ \mathsf{ins}(X6, s, 2, X7),\ C_{X_7},\ \mathsf{ins}(X7, u, 2, X8), \\ & & C_{X_8},\ \mathsf{sub}(X8, o, X9, X10),\ C_{X_{10}},\ \mathsf{sub}(X10, s, X11, X1). \end{array}$$

Tandis que si le premier exemple présenté est e_3, *la première clause de transformation apprise sera :*

$$\begin{array}{rcl} \mathsf{transfo}(t_0, t_5) & \leftarrow & C_{t_0},\ \mathsf{copy}(t_0, 1, 2, 0, t_1),\ C_{t_1},\ \mathsf{ins}(t_1,'\ ', 2, t_2), C_{t_2}, \mathsf{ins}(t_2, n, 0, t_3), \\ & & C_{t_3},\ \mathsf{ins}(t_3, n, 8, t_4),\ C_{t_4},\ \mathsf{ins}(t_4, s, 3, t_5),\ C_{t_5},\ \mathsf{sub}(t_5, o, 8, t_6),\ C_{t_6},\ \mathsf{sub}(t_6, s, 10, t_7). \end{array}$$

où :

$$\begin{array}{rcl} t_0 & = & cons(j, cons(o, cons(u, cons(e, cons(r, nil))))) \\ t_1 & = & cons(o, cons(u, cons(j, cons(o, cons(u, cons(e, cons(r, nil))))))) \\ t_2 & = & cons(o, cons(u, cons('\ ', cons(j, cons(o, cons(u, cons(e, cons(r, nil)))))))) \\ t_3 & = & cons(n, cons(o, cons(u, cons('\ ', cons(j, cons(o, cons(u, cons(e, cons(r, nil))))))))) \\ t_4 & = & cons(n, cons(o, cons(u, cons('\ ', cons(j, cons(o, cons(u, cons(e, cons(n, cons(r, nil)))))))))) \\ t_5 & = & cons(n, cons(o, cons(u, cons(s, cons('\ ', cons(j, cons(o, cons(u, cons(e, cons(n, cons(r, nil)))))))))))) \\ t_6 & = & cons(n, cons(o, cons(u, cons(s, cons('\ ', cons(j, cons(o, cons(u, cons(o, cons(n, cons(r, nil)))))))))))) \\ t_7 & = & cons(n, cons(o, cons(u, cons(s, cons('\ ', cons(j, cons(o, cons(u, cons(o, cons(n, cons(s, nil)))))))))))) \end{array}$$

Les contextes $C_{t_0}, \cdots, C_{t_{n-1}}$ sont ici omis pour plus de lisibilité. Cette clause de transformation ne couvre que l'exemple e_3 et n'a donc pas été généralisée avec les clauses suivantes. Cette hypothèse sera suivie de celle de l'exemple 63 couvrant les exemples e_1 et e_2.

Ce constat fait, nous proposons dans ce qui suit une approche destinée à éviter ce genre de comportement. Cette approche se base sur une recherche non plus guidée par un seul exemple mais par un tuple d'exemples. Ceci a pour objectif de trouver une transformation commune à plusieurs exemples et ainsi éviter la construction d'une clause de transformation dirigée par un unique exemple.

6.3.8 Recherche guidée par un tuple d'exemples

Nous avons précédemment vu avec l'approche proposée que la construction des atomes de transformation est parfois trop spécifique à l'exemple. Ceci pénalise la couverture de la clause. Ainsi, nous proposons d'effectuer la construction de ces atomes non plus à partir d'un seul élément de $\mathcal{X}$ mais d'un tuple d'éléments de $\mathcal{X}$. Cette construction sera toujours réalisée par exploration du graphe de recherche à l'aide de l'algorithme A^*. Ainsi, contrairement à la première approche, l'exploration du graphe est dirigée par l'ensemble des éléments du tuple. Il nous faut alors définir des opérations d'édition, une fonction de coût, des heuristiques ainsi que des distances d'édition adaptées au cas des tuples.

Opérations élémentaires d'édition

Nous notons $(\mathcal{X})^k$ l'ensemble des k-tuplets d'éléments de $\mathcal{X}$ défini formellement par :

$$(\mathcal{X})^k = \underbrace{\mathcal{X} \times \cdots \times \mathcal{X}}_{k \text{ fois}}$$

Nous désignons également par Δ_k l'ensemble des opérations d'édition définies sur $(\mathcal{X})^k$. Cet ensemble d'opérations est construit à partir d'un ensemble Δ d'opérations élémentaires définies sur $\mathcal{X}$. Une opération élémentaire $\delta_k \in \Delta_k$ est définie pour un élément $(x_1, \cdots, x_k) \in (\mathcal{X})^k$ si et seulement s'il existe une opération élémentaire $\delta \in \Delta$ telle que δ est définie pour les éléments $x_1, \cdots, x_k$ et $\delta_k((x_1, \cdots, x_k)) \in (\mathcal{X})^k$ où $\delta_k((x_1, \cdots, x_k)) = (\delta(x_1), \cdots, \delta(x_k))$. Enfin, nous notons S_{Δ_k} l'ensemble des scripts d'édition utilisant les opérations d'édition de Δ_k.

Nous nous intéressons maintenant à la construction d'un ensemble Δ_k et à la génération des voisins d'un terme. Ce dernier point est une opération nécessaire à l'exploration de notre graphe.

Exemple 66. *Soit l'opération élémentaire d'insertion :*

$$\mathsf{ins}_{i,a} : \mathcal{T}_{\Sigma^\star} \rightarrow \mathcal{T}_{\Sigma^\star}$$

avec $i \in \mathbb{N}$ et $a \in \Sigma$. Nous définissons, à partir de cette opération, l'opération élémentaire :

$$\mathsf{ins}_{i,a} : \mathcal{T}_{(\Sigma^*)^k} \rightarrow \mathcal{T}_{(\Sigma^*)^k}$$

Les paramètres a et i doivent être instanciés. La façon la plus simple consiste à les instancier statiquement. Dans le cas de l'opération ins*, une instanciation statique dépend des mots présents dans le tuple regardé. Dans le cas d'un tuple $(w_1, \cdots, w_k) \in (\Sigma^*)^k$, seules les opérations élémentaires statiquement paramétrées $\mathsf{ins}_{i,a}$ avec $i \in [0..(min(\{|w_1|, \cdots, |w_k|\})+1)]$ et $a \in \Sigma$ sont définies pour ce tuple. Illustrons cela avec, pour exemple, les couples $c_1 = (a, aa)$ et $c_2 = (ab, aab)$ construits à partir d'un alphabet $\Sigma = \{a, b\}$. Le passage du couple c_1 au*

couple c_2 est assuré par l'ajout de la lettre b à la fin de chaque mot du couple c_1. Il nous est cependant impossible à partir des opérations élémentaires paramètrées statiquement de représenter cette transformation puisque la taille des mots différe.

Ainsi, il nous est nécessaire, comme dans l'exemple 59, de prendre le contexte en compte.

L'exemple 66 met en avant le fait qu'il n'est plus possible de construire les atomes de transformation sans tenir compte du contexte. Le paramètrage statique des opérations limite en effet grandement leur utilisation.

Algorithme 24 Algorithme de génération des voisins : $succ_{\Delta_k}(t, T, M_T, \Theta)$

function $succ_{\Delta_k}(t, T, M_T, \Theta)$
1: **let** $t = (t_1, \cdots, t_k)$;
2: **let** $T = T_\Delta \cup T_C$;
3: **let** $M_T = M_\Delta \cup M_C$;
4: $G_\Delta := \mathsf{enhance}_{\mathsf{atom}}(t_1, M_\Delta, T_\Delta)$;
5: $G_C := \mathsf{enhance}_{\mathsf{atom}}(t_1, M_C, T_C)$;
6: $X := t_1$;
7: **for** $i = 2$ **to** k **do**
8: $\quad G'_\Delta := \mathsf{enhance}_{\mathsf{atom}}(t_i, M_\Delta, T_\Delta)$;
9: $\quad G'_C := \mathsf{enhance}_{\mathsf{atom}}(t_i, M_C, T_C)$;
10: $\quad G_\Delta := \mathsf{gen\_atoms\_set}(G_\Delta, G'_\Delta, \Theta)$;
11: $\quad G_C := \mathsf{gen\_atoms\_set}(G_C, G'_C, \Theta)$;
12: $\quad X := X'$ s.t. $X/t_i/X' \in \Theta$;
13: $\quad G_C := \mathsf{get\_consistent\_context}(G_C, M_C, \{X\})$
14: $\quad$ **if** $G_C = null$ **then**
15: $\quad\quad$ **return** $\emptyset$;
16: $\quad V := \mathsf{input}^{M_C}(G_C) \cup \mathsf{output}^{M_C}(G_C)$;
17: $\quad G_\Delta := G_\Delta \setminus \{\ell \in G_\Delta \mid \mathsf{input}^{M_\Delta}(\ell) \not\subseteq V\}$;
18: $\quad$ **if** $G_\Delta = \emptyset$ **then**
19: $\quad\quad$ **return** $\emptyset$;
20: **end for**
21: $succ := \emptyset$;
22: **let** $G_C = \ell^0_{in}, \cdots, \ell^{i_j}_{in}$ s.t. in is an input term of type $\mathcal{T}_\mathcal{X}$;
23: **for each** $\ell^\delta_{in} \in G_\Delta$ **do**
24: $\quad$ **let** out be the output term of ℓ^δ_{in} which type is $\mathcal{T}_\mathcal{X}$;
25: $\quad R := \ell^0_{in}, \cdots, \ell^{i_j}_{in}, \ell^\delta_{in}$. ;
26: $\quad succ := succ \cup \{((t_1, \cdots, t_k), R, (t'_1, \cdots, t'_k))\}$
$\quad\quad$ with t'_l be a term s.t. $((in = t_l) \wedge R \wedge T) \vdash ((in = t_l) \wedge (out = t'_l) \wedge R)$ for $l \in [1..k]$;
27: **end for**
28: **return** $succ$;
end function

Nous proposons ainsi la fonction $succ_{\Delta_k}$ présente à l'algorithme 24 pour la génération des voisins d'un terme. La génération des voisins d'un exemple $t = (t_1, \cdots, t_k)$ à partir d'une théorie de transformation T et de déclarations de mode M_T commence par la génération des atomes de transformation et contexte fondés pour l'élément t_1. Les atomes de transformation sont ensuite généralisés avec ceux du terme suivant. Il en est de même pour les atomes de contexte. La généralisation obtenue est ensuite élaguée de tout atome non respectueux des déclarations de mode. Ce processus est répété jusqu'à ce que la généralisation de tous

les termes soit réalisée. Nous disposons alors d'atomes de transformation et de contexte communs à chaque terme du tuple. L'association d'un atome de transformation avec tous les atomes de contexte permet la création d'une opération dynamiquement paramètrée. Celle-ci est utilisée pour générer un voisin. Nous disposons ainsi d'une méthode de génération des voisins tirant parti du contexte.

Cette méthode de génération des voisins est une brique nécessaire à l'implémentation de l'algorithme A^* destinée aux tuples. L'algorithme nécessitera également la définition d'heuristique et celle de coût propres aux tuples. Avant de nous attaquer à cela, nous présentons l'algorithme principal réalisant l'apprentissage de clauses de transformation.

Algorithme 25 Cet algorithme prend en entrée un ensemble d'exemples E défini à partir d'un ensemble $E_{\mathcal{X}}$, une théorie de transformation T et des déclarations de mode M_T pour le type $\mathcal{T}_{\mathcal{X}}$. Il retourne un ensemble de clauses de transformation.

function learn_tuple(E, T, M_T)
1: $H := \emptyset$;
2: **while** $E \neq \emptyset$ **do**
3: $E' := E$;
4: **while** $E' \neq \emptyset$ **do**
5: $E' := E' \setminus \{\mathsf{transfo}(e_j, s_j)\}$;
6: **let** $seq_{in} = (e_1, \cdots, e_k)$ **and** $seq_{out} = (s_1, \cdots, s_k)$ with $0 \leqslant k$;
7: $seq_{in} := (e_1, \cdots, e_k, e_j)$;
8: $seq_{out} := (s_1, \cdots, s_k, s_j)$;
9: $h := A^*_{\mathsf{transfo}}(seq_{in}, seq_{out})$;
10: **if** $h = null$ **then**
11: $seq_{in} := (e_1, \cdots, e_k)$;
12: $seq_{out} := (s_1, \cdots, s_k)$;
13: **if** $|seq_{in}| = 0$ **then return** null ;
14: **end if**
15: **end while**
16: $h := \mathsf{prune}(h, E, T, M_T)$;
17: **for each** $e \in E$ **do**
18: **if** $h \wedge T \vdash e$ **then** $E := E \setminus \{e\}$;
19: **end for**
20: $H := H \cup \{h\}$;
21: **end while**
22: **return** H ;
end function

Cet algorithme apprend à partir d'une séquence d'éléments de $\mathcal{X}$ une clause de transformation. Il se décompose en trois parties :

1. la construction d'une clause de transformation avec un algorithme A^*_{transfo} ;
2. l'élagage de la clause de transformation apprise ;
3. la suppression des exemples couverts de E par la clause de transformation apprise.

Ces deux derniers points sont similaires à ceux de l'algorithme 6.3. Nous procédons maintenant à la définition du coût ainsi que d'heuristiques.

Fonctions de coût et A^*

Avant de définir des fonctions de coûts pour l'ensemble $(\mathcal{X})^k$, rappelons que les fonctions cost_δ avec $\delta \in \Delta$ définies pour l'ensemble $\mathcal{X}$ respectent la propriété suivante :

$$\forall \delta \in \Delta \ \forall x \in \mathcal{X},\ \mathsf{cost}_\delta(x) \geqslant \mathsf{d}(x, \delta(x))$$

On définit maintenant plusieurs fonctions de coût sur $(\mathcal{X})^k$ pour une opération d'édition telles que $\forall \delta \in \Delta\ \forall (x_1, \cdots, x_k) \in (\mathcal{X})^k\ \exists \delta_k \in \Delta_k$ pour laquelle on a :

— $\mathsf{sum\text{-}cost}_{\delta_k}((x_1, \cdots, x_k)) = \sum_{i=1}^{k} \mathsf{cost}_\delta(x_i)$;
— $\mathsf{min\text{-}cost}_{\delta_k}((x_1, \cdots, x_k)) = \min(\mathsf{cost}_\delta(x_1), \cdots, \mathsf{cost}_\delta(x_k))$;
— $\mathsf{max\text{-}cost}_{\delta_k}((x_1, \cdots, x_k)) = \max(\mathsf{cost}_\delta(x_1), \cdots, \mathsf{cost}_\delta(x_k))$;

De la même manière, on définit les extensions de la distance d'édition suivantes telles que $\forall (x_1, \cdots, x_k), (y_1, \cdots, y_k) \in (\mathcal{X})^k$, on a :

— $\mathsf{sum\text{-}d}((x_1, \cdots, x_k), (y_1, \cdots, y_k)) = \sum_{i=1}^{k} \mathsf{d}(x_i, y_i)$;
— $\mathsf{min\text{-}d}((x_1, \cdots, x_k), (y_1, \cdots, y_k)) = \min(\mathsf{d}(x_1, y_1), \cdots, \mathsf{d}(x_k, y_k))$;
— $\mathsf{max\text{-}d}((x_1, \cdots, x_k), (y_1, \cdots, y_k)) = \max(\mathsf{d}(x_1, y_1), \cdots, \mathsf{d}(x_k, y_k))$;

On finit par déterminer le coût d'un script d'édition $S = \delta_{k_1}, \cdots, \delta_{k_n}$:

— $\forall x \in (\mathcal{X})^k\ \mathsf{sum\text{-}cost}_S(x) = \mathsf{sum\text{-}cost}_{\delta_{k_1}}(x) + \cdots + \mathsf{sum\text{-}cost}_{\delta_{k_n}}(\delta_{k_1} \circ \cdots \circ \delta_{k_{n-1}}(x))$;
— $\forall x \in (\mathcal{X})^k\ \mathsf{min\text{-}cost}_S(x) = \mathsf{min\text{-}cost}_{\delta_{k_1}}(x) + \cdots + \mathsf{min\text{-}cost}_{\delta_{k_n}}(\delta_{k_1} \circ \cdots \circ \delta_{k_{n-1}}(x))$;
— $\forall x \in (\mathcal{X})^k\ \mathsf{max\text{-}cost}_S(x) = \mathsf{max\text{-}cost}_{\delta_{k_1}}(x) + \cdots + \mathsf{max\text{-}cost}_{\delta_{k_n}}(\delta_{k_1} \circ \cdots \circ \delta_{k_{n-1}}(x))$;

Nous disposons maintenant d'heuristiques pour les tuples et allons établir quelques propriétés portant sur ces dernières. Nous commençons par vérifier que l'inégalité triangulaire est toujours vraie pour $\mathsf{sum\text{-}d}, \mathsf{min\text{-}d}$ et $\mathsf{max\text{-}d}$ sur $(\mathcal{X})^k$.

Lemme 31. $\forall (x_1, \cdots, x_k), (y_1, \cdots, y_k), (t_1, \cdots, t_k) \in (\mathcal{X})^k.\ \mathsf{sum\text{-}d}((x_1, \cdots, x_k), (t_1, \cdots, t_k)) \leqslant \mathsf{sum\text{-}d}((x_1, \cdots, x_k), (y_1, \cdots, y_k)) + \mathsf{sum\text{-}d}((y_1, \cdots, y_k), (t_1, \cdots, t_k))$

Preuve. Soit $(x_1, \cdots, x_k), (y_1, \cdots, y_k), (t_1, \cdots, t_k) \in (\mathcal{X})^k$, on sait que les inégalités $\mathsf{d}(x_1, t_1) \leqslant \mathsf{d}(x_1, y_1) + \mathsf{d}(y_1, t_1), \cdots, \mathsf{d}(x_k, t_k) \leqslant \mathsf{d}(x_k, y_k) + \mathsf{d}(y_k, t_k)$ sont vérifiées puisque l'inégalité triangulaire pour la distance d'édition est vérifiée. Pour $\mathsf{sum\text{-}d}$, on peut donc construire l'inégalité suivante :

$$\mathsf{d}(x_1, t_1) + \cdots + \mathsf{d}(x_k, t_k) \leqslant \mathsf{d}(x_1, y_1) + \mathsf{d}(y_1, t_1) + \cdots + \mathsf{d}(x_k, y_k) + \mathsf{d}(y_k, t_k)$$

$$\mathsf{d}(x_1, t_1) + \cdots + \mathsf{d}(x_k, t_k) \leqslant \mathsf{d}(x_1, y_1) + \cdots + \mathsf{d}(x_k, y_k) + \cdots + \mathsf{d}(y_1, t_1) + \cdots + \mathsf{d}(y_k, t_k)$$

or :

$$\mathsf{sum\text{-}d}((x_1, \cdots, x_k), (y_1, \cdots, y_k)) = \sum_{i=1}^{k} \mathsf{d}(x_i, y_i)$$

d'où :

$$\mathsf{sum\text{-}d}((x_1, \cdots, x_k), (t_1, \cdots, t_k)) \leqslant \mathsf{sum\text{-}d}((x_1, \cdots, x_k), (y_1, \cdots, y_k)) + \mathsf{sum\text{-}d}((y_1, \cdots, y_k), (t_1, \cdots, t_k))$$

L'inégalité triangulaire est donc vérifiée pour $\mathsf{sum\text{-}d}$ sur $(\mathcal{X})^k$. □

Lemme 32. $\forall (x_1, \cdots, x_k), (y_1, \cdots, y_k), (t_1, \cdots, t_k) \in (\mathcal{X})^k.\ \mathsf{max\text{-}d}((x_1, \cdots, x_k), (t_1, \cdots, t_k)) \leqslant \mathsf{max\text{-}d}((x_1, \cdots, x_k), (y_1, \cdots, y_k)) + \mathsf{max\text{-}d}((y_1, \cdots, y_k), (t_1, \cdots, t_k))$;

Preuve. Soit $(x_1, \cdots, x_k), (y_1, \cdots, y_k), (t_1, \cdots, t_k) \in (\mathcal{X})^k$, on sait que les inégalités $\mathsf{d}(x_1, t_1) \leqslant \mathsf{d}(x_1, y_1) + \mathsf{d}(y_1, t_1), \cdots, \mathsf{d}(x_k, t_k) \leqslant \mathsf{d}(x_k, y_k) + \mathsf{d}(y_k, t_k)$ sont vérifiées puisque l'inégalité triangulaire pour une distance d'édition est vérifiée. Pour max-d, on peut donc construire l'inégalité suivante :

$$\max(\mathsf{d}(x_1, t_1), \cdots, \mathsf{d}(x_k, t_k)) \leqslant \max(\mathsf{d}(x_1, y_1), \cdots, \mathsf{d}(x_k, y_k)) + \max(\mathsf{d}(y_1, t_1), \cdots, \mathsf{d}(y_k, t_k))$$

or :

$$\mathsf{max\text{-}d}((x_1, \cdots, x_k), (y_1, \cdots, y_k)) = \max(\mathsf{d}(x_1, y_1), \cdots, \mathsf{d}(x_k, y_k))$$

d'où :

$$\mathsf{max\text{-}d}((x_1, \cdots, x_k), (t_1, \cdots, t_k)) \leqslant \mathsf{max\text{-}d}((x_1, \cdots, x_k), (y_1, \cdots, y_k)) + \mathsf{max\text{-}d}((y_1, \cdots, y_k), (t_1, \cdots, t_k))$$

L'inégalité triangulaire est donc vérifiée pour max-d sur $(\mathcal{X})^k$. □

Lemme 33. $\forall (x_1, \cdots, x_k), (y_1, \cdots, y_k), (t_1, \cdots, t_k) \in (\mathcal{X})^k.\ \mathsf{min\text{-}d}((x_1, \cdots, x_k), (t_1, \cdots, t_k)) \not\leqslant \mathsf{min\text{-}d}((x_1, \cdots, x_k), (y_1, \cdots, y_k)) + \mathsf{min\text{-}d}((y_1, \cdots, y_k), (t_1, \cdots, t_k))$;

Preuve. Soit $(x_1, \cdots, x_k), (y_1, \cdots, y_k), (t_1, \cdots, t_k) \in (\mathcal{X})^k$, on sait que les inégalités $\mathsf{d}(x_1, t_1) \leqslant \mathsf{d}(x_1, y_1) + \mathsf{d}(y_1, t_1), \cdots, \mathsf{d}(x_k, t_k) \leqslant \mathsf{d}(x_k, y_k) + \mathsf{d}(y_k, t_k)$ sont vérifiées puisque l'inégalité triangulaire pour une distance d'édition est vérifiée. Pour min-d, on suppose que $x_1 = y_1$, $y_2 = t_2$ et $\forall i \in [1..k], x_i \neq t_i$. On a donc $\mathsf{min\text{-}d}((x_1, \cdots, x_k), (y_1, \cdots, y_k)) = 0$ car $\mathsf{d}(x_1, y_1) = 0$, $\mathsf{min\text{-}d}((y_1, \cdots, y_k), (t_1, \cdots, t_k)) = 0$ car $\mathsf{d}(y_2, t_2) = 0$ et $\mathsf{d}((x_1, \cdots, x_k), (t_1, .., t_k)) \neq 0$ car $\forall i \in [1..k], x_i \neq t_i$. Ainsi :

$$\min(x_1, t_1), \cdots, \mathsf{d}(x_k, t_k)) \not\leqslant \min(\mathsf{d}(x_1, y_1), \cdots, \mathsf{d}(x_k, y_k)) + \min(\mathsf{d}(y_1, t_1), \cdots, \mathsf{d}(y_k, t_k))$$

$$\mathsf{min\text{-}d}((x_1, \cdots, x_k), (t_1, \cdots, t_k)) \not\leqslant \mathsf{min\text{-}d}((x_1, \cdots, x_k), (y_1, \cdots, y_k)) + \mathsf{min\text{-}d}((y_1, \cdots, y_k), (t_1, \cdots, t_k))$$

L'inégalité triangulaire pour min-d n'est donc pas vérifiée sur $(\mathcal{X})^k$. □

L'inégalité triangulaire est vérifiée sur $(\mathcal{X})^k$ pour les fonctions sum-d et max-d contrairement à min-d. Elle permet d'affirmer que le chemin le plus court entre deux éléments est le chemin contenant le moins d'opérations d'édition. Cette propriété est donc importante pour effectuer l'exploration. Nous nous intéressons donc aux fonctions sum-d et max-d uniquement et établissons maintenant l'admissibilité de ces fonctions.

Lemme 34. $\forall x \in (\mathcal{X})^k\ \forall S \in S_\Delta.\ \mathsf{sum\text{-}cost}_S(x) \geqslant \mathsf{sum\text{-}d}(x, S(x))$

Preuve. Soit un ensemble d'opérations d'édition Δ définies sur $\mathcal{X}$ et son ensemble Δ_k d'opérations élémentaires définies sur $(\mathcal{X})^k$. On veut montrer que : $\forall x \in (\mathcal{X})^k\ \forall S \in S_\Delta.\ \mathsf{sum\text{-}cost}_S(x) \geqslant \mathsf{sum\text{-}d}(x, S(x))$ avec $S = \delta_{k_1}, \cdots, \delta_{k_n}$. On sait que :

$$\forall \delta \in \Delta\ \forall (x_1, \cdots, x_n) \in (\mathcal{X})^k\ \exists \delta_k \in \Delta_k,\ \mathsf{sum\text{-}cost}_{\delta_k}((x_1, \cdots, x_k)) = \sum_{i=1}^{k} \mathsf{cost}_\delta(x_i)$$

et que : $\forall x \in \Sigma^*\ \forall \delta \in \Delta,\ \mathsf{cost}_\delta(x) \geqslant \mathsf{d}(x, \delta(x))$ d'où :

$$\forall \delta_k \in \Delta_k \forall (x_1, \cdots, x_n) \in (\mathcal{X})^k,\ \mathsf{sum\text{-}cost}_{\delta_k}((x_1, \cdots, x_k)) \geqslant \sum_{i=1}^{k} \mathsf{d}(x_i, \delta_k(x_i))$$

puis :

$$\forall \delta_k \in \Delta_k \forall (x_1, \cdots, x_n) \in (\mathcal{X})^k,\ \mathsf{sum\text{-}cost}_{\delta_k}((x_1, \cdots, x_k)) \geqslant \mathsf{sum\text{-}d}((x_1, \cdots, x_k), \delta_k((x_1, \cdots, x_k)))$$

On peut donc transformer l'égalité suivante :

$$\forall x \in (\mathcal{X})^k \ \mathsf{sum\text{-}cost}_S(x) = \mathsf{sum\text{-}cost}_{\delta_{k_1}}(x) + \cdots + \mathsf{sum\text{-}cost}_{\delta_{k_n}}(\delta_{k_1} \circ \cdots \circ \delta_{k_{n-1}}(x))$$

en l'inégalité suivante :

$$\forall x \in (\mathcal{X})^k \ \mathsf{sum\text{-}cost}_S(x) \geqslant \mathsf{sum\text{-}d}(x, \delta_{k_1}(x)) + \cdots + \mathsf{sum\text{-}d}(\delta_{k_1} \circ \cdots \circ \delta_{k_{n-1}}(x), S(x))$$

puis grâce à l'inégalité triangulaire :

$$\mathsf{sum\text{-}cost}_S(x) \geqslant \mathsf{sum\text{-}d}(x, \delta_{k_1} \circ \cdots \circ \delta_{k_m}(x))$$
$$\mathsf{sum\text{-}cost}_S(x) \geqslant \mathsf{sum\text{-}d}(x, S(x))$$

□

Lemme 35. $\forall x \in (\mathcal{X})^k \ \forall S \in S_\Delta.\ \mathsf{max\text{-}cost}_S(x) \geqslant \mathsf{max\text{-}d}(x, S(x))$

Preuve. Soit un ensemble d'opérations d'édition Δ définies sur $\mathcal{X}$ et son ensemble Δ_k d'opérations élémentaires définies sur $(\mathcal{X})^k$. On veut montrer que :

$$\forall x \in (\mathcal{X})^k \ \forall S \in S_\Delta.\ \mathsf{max\text{-}cost}_S(x) \geqslant \mathsf{max\text{-}d}(x, S(x))$$

avec $S = \delta_{k_1} \cdots \delta_{k_m}$. On sait que :

$\forall \delta \in \Delta \, \forall (x_1, \cdots, x_n) \in (\mathcal{X})^k \, \exists \delta_k \in \Delta_k,\ \mathsf{max\text{-}cost}_{\delta_k}((x_1, \cdots, x_k)) = \max(\mathsf{cost}_\delta(x_1), \cdots, \mathsf{cost}_\delta(x_k))$

et que $\forall x \in \Sigma^* \ \forall \delta \in \Delta,\ \mathsf{cost}_\delta(x) \geqslant \mathsf{d}(x, \delta(x))$ d'où :

$\forall \delta_k \in \Delta_k \forall (x_1, \cdots, x_n) \in (\mathcal{X})^k,\ \mathsf{max\text{-}cost}_{\delta_k}((x_1, \cdots, x_k)) \geqslant \max(\mathsf{cost}_\delta(x_1), \cdots, \mathsf{cost}_\delta(x_k))$

$\forall \delta_k \in \Delta_k \forall (x_1, \cdots, x_n) \in (\mathcal{X})^k,\ \mathsf{max\text{-}cost}_{\delta_k}((x_1, \cdots, x_k)) \geqslant \mathsf{max\text{-}d}((x_1, \cdots, x_k), \delta_k(x_1, \cdots, x_k))$

On peut donc transformer l'égalité suivante :

$$\forall x \in (\mathcal{X})^k \ \mathsf{max\text{-}cost}_S(x) = \mathsf{max\text{-}cost}_{\delta_{k_1}}(x) + \cdots + \mathsf{max\text{-}cost}_{\delta_{k_n}}(\delta_{k_1} \circ \cdots \circ \delta_{k_{n-1}}(x))$$

en l'inégalité suivante :

$$\forall x \in (\mathcal{X})^k \ \mathsf{max\text{-}cost}_S(x) \geqslant \mathsf{max\text{-}d}(x, \delta_{k_1}(x)) + \cdots + \mathsf{max\text{-}d}(\delta_{k_1} \circ \cdots \circ \delta_{k_{n-1}}(x), \delta_{k_1} \circ \cdots \circ \delta_{k_n}(x))$$

$$\forall x \in (\mathcal{X})^k \ \mathsf{max\text{-}cost}_S(x) \geqslant \mathsf{max\text{-}d}(x, \delta_{k_1}(x)) + \cdots + \mathsf{max\text{-}d}(\delta_{k_1} \circ \cdots \circ \delta_{k_{n-1}}(x), S(x))$$

puis grâce à l'inégalité triangulaire :

$$\mathsf{max\text{-}cost}_S(x) \geqslant \mathsf{max\text{-}d}(x, S(x))$$

□

On s'intéresse maintenant à la monotonie des fonctions sum-d et max-d.

Lemme 36. $\forall x, t \in (\mathcal{X})^k \ \forall S \in S_\delta.\ \mathsf{sum\text{-}d}(x, t) - \mathsf{sum\text{-}d}(S(x), t) \leqslant \mathsf{sum\text{-}cost}_S(x)$

Preuve. On veut montrer : $\forall x, t \in (\mathcal{X})^k \ \forall S \in S_\delta.\ \mathsf{sum\text{-}d}(x, t) - \mathsf{sum\text{-}d}(S(x), t) \leqslant \mathsf{sum\text{-}cost}_S(x)$
On sait que la distance sum-d vérifie l'inégalité triangulaire d'où :

$$\mathsf{sum\text{-}d}(x, t) \leqslant \mathsf{sum\text{-}d}(x, S(x)) + \mathsf{sum\text{-}d}(S(x), t)$$
$$\mathsf{sum\text{-}d}(x, t) - \mathsf{sum\text{-}d}(S(x), t) \leqslant \mathsf{sum\text{-}d}(x, S(x))$$

Or $\forall x \in (\mathcal{X})^k \ \forall S \in S_\Delta.\ \mathsf{sum\text{-}cost}_S(x) \geqslant \mathsf{sum\text{-}d}(x, S(x))$:

$$\mathsf{sum\text{-}d}(x, t) - \mathsf{sum\text{-}d}(S(x), t) \leqslant \mathsf{sum\text{-}cost}_S(x)$$

□

Lemme 37. $\forall x, t \in (\mathcal{X})^k \ \forall S \in S_\delta.\ \mathsf{max\text{-}d}(x,t) - \mathsf{max\text{-}d}(S(x),t) \leqslant \mathsf{max\text{-}cost}_S(x)$

Preuve. On veut montrer : $\forall x, t \in (\mathcal{X})^k \ \forall S \in S_\delta.\ \mathsf{max\text{-}d}(x,t) - \mathsf{max\text{-}d}(S(x),t) \leqslant \mathsf{max\text{-}cost}_S(x)$
On sait que la distance max-d vérifie l'inégalité triangulaire d'où :

$$\mathsf{max\text{-}d}(x,t) \leqslant \mathsf{max\text{-}d}(x,S(x)) + \mathsf{max\text{-}d}(S(x),t)$$

$$\mathsf{max\text{-}d}(x,t) - \mathsf{max\text{-}d}(S(x),t) \leqslant \mathsf{max\text{-}d}(x,S(x))$$

Or $\forall x \in (\mathcal{X})^k \ \forall S \in S_\Delta.\ \mathsf{max\text{-}cost}_S(x) \geqslant \mathsf{max\text{-}d}(x,S(x))$:

$$\mathsf{max\text{-}d}(x,t) - \mathsf{max\text{-}d}(S(x),t) \leqslant \mathsf{max\text{-}cost}_S(x)$$

□

Nous savons donc grâce aux lemmes 34 et 36 que l'emploi de l'algorithme A^* sur un graphe de transformation $G = (N, E, \Delta_k)$ où $N = (\mathcal{X})^k$, E et Δ_k avec pour fonction d'heuristique sum-d et pour fonction de coût sum-cost est efficace pour la recherche d'un plus court chemin entre deux points de $(\mathcal{X})^k$. Le résultat similaire est vérifié à l'aide des lemmes 35 et 37 pour la fonction d'heuristique max-d et pour la fonction de coût max-cost. Nous disposons ainsi d'un algorithme efficace, théoriquement parlant, de recherche commun à plusieurs couples d'éléments de $\mathcal{X}$.

La recherche guidée par un tuple présente cependant un inconvénient. En effet, dans le cas de la recherche dirigée par un seul élément, nous savons que l'inclusion de l'ensemble Δ_d des opérations d'édition propres à la distance d'édition d nous assure de trouver un plus court chemin. Cependant, dans le cas d'un tuple, cela n'est plus vrai puisque nous ne sommes pas sûrs de l'existence d'une transformation commune à l'ensemble des éléments du tuple. Il est donc nécessaire de prendre en compte un critère d'arrêt pour la recherche. Nous faisons le choix de ne pas considérer un nœud $t = (t_1, \cdots, t_k)$ lors de la recherche d'un chemin entre les tuples $e = (e_1, \cdots, e_k)$ et $s = (s_1, \cdots, s_k)$ si pour un élément t_i avec $i \in [1..k]$, on a $f(t_i) > f(s_i)$ où f est l'estimation du coût de la distance à parcourir auquel est ajouté le coût de la distance parcourue. Il s'agit de la fonction f définie pour l'algorithme A^* d'un élément vers un autre. Elle utilise la même l'heuristique que celle utilisée pour définir sum-d ou max-d. Nous justifions cela par l'admissibilité et la monotonicité de la distance d'édition d sur l'ensemble $\mathcal{X}$ qui est utilisée pour la construction de sum-d et max-d.

Nous testons expérimentalement dans la section suivante nos algorithmes d'apprentissage.

6.4 Expérimentations

Lors de ce chapitre, plusieurs algorithmes ont été proposés afin de permettre l'apprentissage de clauses de transformation. Ces clauses permettent la représentation de scripts d'édition associés à des contextes et ont pour but de représenter des transformations d'éléments à éléments. Nous procédons maintenant à des expériences visant à apprendre des transformation d'éléments à éléments à l'aide de ce type de représentation. Les éléments envisagés seront, comme mentionné précédemment, des arbres et des mots. Elles ont pour but d'étudier l'efficacité d'un apprentissage de cette nature en utilisant nos représentations clausales et nos algorithmes. Nous utilisons les critères d'évaluation définis lors du chapitre 3 à la section 3.5 que sont la F-mesure, le rappel et la précision. Les mots utilisés pour nos expériences sont issus d'un corpus portant sur la conjugaison de verbes en français tandis que les documents XML proviennent du corpus footballistique présenté lors des expériences de la section 5.6 du chapitre 5. Passons maintenant à leur présentation.

6.4.1 Corpus de documents

Les expériences présentées portent sur deux corpus différents. Le premier a pour thématique la conjugaison au présent de l'indicatif de verbes en français à la première personne du singulier. Le second traite de résultats footballistiques et comporte les documents XML du corpus artificiel présenté lors du chapitre 3.

Afin d'en réaliser l'apprentissage, chaque corpus est divisé par validation croisée 5 fois. Il y a donc cinq apprentissages à effectuer par corpus, chacun sur 80% des données. Les 20% restants sont utilisés en test. Nous répétons ces opérations 10 fois et faisons la moyenne des résultats obtenus. Plus de détails sur la validation croisée se trouvent dans le chapitre 3 à la section 3.5. Nous présentons maintenant ces différents corpus.

Conjugaison de verbes

Notre premier corpus porte sur la conjugaison de verbes en français. Il contient un ensemble de couples dont le premier élément est un verbe à l'infinitif et le second ce même verbe conjugué à la première personne du présent de l'indicatif. Ce corpus contient 3 classes. Chaque classe correspond à un groupe de conjugaison propre à la langue française que sont le 1^{er}, $2^{ième}$ et $3^{ième}$ groupe.

Les verbes du 1^{er} groupe possèdent un infinitif qui se termine en *-er*. La conjugaison de ces verbes est régulière mais certains présentent des modifications orthographiques et phonétiques. En fonction de cela, plusieurs règles de conjugaison existent afin de prendre en compte ces modifications. La classe de notre corpus correspondant à ce groupe contient 62 verbes dont la longueur moyenne est de 7.8 lettres. Ils respectent 7 règles de conjugaison différentes afin de permettre la conjugaison de verbes comme :

posséder	,	je possède
opérer	,	j'opère
profiter	,	je profite

Les verbes du 2^{nd} groupe ont un infinitif terminant en *-ir*. Leur conjugaison est régulière et ils ont un seul radical. Seule la forme du pronom personnel est adapté en fonction du verbe à conjuguer. La classe de notre corpus correspondant à ce groupe contient 14 verbes dont la longueur moyenne est de 7.6 lettres. Ils respectent 2 règles de conjugaison différentes précisant le choix du pronom en fonction du verbe à conjuguer à la manière de :

finir	,	je finis
affranchir	,	j'affranchis

Enfin, les verbes du $3^{ème}$ groupe ont un infinitif pouvant se terminer en *-re*, *-ir*, *-oir* ou *-oire*. Le troisième groupe contient également le verbe *aller*. La classe de notre corpus correspondant à ce groupe contient 249 verbes dont la longueur moyenne est de 8.9 lettres. Ils respectent 55 règles de conjugaison différentes. Le nombre moyen d'opérations d'édition nécessaires pour transformer le premier élément d'un couple en le second est 5 toutes classes confondues.

Le but de l'apprentissage portant sur ce corpus est de retrouver des règles de conjugaison sous la forme de clauses de transformation. Ces clauses doivent être capables de conjuguer les verbes de ce corpus à la première personne du présent de l'indicatif.

Données footballistiques

Le corpus de documents que nous allons utiliser pour les transformations d'arbres se base sur les documents des données footballistiques présentées au chapitre 3. Rappelons

que ces données constituent un corpus sur les résultats sportifs d'équipes françaises de football. Y sont décrits les différentes rencontres ainsi que les résultats sportifs associés aux championnats. Ce corpus de documents a été généré artificiellement à partir d'une base de données. Il contient au total 170 pages XHTML. Chaque page est associée à une unique classe qui correspond à une manière de représenter des informations dans une structure de documents particulière. Les différentes classes, que sont L_0, ..., L_7 et L_8, contiennent ainsi les mêmes données représentées différemment. On a ainsi plusieurs classes contenant des documents caractérisés par leur structure. Ces documents sont repartis en 9 classes, chaque classe contient en moyenne 19 documents. Un arbre représentant l'un de ces documents contient en moyenne 296 nœuds, a une profondeur de 7 ainsi qu'une largeur de 6. La nombre moyen de nœuds d'un document de ce corpus est de 300.

En plus de leur structure, il est possible de regrouper les classes en fonction des données à partir desquelles elles ont été générées. On a ainsi un premier groupe contenant les classes L_0 et L_3 et un second contenant les classes L_1, L_2, L_4, L_5, L_6, L_7 et L_8.

Chaque document d'une classe d'un de ces groupes dispose ainsi, dans une autre classe de ce même groupe, d'un document homologue portant sur les mêmes données. Il existe donc une bijection entre les documents de deux classes d'un même groupe. Elle nous permet d'établir un ensemble de couples d'éléments avant et après transformation. Pour deux classes L_i et L_j d'un même groupe, nous disposons d'un ensemble de couples d'éléments, noté $L_{i,j}$, suivant cette bijection. Notre corpus footballistique pour la transformation contient ainsi les classes suivantes : $L_{0,3}$, $L_{3,0}$, $L_{1,2}$, $L_{1,4}$, $L_{1,5}$, ..., $L_{8,6}$, et $L_{8,7}$ soit un total de 44 classes regroupant 520 couples d'éléments. Le nombre moyen d'opérations d'édition nécessaires pour transformer le premier élément d'un couple en le second est 203.

Le but de l'apprentissage portant sur ce corpus est alors de retrouver pour chaque classe $L_{i,j}$ l'ensemble des clauses de transformation assurant le passage du premier élément au second.

6.4.2 Résultats pour les mots

Donnons maintenant les résultats d'apprentissage pour notre corpus de conjugaison. Ces résultats permettent de quantifier l'efficacité de notre approche. Nous présentons ci-dessous différentes tables récapitulant les résultats de rappel, précision et F-mesure obtenus sur ce corpus de documents. Sont indiqués la taille moyenne d'une hypothèse apprise, son nombre d'atomes, les mesures de rappel, de précision et de F-mesure ainsi que le temps d'exécution. Ces résultats sont des moyennes obtenues tous runs confondus.

Exploration guidée par un exemple

La table 6.1 présente les résultats obtenus pour l'approche dirigée par un exemple sur le corpus de conjugaison. Y sont spécifiés les scores associés aux différents groupes de verbes. Ces mesures traduisent les difficultés qu'a eu notre algorithme à apprendre des règles de transformation à partir de ces groupes. L'apprentissage de clauses de transformation pour les deux premiers groupes ne semble pas avoir posé de réels problèmes. En effet, le rappel, la précision et la F-mesure associés à chacun de ces groupes sont égaux ou supérieurs à 70%. Les mesures obtenues pour le $2^{\text{ième}}$ groupe ne sont pas étonnantes puisque ce groupe possède les règles de conjugaison les plus simples. Les règles de conjugaison du 1^{ier} groupe s'avèrent, quant-à-elles, légèrement plus complexes avec notamment des exemples possédant des modifications d'accents. La précision des hypothèses apprises pour le $2^{\text{ième}}$ groupe est environ 10% inférieure à celle du 1^{ier} groupe. Ainsi, malgré une taille moyenne de clauses apprises pour le 1^{ier} groupe 3 fois supérieure à celle du $2^{\text{ième}}$ groupe, les clauses apprises

autorisent davantage de transformations non autorisées.

	1er groupe	2ième groupe	3ième groupe
Temps d'exécution (j :h :m :s :ms)	2 :24 :255	1 :22 :949	1 :19 :32 :252
Rappel	0,760	0,767	0,566
Précision	0,681	0,900	0,695
F-mesure	0,705	0,776	0,619
Nombre total d'atomes par clause	325,2	92,6	704,7
Nombre total d'hypothèses par classe	6	2	35,7

TABLE 6.1 – Résultats d'apprentissage de l'approche guidée par un exemple pour le corpus de conjugaison

La F-mesure du dernier groupe est légèrement moins élevée avec une valeur dépassant les 60%. Ce score n'est pas surprenant puisque de nombreuses règles du 3ième groupe sont spécifiques à 2 ou 3 verbes tout au plus. Ainsi, notre processus de généralisation nécessitant au moins deux exemples s'avère inefficace si un seul de ces verbes couverts par ce genre de règles est présent dans l'ensemble d'apprentissage. Remarquons que la mesure de rappel est proche de 50% ce qui confirme notre propos.

L'apprentissage réalisé à l'aide de notre algorithme basé sur une exploration guidée par un exemple permet d'obtenir des résultats corrects vis-à-vis de ce corpus. Nous nous intéressons maintenant à notre algorithme proposant une approche cette fois guidée par un tuple d'exemples.

Exploration guidée par un tuple d'exemples

La table 6.2 présente les résultats obtenus par notre algorithme basé sur une exploration dirigée par un tuple d'exemples. Ces résultats avoisinent ceux obtenus par une exploration dirigée par un exemple. Plus précisément, les scores de F-mesure, rappel et précision sont, pour le 1ier et le 3ième groupe, légèrement supérieurs à ceux d'une exploration guidée par un exemple. Ceci n'est cependant pas le cas pour le 2nd groupe dont la F-mesure est environ 20% moins élevée.

Nous expliquons cette baisse du score par la manière dont est généré le voisinage d'un nœud. En effet, contrairement à notre première approche, l'ensemble des opérations d'édition autorisées n'est plus propre à un élément mais à chaque élément du tuple. Elles sont construites par généralisation des opérations d'édition de chaque élément d'un tuple. Ainsi, une opération d'édition peut être paramétrée dynamiquement ou statiquement et dépendra ainsi du contexte propre à ces éléments. Suivant cela, plusieurs manières d'obtenir un même voisin peuvent co-exister. Un choix arbitraire est alors nécessaire. Celui-ci modifiera le comportement d'une hypothèse vis-à-vis de l'ensemble de test.

	1er groupe	2ième groupe	3ième groupe
Temps d'exécution (j :h :m :s :ms)	48 :39 :147	3 :12 :970	10 :06 :12 :329
Rappel	0,763	0,553	0,615
Précision	0,685	0,767	0,699
F-mesure	0,709	0,589	0,654
Nombre total d'atomes par clause	273,8	691,9	651,3
Nombre total d'hypothèses par classe	6,8	2	39,6

TABLE 6.2 – Résultats d'apprentissage de l'approche guidée par un tuple d'exemples pour le corpus de conjugaison

Terminons en commentant le temps d'exécution nécessaire à cette tâche. Celui-ci est de 3 à 24 fois supérieur par rapport à l'apprentissage dirigé par un exemple. Ce point n'est pas négligeable au vu des scores obtenus. Ainsi, l'approche dirigée par un exemple semble disposer d'une supériorité flagrante à ce niveau. Il convient donc de l'utiliser en premier lieu.

6.4.3 Résultats pour les arbres

Les expérimentations portant sur les arbres sont moins abouties que celles présentées précédemment pour les mots. En effet, à cause de la complexité de la mesure d'édition utilisée, nous ne sommes pas en capacité de présenter des mesures pour l'apprentissage d'arbres à partir des documents de notre corpus footballistique. Ainsi, malgré la certitude d'effectuer une recherche optimale dans un graphe de transformation, l'emploi intensif de la distance d'édition standard [Demaine 2009, Pawlik 2011] portant sur les arbres rend inutilisable nos algorithmes. Ce constat s'explique par le fait que cette distance d'édition est polynomiale en temps dans la taille des arbres. Plus précisément, la complexité de l'algorithme calculant cette distance entre deux arbres t_1 et t_2 est en temps cubique $\mathcal{O}(|N_{t_1}|^3)$ et est quadratique $\mathcal{O}(|N_{t_1}|^2)$ en espace, comme dans [Pawlik 2011], où N_{t_1} est l'ensemble des nœuds de t_1. Son emploi est donc à l'origine coûteux. Elle est utilisée dans notre approche par l'algorithme A^* afin d'estimer la distance entre l'arbre courant et l'arbre cible à atteindre lors d'une recherche d'un plus court chemin. L'impact négatif de cette complexité est donc étroitement lié au nombre de voisins visités lors d'une recherche. Notre recherche étant optimale, on pourrait supposer que ces derniers sont en faible nombre et que la complexité de la distance d'édition ne pénalisera que modérément la recherche d'un plus court chemin. Cependant, afin d'être certain de trouver un chemin entre deux nœuds, le choix de considérer les opérations d'édition propres à cette mesure d'édition que sont l'insertion, la délétion et la substitution comme opération de transformation a été fait. Or, parmi ces opérations, l'une d'elles vient lourdement pénaliser la génération du voisinage d'un arbre. Il s'agit de l'insertion. L'insertion dans un arbre $t \in T_\Sigma$ dépend de 3 paramètres que sont :

— un nœud $n \in N_t$ allant devenir le père du nœud à insérer,
— un symbole de Σ allant être le label du nœud inséré
— et une sous-séquence de sous-arbres issue de la séquence des fils de n.

Une insertion permet ainsi, lorsque t est fixé, la génération de $|N_t| \times |\Sigma| \times |subtree_t(n)|$ nouveaux arbres sur lesquels il est nécessaire d'appliquer la distance d'édition. Cette observation vient augmenter significativement le temps nécessaire à la recherche. Ainsi, dans le cas du corpus de football, nous estimons un temps d'apprentissage supérieur à un semestre.

Remarquons que les opérations de substitution et de délétion n'alourdissent pas le processus de recherche puisque le nombre de voisins générés est respectivement $|\Sigma| \times |N_t|$ et $|N_t|$ ce qui demeure raisonnable.

Il convient cependant de relativiser ces propos. En effet, le choix de la distance d'édition utilisée avec nos algorithmes dépend de la tâche étudiée et est laissé libre à l'utilisateur. Ainsi, dans le cas des arbres, nous pensons qu'il existe d'autres distances d'édition pouvant se révéler intéressantes pour notre problème comme par exemple la distance de Selkow [Selkow 1977]. Cette dernière a l'avantage d'être de complexité quadratique dans la taille des arbres sur lesquels elle est appliquée. De plus, contrairement à la distance d'édition que nous avons étudiées, l'usage de insertion et la suppression n'est possible que sur les feuilles de l'arbre traité. Ainsi, les problèmes que nous avons observés devrait être amenuisés. La recherche et l'étude de distances d'édition adaptées à notre problème feront donc l'objet de futurs travaux.

6.5 Travaux Apparentés

Dans cette section, nous présentons plusieurs travaux se rapprochant de notre démarche. Les avantages respectifs de ces méthodes ainsi que de nos travaux vis-à-vis de notre problème de transformation sont mis en avant. Nous regroupons ces différents travaux en deux catégories. La première catégorie correspond à des travaux issus de la PLI tandis que la seconde regroupe des travaux provenant du domaine des transducteurs.

6.5.1 En PLI

De nombreux travaux en PLI [Lavrac 1991, Quinlan 1995, Muggleton 1995, King 2001, Srinivasan 2004, Camacho 2004] ont proposé des algorithmes afin de permettre l'apprentissage de programme logique en tout genre. Nous présentons donc des systèmes capables d'apprendre des programmes non spécifiques à une tâche particulière. Certains d'entre eux reposent cependant sur des restrictions syntaxiques et sémantiques. Ceci s'explique par le fait que l'un des principaux problèmes en PLI est la taille de l'espace de recherche. Pour de nombreux problèmes, cet espace de recherche est exponentiel voire infini comme pour notre problème. Il n'est alors pas probant de le construire. De nombreux algorithmes [Muggleton 1995, Srinivasan 2004] optent ainsi pour la recherche d'une solution dans cet espace.

Une manière d'aborder cet espace de recherche consiste à l'explorer. C'est le cas pour l'algorithme Progol [Muggleton 1995]. Cet algorithme propose une approche hybride associant une recherche ascendante à une recherche descendante. Il prend en entrée des exemples positifs et négatifs. L'apprentissage d'une hypothèse commence par la construction d'une bottom clause $\perp(e)$ à partir d'un exemple positif e grâce à une saturation. La construction de cette clause utilise les biais de langage que sont les modes et les types pour ordonner les littéraux de la bottom clause. La taille d'une bottom clause d'un exemple peut être infinie. Pour cette raison, la profondeur de la saturation est généralement bornée. Une fois la bottom clause construite, l'algorithme cherche à construire une clause plus générale que $\perp(e)$. Cela se fait par l'ajout progressif de littéraux de $\perp(e)$ dont certains termes ont été remplacés par des variables à la clause la plus générale, i.e. la clause vide. La recherche de

cette clause est réalisée à l'aide de l'algorithme A^* basé sur une heuristique appelée "mesure de compression". Celle-ci correspond au nombre d'exemples positifs couverts moins le nombre d'exemples négatifs couverts. Notre approche présente de fortes similitudes avec cet algorithme. Nous utilisons par exemple la saturation mais ne construisons qu'une partie de la bottom clause. La construction d'une hypothèse utilise également l'algorithme A^*. Cependant, nous avons fait le choix de guider la construction avec, pour heuristique, une distance d'édition propre à notre problème. Celle-ci nous semble plus adaptée qu'une mesure de compression généraliste. En effet, la couverture d'une clause de transformation n'a de sens que s'il existe un chemin liant l'entrée du concept à apprendre à sa sortie. Remarquons que l'algorithme proposé par Progol est encore d'actualité et a été généralisé par ALEPH [Srinivasan 2004]. ALEPH est une plateforme logicielle permettant d'émuler un grand nombre d'algorithmes d'ILP comme par exemple CProgol (implémentation de Progol en C), FOIL [Quinlan 1990, Quinlan 1993, Quinlan 1995], Indlog [Camacho 2004], WARMR [King 2001], LINUS [Lavrac 1991], ... L'algorithme de base d'ALEPH est décomposable en 4 étapes paramètrables que sont la sélection d'un exemple positif, la saturation de cet exemple afin de construire la bottom clause $\bot(e)$, l'étape de réduction qui construit une hypothèse plus générale à partir de $\bot(e)$ et la suppression des exemples positifs couverts de l'ensemble d'exemples à couvrir. Ces points sont répétés jusqu'à épuisement des exemples positifs. Cette démarche que nous avons également reprise est ainsi un standard en programmation logique inductive.

Progolem [Muggleton 2010] est un autre système présentant des similitudes avec notre approche. Il est le successeur de l'algorithme Golem [Muggleton 1990]. Ce dernier est une approche ascendante basée sur le moindre généralisé relatif à une théorie ainsi que l'emploi de clauses i, j-déterminée (cf. chapitre 4). Progolem est également un système ascendant dirigé par les données. Il réutilise la construction d'une bottom clause définie dans Progol. Une bottom clause obtenue à partir d'un exemple est ensuite généralisée par l'*Asymmetric Relative Minimal Generalisation* (ARGM). L'ARGM est une variante non déterministe de la méthode de moindre généralisée relative à une théorie utilisée dans Golem. L'ARGM d'une bottom clause d'un exemple e par rapport à un exemple e' consiste en la suppression d'un nombre minimum d'atomes de $\bot(e)$. Seuls des atomes bloquants sont supprimés. Un atome b_i d'un clause $C = h \leftarrow b_1, \cdots, b_n$ est *bloquant par rapport à un exemple positif e et une théorie T* si et seulement si i est l'indice le plus petit tel que pour toute substitution θ avec $h = e\theta$, on ait $T \nvdash (b_1, \cdots, b_i)\theta$. Cette approche présente l'avantage de borner la taille des clauses généralisées par celle de la bottom clause utilisée mais demeure dépendante de l'ordre de généralisation des exemples. Nous avons repris ce principe de suppression des atomes bloquants pour notre algorithme d'élagage. Cependant, ce dernier n'est pas appliqué à la bottom clause mais à un sous-ensemble du moindre généralisé de clauses de transformation qui sont eux-mêmes des sous-ensembles de bottom clause issus d'exemples.

D'autres approches ont tenté de tirer profit de la forme syntaxique des programmes à apprendre. On retrouve ainsi les *Head Output Connected clauses* [Santos 2009]. Il s'agit de clauses dont la déclaration du mode du concept à apprendre contient au moins une sortie. Cette famille de clauses a été développée suite au constat que la mesure de compression ou de couverture orientant la recherche dans de nombreux algorithmes en PLI comme ALEPH [Srinivasan 2004] et Progol [Muggleton 1995] n'était pas pertinente pour l'apprentissage de concepts de ce genre. En effet, ces mesures sont utilisées tout au long de la construction d'une clause hypothèse afin de sélectionner les littéraux ajoutés ou supprimés de celle-ci. Or, un tel test n'est vraiment pertinent qu'une fois la clause entièrement construite afin de la tester sur les exemples. Pour cette raison, l'algorithme proposé dans [Santos 2009] construit une clause connectée respectueuse d'un ensemble de déclarations de mode dont la taille est minimale. Cette clause doit connecter les variables d'entrée de la tête de la clause aux sorties

de ce même atome grâce aux littéraux de son corps. Contrairement à notre approche qui effectue une recherche dirigée par une heuristique, cet algorithme réalise une recherche en largeur. Il énumère ainsi les clauses de l'espace de recherche en fonction de la profondeur de leurs variables. Or, leur profondeur peut rapidement devenir élevée et dépasser la dizaine dans le cas d'une clause de transformation. Notre espace de recherche étant exponentiel, cela reviendrait à énumérer des milliards de transformations. Notre approche nous semble donc plus appropriée pour la tâche de transformation puisque nous évitons cette énumération.

Enfin, la notion de fonctionnalité a elle aussi été source de nombreux travaux [Paulson 1991, Bergadano 1993b, Bergadano 1993a, Hernández-orallo 1999]. Nous retenons parmi ceux-ci l'algorithme FILP [Bergadano 1993b] ainsi que l'algorithme FFOIL [Quinlan 1996]. FILP est un système descendant qui permet l'apprentissage de programmes récursifs ou non sans fonction. Son squelette est proche de celui de FOIL. Pour cette raison, nous ne présentons que le fonctionnement de l'algorithme FFOIL qui est une extension de FOIL [Quinlan 1990, Quinlan 1993, Quinlan 1995]. L'algorithme FFOIL [Quinlan 1996] est un algorithme descendant. Il prend en entrée un ensemble de tuples de la forme $(e_1, \cdots, e_{n-1}, e_n)$ tel qu'il existe une fonction f pour laquelle $f(e_1, \cdots, e_{n-1}) = e_n$. Il a pour but de construire une ou plusieurs clauses ordonnées à partir d'exemples positifs et négatifs. Ces clauses sont une définition de la fonction f. La construction d'une telle clause consiste en l'ajout des littéraux à une clause vide jusqu'à l'obtention d'une connexion entre les variables d'entrée et celles de sortie. Le choix des littéraux est déterminé par une heuristique qui n'est pas détaillée dans [Quinlan 1996]. Nous supposons cependant qu'il s'agit de la mesure de Gain décrite pour l'algorithme FOIL :

$$Gain(C, \ell) = P(C + \ell) \times (I(C) - I(C + \ell)) \text{ avec } I(C) = -\log \frac{P(C)}{P(C) + N(C)}$$

où $Gain(C, \ell)$ traduit le coût d'ajout du littéral ℓ à la clause C, $C + l$ dénote la clause C à laquelle a été ajouté le littéral ℓ, $P(C)$ le nombre d'exemples positifs couverts par la clause C et $N(C)$ le nombre d'exemples négatifs couverts par C. La clause construite est ensuite élaguée. L'élagage consiste en la suppression de littéraux ne devant pas augmenter le taux d'erreur de l'hypothèse. L'apprentissage d'une hypothèse se termine par l'ajout d'un cut (!). De la même manière que pour Progol, l'heuristique choisie ne semble pas être adéquate par rapport à notre problème. De plus FFOIL nécessite des exemples positifs et négatifs pour fonctionner là où nous ne disposons que d'exemples positifs.

Remarquons que les méthodes décrites ci-dessus permettent l'expression de n'importe quelle opération d'édition. Nous continuons la présentation de systèmes capables d'apprendre des transformations avec les transducteurs.

6.5.2 Les Transducteurs

Une autre approche capable de réaliser l'apprentissage de transformations d'objets sont les transducteurs. Une *transduction de $\mathcal{X}$ en $\mathcal{Y}$* est une relation inclue dans $\mathcal{X} \times \mathcal{Y}$. Un processus d'édition peut être ainsi vu comme une transduction entre un élément d'un ensemble $\mathcal{X}$ d'entrées et un élément d'un ensemble $\mathcal{Y}$ de sorties. Les transducteurs ont été initialement introduits pour la transformation de mot à mot par [Mealy 1955, E.F. Moore 1956] puis généralisés au cas des arbres [Rounds 1970, Thatcher 1970]. Les transducteurs sont une extension de la notion d'automate. Contrairement à un automate à état fini qui se contente de reconnaître les éléments d'un langage d'entrée et donc de le lire, un transducteur est un automate étendu qui permet pour chaque étape de transition de produire une sortie. Il le permet au cours de sa lecture et bénéficie donc, contrairement à un automate, d'une sortie. Nous ne nous intéressons qu'aux transducteurs qui retournent une unique sortie de même

type que son entrée et plus particulièrement aux transducteurs déterministes qui assurent une exécution unique pour cette entrée. Ceux-ci permettent la représentation de fonctions partielles et sont ainsi utilisables pour la résolution de problèmes en informatique. Nous répertorions ci-dessous les principaux types de transducteurs définis sur les mots ainsi que sur les arbres.

Transducteurs de mots

Nous commençons avec la classe de *transducteurs déterministes de mots*, que l'on retrouve également dans la littérature sous l'appellation de *transducteurs sous-séquentiels* [Schützenberger 1975]. De tels transducteurs permettent la transduction d'un mot de la gauche vers la droite et ont l'avantage de ne réaliser qu'une exécution sur ce mot. Elle assure également que si une transduction existe alors elle est unique puisque qu'à chaque lettre lue une seule possibilité de continuation lui est offerte. Ces transducteurs peuvent être vus comme une succession d'opérations d'édition standards des mots que sont la substitution, l'insertion et la délétion. Les opérations sont appliquées au fur et à mesure de la transduction, elles dépendent du parcours du mot d'entrée. Ainsi, des opérations comme l'inversion d'un mot ne peuvent être représentée par un transducteur déterministe car elle nécessite de connaître la partie non parcourue du mot.

Les transducteurs déterministes dépendent ainsi de l'ordre de parcours du mot d'entrée. Dans le but de les libérer de cette contrainte, ils ont été étendus avec la notion de *look-ahead* [Elgot 1965]. Le look-ahead correspond à un pré-traitement effectué sur l'entrée dont le but est d'apporter un contexte sur celle-ci. Ce contexte est ensuite utilisé par les opérations lors d'une étape de transduction et peut servir comme conditions à valider afin d'effectuer une étape de transduction. Ces transducteurs sont apprenables polynomialement en temps et en données [Boiret 2012].

Les transducteurs de mots imbriqués à mots imbriqués ont été introduits par [Alur 2012] et sont également connus sous le nom de *well nested visibly pushdown transducers* [Caralp 2013]. Ils constituent une extension du modèle des transducteurs de mots imbriqués à mots dénommés *visibly pushdown transducers* défini dans [Raskin 2008]. Rappelons qu'un nested word ou mot imbriqué bien formé correspond à la linéarisation d'un arbre. Un transducteur de mots imbriqués à mots imbriqués permet ainsi de traiter les arbres d'arité non bornée tout en se rapprochant du cas classique des mots. Il est capable d'exprimer toutes les opérations d'édition définies sur les arbres que sont l'insertion, la délétion et la substitution. Il ne permet cependant pas la copie ni la réorganisation des nœuds. Ce modèle se limite à un unique passage sur la linéarisation ce qui lui permet de calculer la sortie en temps linéaire dans la taille de l'entrée. Il ne peut alors utiliser la partie non parcourue du mot pour orienter la transformation. Afin de pallier ce manque, la notion de *look-ahead* [Filiot 2012] a été proposée pour ce type de transducteur. Elle permet l'annotation du parcours d'un mot et est ainsi capable de générer des conditions utilisables par les opérations lors d'une transduction.

L'approche présentée au cours de ce chapitre permet une plus grande liberté d'expression au niveau des opérations d'édition. Elle est ainsi moins spécialisée à la tâche de transformation de mots à mots que sont les transducteurs de mots à mots. Cependant, comme pour les transducteurs, nous sommes capables de générer un contexte pour les entrées. La définition d'un contexte est alors laissée à l'utilisateur. Celui-ci peut ainsi définir ce dont il a réellement besoin. Cette liberté a néanmoins un coût. En effet, malgré l'admissibilité et la monotonicité des heuristiques de nos algorithmes qui nous assurent une recherche efficace, nous n'avons pas prouvé la polynomialité de notre méthode contrairement à [Boiret 2012]. Pour cette raison, nous ne sommes pas en mesure de garantir une complexité polynomiale

pour nos algorithmes. Malgré cela, il convient de relativiser ce propos. En effet, ce constat n'est vrai que pour les transducteurs avec look-ahead définis sur les mots. Nous pensons ainsi que notre approche peut rivaliser avec celle des transducteurs par le choix judicieux d'opérations d'édition et de fonctions de contexte dans le cas de transformations non récursives, la question de la récursivité n'étant pas abordée dans cette thèse. Cette dernière fera l'objet de futurs travaux. Nous nous intéressons maintenant aux transducteurs définis pour arbres.

Transducteurs d'arbres

Les transducteurs présentés ci-dessous portent sur des arbres d'arité bornée. Rappelons qu'il est possible de transformer un arbre d'arité non bornée en un arbre d'arité bornée avec un codage *first-child next-sibling* (cf. chapitre 3).

Nous commençons par le modèle des *transducteurs descendants d'arbres d'arité bornée*. Il a été défini par [Rounds 1970, Thatcher 1970]. Le parcours d'un arbre lors d'une transduction descendante débute par la racine de l'arbre. Ce nœud est ensuite transformé. Le processus de transformation continue avec chaque sous-arbre du nœud courant en le transformant en parallèle. Un tel transducteur permet la représentation de transformation utilisant l'opération de copie, la réutilisation de sous-arbres afin de produire plusieurs sorties, la réorganisation des sous-arbres ainsi que la suppression des sous-arbres à chaque étape de la transduction. Cette suppression concerne un sous-arbre dans sa totalité et ne permet pas la suppression d'un nœud ou d'une partie interne du sous-arbre examiné. Il a ainsi comme inconvénient de ne pouvoir utiliser que l'information issue des nœuds déjà parcourus. Ce modèle possède l'avantage d'être identifiable polynomialement à la limite en temps et en données [Lemay 2010].

Afin de pallier l'absence de connaissance sur le sous-arbre d'un nœud qui permettrait le traitement de ce nœud en fonction du sous-arbre, la notion de *transducteur d'arbres avec lookahead* a été proposée [Engelfriet 1977]. Elle correspond à une extension des transducteurs descendants et ajoute la capacité d'inspecter le sous arbre dont la racine est le nœud en cours de traitement. La quantité d'information obtenue lors de cette inspection doit être finie. Cette contrainte nous est également imposée dans notre approche.

Une autre manière de parcourir l'arbre lors d'une transduction consiste à partir des feuilles pour arriver à la racine. Cette approche est la base des *transducteurs ascendants d'arbres* [Rounds 1970, Thatcher 1970]. Les opérations utilisées par ce transducteur sont les mêmes que celles des transducteurs descendants mais sont appliquées différemment du fait du parcours de l'arbre. Ainsi, la copie ne porte que sur les sous-arbres obtenus suite à la transduction. La suppression, quant-à-elle, nécessite le parcours intégral du sous-arbre à supprimer. Elle permet cependant et contrairement aux transducteurs descendants la suppression d'un nœud interne à un sous-arbre. Ces points font que les transducteurs d'arbres ascendants et descendants ont une expressivité différente. Il n'existe pour le moment aucun résultat d'apprenabilité pour ce modèle, plus particulièrement l'existence d'un algorithme polynomial d'apprentissage reste un problème ouvert.

Comme pour le cas des mots, nos algorithmes d'apprentissage se rapprochent davantage des transducteurs d'arbres avec look-ahead. Contrairement à ce dernier, nous ne subissons pas de limitation au niveau de la suppression ou encore de la copie. Notre approche semble bénéficier donc d'un léger avantage à ce niveau. Il ne nous est pas non plus nécessaire d'effectuer le codage d'un arbre d'arité bornée en un d'arité non bornée. Il faut cependant relativiser ces remarques. En effet, la complexité de ces différentes approches n'est pas connue. Nous ne pouvons donc les comparer sur ce point. Plus de détails sur les transducteurs d'arbres ainsi que des comparaisons entre les différents modèles peuvent être trouvés

dans [Comon 1997, Maletti 2009, Razmara 2011].

6.6 Conclusion

Nous avons modélisé au cours de ce chapitre un langage de clauses capable de représenter des transformations d'éléments d'un ensemble $\mathcal{X}$ vers un autre élément de ce même ensemble. Ce type de clause a été appelée clause de transformation. Il s'agit d'une adaptation de la notion de script d'édition à la PLI. Un parallèle a ainsi été fait entre une opération d'édition d'un script d'édition et un atome de transformation dans une clause de transformation. Afin d'outrepasser l'expressivité imposée par les scripts d'édition, nous avons introduit la notion d'atome de contexte. Ces atomes ont pour objectif de donner des informations sur un terme utilisé comme entrée d'un atome de transformation. Ils ont également pour effet de paramétrer dynamiquement ces atomes de transformation. Notre représentation d'une transformation introduite, nous avons ensuite proposé deux algorithmes d'apprentissage de clause de transformation. Ces algorithmes sont basés sur l'exploration de l'espace de recherche par un algorithme A^* dans le but de trouver une clause de transformation. Ils utilisent comme heuristique une distance d'édition à définir en fonction du problème de transformation. Nous avons prouvé génériquement qu'une distance d'édition est une bonne heuristique pour ce problème. En effet, elle est monotone et admissible.

Ces deux méthodes constituent un framework d'apprentissage pour les transformations. Elles ont l'avantage de ne pas être spécialisées à la transformation d'un objet en particulier comme les mots ou les arbres. La capacité d'apprendre des transformations pour ces derniers a été testée par nos expériences. De nombreux points restent cependant à éclaircir. Ainsi, il serait intéressant de bénéficier d'opérations de transformation et de contexte prédéfinies pour ces deux problèmes. Ceci nous permettrait d'établir des résultats de complexité. Ce travail reste donc à faire. Enfin, la question de l'apprentissage de programme récursif basé sur ce type de clauses reste à étudier.

Chapitre 7

Conclusion

Nous avons abordé au cours de cette thèse les tâches de classification et de transformation de documents XML.

Concernant la tâche de classification, notre contribution consiste en la définition d'un framework disposant d'opérations polynomiales de θ-subsomption et de moindre généralisation. Ces opérations ont été définies pour plusieurs familles de clauses : $\mathcal{F}_{sop}, \mathcal{F}_{\mathsf{QD}}, \mathcal{F}_{\mathsf{MQD}}$ et $\mathcal{F}_{\mathsf{FDP}}$.

La famille $\mathcal{F}_{sop}$ est une sous-famille des clauses déterminées. Elles bénéficient d'opérations de θ-subsomption et de moindre généralisation linéaires dans la taille des clauses. Son moindre généralisé, contrairement au cas général des clauses, est réduit. Ainsi, aucune réduction n'est nécessaire. De plus, sa taille est bornée par la taille minimale des clauses généralisées. Elle n'est donc jamais exponentielle dans le nombre d'exemples contrairement au cas des clauses de Horn, par exemple. La famille $\mathcal{F}_{sop}$ bénéficie ainsi de propriétés plus intéressantes que celle des clauses déterminées. Malgré son appartenance à la famille des clauses déterminées, elle contient des langages de clauses capables de représenter, avec ou sans perte, des structures arborescentes. Nous avons ainsi proposé les langages $\mathcal{L}_{pc}^{\Sigma\cup V}$, $\mathcal{L}_{df}^{\Sigma}$ et $\mathcal{L}_{peano}^{\Sigma}$. Ces langages reposent sur un principe d'identification unique des nœuds, des arcs ou des labels dans un arbre. Le premier langage permet la représentation d'arbres ordonnés, d'arité non bornée, partiellement, totalement ou non étiquetés. Il utilise des variables afin de spécifier si des nœuds ont un même label, ce label peut être connu ou non. Les deux autres langages sont des représentations avec perte des arbres de T_Σ. Le langage $\mathcal{L}_{df}^{\Sigma}$ représente un arbre par son parcours transversal tandis que le langage $\mathcal{L}_{peano}^{\Sigma}$ compte le nombre d'occurrences de chaque label dans un arbre. Afin de les rendre sans perte, nous avons montré qu'il est possible de combiner plusieurs langages SOP sans prédicat commun. Nous avons ensuite prouvé que le langage $\mathcal{L}_{pc}^{\Sigma\cup V}$ est identifiable polynomialement à la limite en temps et en données par moindre généralisation à partir de positifs seuls et de positifs et négatifs. Ainsi, un apprentissage à partir de ce langage est possible et devrait être simple au vu de la complexité des méthodes qui lui sont associées. Les clauses de ce langage sont contraintes. Elles ne doivent contenir qu'au plus un atome pour chaque prédicat autorisé d'un langage. Cette propriété garantit le déterminisme et a permis la proposition d'algorithmes efficaces. Cependant, elle contraint la forme des clauses autorisées.

Afin d'outrepasser cette restriction des langages SOP, nous avons présenté les familles de clauses quasi-déterminées et multi-quasi-déterminées. La famille $\mathcal{F}_{\mathsf{QD}}$ englobe celle des clauses déterminées. Une clause quasi-déterminée est ordonnée. Elle autorise, contrairement à une clause déterminée, plusieurs appariements possibles pour le premier littéral. Une fois ce choix fait, le reste de la clause est déterminé. Cette définition nous a permis de définir une θ-subsomption quadratique dans la taille des clauses. La famille $\mathcal{F}_{\mathsf{QD}}$ contient des langages capables de représenter les arbres de T_Σ comme $\mathcal{L}_{fcns}^{T_\Sigma}$ ou des mots imbriqués bien formés comme $\mathcal{L}_{nested}^{T_\Sigma}$. Cependant, ils ne sont pas clos par moindre généralisation. Pour cette raison, nous avons défini la famille $\mathcal{F}_{\mathsf{MQD}}$ qui inclut la famille $\mathcal{F}_{\mathsf{QD}}$. Une clause multi-quasi-déterminée est une multi-clause dont les composantes sont QD. Elle dispose d'une θ-subsomption quadratique dans la taille des clauses. Afin de transformer une clause en

multi-clause, une décomposition d'une clause variabilisée en multi-clause a été présentée. Elle est utilisable sur une sous-famille de clauses de $\mathcal{F}_{\mathsf{MQD}}$ appelée $\mathcal{F}_{\mathsf{FDP}}$. Cette famille de clauses bénéficie des algorithmes polynomiaux définis pour la famille $\mathcal{F}_{\mathsf{MQD}}$. À l'instar des clauses de Horn, le calcul du moindre généralisé de deux clauses de $\mathcal{F}_{\mathsf{FDP}}$ est effectué en temps quadratique dans la taille des clauses à généraliser. Cette famille contient des langages capables de représenter des structures arborescentes. Parmi ces derniers se trouvent les langages MQD : $\mathcal{L}^{\Sigma}_{fcns}$ et $\mathcal{L}^{\Sigma}_{nested}$. Le premier est une représentation d'arbres ordonnées, d'arité non bornée, totalement ou partiellement étiquetés. Elle est basée sur l'usage d'un codage *first-child next-sibling*. Contrairement, au langage $\mathcal{L}^{\Sigma\cup V}_{pc}$, il ne gère pas la présence de variable mais permet la représentation de motifs d'arbres totalement ou partiellement étiquetés. Le second est une représentation des arbres à l'aide de mots imbriqués bien formés. Chacun de ces langages peut souffrir d'un moindre généralisé de taille exponentielle dans le nombre de clauses généralisées. Comme pour les langages SOP, il est possible de combiner plusieurs langages FDP ne partageant pas de prédicat en commun. Nous avons montré que les langages $\mathcal{L}^{\Sigma}_{fcns}$ et $\mathcal{L}^{\Sigma}_{nested}$ sont identifiables à la limite par moindre généralisation à partir de positifs seuls et de positifs et négatifs. Cependant, l'apprentissage d'une hypothèse $\mathcal{L}^{\Sigma}_{fcns}$ à partir d'une clause représentant un arbre est un problème NP-complet. Cette apprentissage risque donc de s'avérer difficile contrairement à celui des clauses SOP.

Les langages $\mathcal{L}^{\Sigma\cup V}_{pc}$ et $\mathcal{L}^{\Sigma}_{fcns}$ diffèrent par les arbres et motifs d'arbres qu'ils sont capables de représenter. Ainsi, la généralisation des clauses représentant le même ensemble d'arbres de T_{Σ} n'est pas la même pour ces langages. Afin de bénéficier de ces deux représentations, nous avons proposé de les coupler. Les opérations de θ-subsomption et de moindre généralisation des langages couplés ont une complexité égale à celle de leurs homologues de la famille $\mathcal{F}_{\mathsf{FDP}}$. Nous avons montré expérimentalement que ces familles et langages de clauses peuvent être utilisés pour un apprentissage sur des données réelles.

L'espace des clauses MQD est vaste. Nous sommes persuadés qu'il existe de nombreuses autres applications pour les langages de la famille $\mathcal{F}_{\mathsf{FDP}}$ mais également pour ceux de la famille $\mathcal{F}_{\mathsf{MQD}}$. La recherche d'autres algorithmes de décomposition nécessaires à des clauses $\mathcal{F}_{\mathsf{MQD}}$ peut donc être intéressante. Il conviendra alors de s'interroger sur les applications de langages basés sur ces décompositions. Notre approche d'apprentissage disjonctif repose sur l'utilisation de l'algorithme DLG. Comme mentionné lors de cette étude, il peut également être intéressant de voir l'apport d'algorithmes de boosting. Enfin, nous avons fait l'ébauche, au cours de l'étude de la famille $\mathcal{F}_{\mathsf{QD}}$, d'une possible extension de la contrainte d'appariements multiples du premier littéral d'une clause QD à k autres littéraux. La famille construite de cette manière bénéficierait d'une plus grande expressivité. Il est de même de l'adaptation de la famille MQD se basant sur cette nouvelle famille. Il nous semble intéressant de regarder ces familles notamment pour les applications ne présentant pas de déterminisme ou de quasi-déterminisme comme l'apprentissage de propriétés chimiques de molécule. Ainsi, nos futurs travaux porteront sur ces différents points.

Concernant la tâche de transformation, nous avons présenté une approche basée sur une adaptation à la PLI des scripts d'édition. Nous avons commencé par modéliser des clauses appelées *clauses de transformation*. Elles sont une adaptation de la notion de script d'édition. Elles s'apparentent ainsi à une composition d'opérations d'édition élémentaires représentées par une séquence d'atomes connectés. Un parallèle a ainsi été établi entre une opération d'édition d'un script d'édition et un atome de ce type de clause, appelé *atome de transformation*. Afin d'outrepasser l'expressivité imposée par les scripts d'édition, nous avons introduit, dans ces dernières, la notion de contexte à l'aide d'atomes appelés *atomes de contexte*. Un atome de contexte donne des informations sur les éléments manipulés lors d'une transformation et dispose, dans la théorie du domaine, d'une définition fonctionnelle. Au moins l'un des arguments de ce type d'atome est utilisé comme paramètre dans un atome

de transformation. Ainsi, dans une clause de transformation, chaque terme transformé est associé à un contexte fini le décrivant. Les autres arguments d'un atome de contexte peuvent être présents dans un atome de transformation ou de contexte. Ils permettent alors de paramétrer dynamiquement les atomes les utilisant. Les clauses de transformation ainsi définies se placent dans la lignée des programmes fonctionnels [Paulson 1991, Bergadano 1993b, Bergadano 1993a, Hernández-orallo 1999, Quinlan 1996, Santos 2009]. Avant de proposer un algorithme d'apprentissage de transformation, nous avons réalisé une étude déterminant la taille de l'espace de recherche dans le cas de transformations simples, i.e. sans contexte. Cette étude a permis de constater que cet espace est de taille exponentielle. Ainsi, afin de permettre l'apprentissage de ces transformations, nous avons opté pour la recherche d'une hypothèse dans ce dernier.

Les algorithmes d'apprentissage de clauses de transformation que nous avons proposés se basent sur une exploration de l'espace de recherche à l'aide d'un algorithme A^* adapté à notre problème. Cette approche est courante en PLI, on la retrouve, par exemple, dans HOC-learner [Santos 2009]. La recherche dans l'ensemble des hypothèses est guidée par une heuristique. Cette dernière est, dans notre cas, une distance d'édition définie sur l'ensemble des exemples. Pour les mots, il s'agit de la distance de Levenshtein et pour les arbres, la distance d'édition standard définie sur les arbres. Nous avons ensuite prouvé qu'une distance d'édition sur un ensemble donné est monotone et admissible pour notre approche. Ainsi, l'efficacité de la recherche est assurée. Le choix d'utiliser une distance d'édition comme heuristique ne nous semble pas fréquent en PLI. En effet, la plupart des algorithmes de PLI ont pour heuristique une mesure générique spécifiant l'efficacité de la solution passée en revue. C'est notamment le cas du *gain* de FOIL [Quinlan 1990, Quinlan 1993, Quinlan 1995]. Ainsi, notre heuristique est propre à la nature des données manipulées et doit être adaptée en fonction du problème de transformation. Il est donc possible, pour un même problème, de tester plusieurs distances d'édition afin d'en trouver une répondant à nos besoins.

Notre premier algorithme s'appuie sur la recherche, pour chaque exemple, d'une transformation commune. Ces transformations sont ensuite généralisées à la manière de DLG. On obtient alors plusieurs généralisations maximalement correctes. Ce procédé est fréquent en PLI et est similaire à celui employé pour résoudre notre problème de classification de documents XML. Cependant, une transformation est commune à un seul exemple. Or, il peut exister, pour un même exemple, plusieurs transformations optimales. Afin de choisir une transformation optimale commune au plus grand nombre d'exemples, nous avons proposé un second algorithme. Il consiste en une recherche dirigée par plusieurs exemples. Ainsi, la transformation trouvée n'est plus commune à un exemple mais à une séquence d'exemples pour laquelle elle a été construite. Elle ne généralise plus les transformations de plusieurs exemples mais des étapes élémentaires propres à chaque élément de cette séquence. Ainsi, seules les opérations d'édition élémentaires communes aux exemples de la séquence sont explorées. Ces deux méthodes constituent un framework d'apprentissage pour nos clauses de transformation. Elles ont l'avantage de ne pas être propres à la transformation d'un objet en particulier comme les mots ou les arbres. Nous les avons testées expérimentalement afin d'évaluer leur efficacité sur des corpus réels ou non. Cependant, la représentation d'une transformation sous forme séquentielle peut s'avérer inappropriée pour certains problèmes. Nous souhaitons donc, dans nos futurs travaux, étendre notre approche ainsi que nos algorithmes à l'apprentissage de programmes récursifs.

Enfin, comme expliqué ci-dessus, nos méthodes utilisent une distance d'édition. Dans le cas des mots et des arbres, ils existent plusieurs choix possibles. Or, nous avons testé celles standards. Ainsi, nous pensons intéressant de tester expérimentalement l'efficacité d'autres distances comme [Selkow 1977] afin de simplifier l'apprentissage.

Annexe A

Annexe

A.1 Langages d'arbres dénombrables

Dans cette section, nous prouvons que l'ensemble T_Σ des arbres ordonnées étiquetés sur un alphabet Σ est dénombrable et qu'il en est de même pour que le langage T_Σ^{fcns} obtenu par codage des arbres de T_Σ en respectant le codage *fcns* présenté dans lors de la sous-section 3.1.4 du chapitre 3.

Définition 108 (Bijection)**.** *Une application $f : A \to B$ entre deux ensembles est une bijection si $\forall y \in B$, il existe exactement $x \in A$ t.q. $f(x) = y$.*

Définition 109 (Ensemble dénombrable)**.** *Un ensemble E est dit dénombrable s'il existe une bijection de E sur un sous ensemble de l'ensemble $\mathbb{N}$ des entiers naturels.*

Dans la suite, les arbres utilisés sont toujours ordonnés. On veut montrer que l'ensemble des arbres non bornés sur un ensemble ordonné $\Sigma = \{l_1, ..., l_n\}$, noté T_Σ, est dénombrable. $\forall t \in T_\Sigma$, on sait que t peut être transformé en un arbre binaire $t' \in T_\Sigma^{fcns}$ à l'aide d'un codage *first-child next-sibling* où T_Σ^{fcns} est l'ensemble des arbres binaires respectant le codage *fcns* sur Σ. On sait alors que $\forall t \in T_\Sigma$, il existe exactement un $t' \in T_\Sigma^{fcns}$ et inversement.

Afin de montrer que T_Σ est dénombrable, on va démontrer que T_Σ^{fcns} l'est. Soit $\mathbb{P} = \{2, 3, 5, ...\}$ l'ensemble ordonné des nombres premiers où $\mathbb{P} \subset \mathbb{N}$. On considère $T_\mathbb{P}$ l'arbre binaire infini parfait dont les noeuds sont les éléments de $\mathbb{P}$ t.q. chaque élément est placé, l'un après l'autre en suivant l'ordre de $\mathbb{P}$, le plus en haut possible dans l'arbre puis le plus à gauche possible. La figure A.1 décrit une partie de cet arbre.

On considère maintenant une fonction *score* qui prend en entrée un noeud d'un arbre $t \in T_\Sigma^{fcns}$ ayant un label $l_i \in \Sigma$ et retourne le nombre premier correspondant dans $T_\mathbb{P}$ (après alignement des arbres t et $T_\mathbb{P}$ à partir de la racine) à la puissance i.

Enfin, on définit une fonction h qui prend en entrée un arbre $t \in T_\Sigma^{fcns}$ et renvoie $h(t) = \prod score(n)$ où n correspond à chacun des noeuds de t. $h(t)$ est donc un produit de nombres premiers d'où $h(t) \in \mathbb{N}$. On sait qu'un nombre premier n'est divisible que par 1 et lui même, qu'un noeud d'un arbre à une position fixée sera toujours associé au même nombre premier et que Σ est un ensemble ordonné (totalement), de ce fait $\forall t, t' \in T_\Sigma^{fcns}$ t.q. $t \neq t'$, on a $h(t) \neq h(t')$ ce qui signifie que deux arbres de T_Σ^{fcns} ne sont jamais associés à un

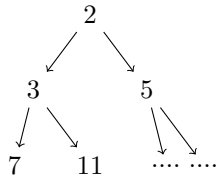


Figure A.1 – Début de l'arbre binaire complet des nombres premiers

même entier. On a donc un ordre total entre les arbres de T_Σ^{fcns}, h est donc une bijection de T_Σ^{fcns} vers un sous ensemble de $\mathbb{N}$. De ce fait T_Σ^{fcns} est dénombrable, tout comme T_Σ.

A.2 Exemple d'explosion de la taille du moindre généralisé de clauses de $\mathcal{L}_{fcns}$ représentant des arbres d'arité bornée à 1

L'exemple suivant illustre l'utilisation de la construction du chapitre 5 à la sous-section 5.3.8. Les arbres suivants sont représentés de manière horizontale à la manière de mot, ainsi nous utiliserons principalement le terme de mot.

Exemple 67. *L'exemple suivant illustre l'utilisation de la construction donnée précédemment pour un alphabet $\Sigma = \{a, b, c, d, e, f\}$. De ce fait, nous nous plaçons dans le cas où $n = 3$. On a donc un ensemble E contenant 3 clauses. Chaque clause ne contient qu'une seule composante. Dans ces composantes, on retrouve un motif commun de la forme (à un renommage de variable près) :*

$$fc(Y_1, Y_2), fc(Y_2, Y_3), fc(Y_3, Y_4), fc(Y_4, Y_5), fc(Y_5, Y_6)$$

En fonction de l'exemple, chaque variable de ce motif sera connectée ou non à un littéral étant le même pour chaque variable et qui correspondra à un label.

$X_1 - b - c - d - e - f - a - b - c - X_{10} - e - f$

Clause 1

$a - X_2 - c - d - e - f - a - b - c - d - X_{11} - f$

Clause 2

$a - b - X_3 - d - e - f - a - b - c - d - e - X_{12}$

Clause 3

FIGURE A.2 – Ensemble E

Nous représenterons chaque exemple sous la forme de mots. Chaque composante d'une clause correspond alors un mot dont les connexions sont représentées par le symbole ns. Un lien horizontal dénotera un lien de fraternité (ns) permettant de connaître la lettre suivante.

$X_1 - X_2 - c - d - e - f - a - b - c - X_{10} - X_{11} - f$

QD_1

$1 - X_2 - c - X_4 - e - f$ $\quad$ $X_1 - b - c - d - X_5 - f$

QD_2 $\quad$ QD_3

FIGURE A.3 – Moindre généralisé de C_1 et C_2

Le moindre généralisé de C_1 et C_2 contient le motif commun constitué de 6 symboles de prédicat. Ce dernier est répété 4 fois en tout : 1 fois dans chaque petite composante et 2 fois dans la grande. Ceci s'explique par la disparition progressive des labels des nœuds au fur et à mesure des généralisations. Ainsi, ce phénomène s'accentuera en continuant la généralisation avec C_3.

$$X_1 - X_2 - X_3 - d - e - f - a - b - c - X_{10} - X_{11} - X_{12}$$
$$QD_1$$

$$X_1 - X_2 - c - d - e - X_6 \qquad a - b - X_3 - X_4 - X_5 - f$$
$$QD_2 \qquad\qquad QD_3$$

$$a - X_2 - X_3 - X_4 - e - f \qquad a - X_2 - c - X_4 - e - X_6$$
$$QD_4 \qquad\qquad QD_5$$

$$X_1 - b - X_3 - X_5 - e - f \qquad X_1 - b - c - d - X_5 - X_6$$
$$QD_6 \qquad\qquad QD_7$$

FIGURE A.4 – Moindre généralisé de C_1, C_2 et C_3

A.3 Illustration de la construction d'un moindre généralisé exponentiel pour le langage $\mathcal{L}_{nested}$

Exemple 68. *Nous reprenons l'exemple précédent et adaptons sa construction au cas des* Nested Words.

$<t><a><c></c><d></d><e></e><f></f>\cdots</t>$
$<t><a_1></a_1><c></c><d></d><e></e><f></f>\cdots</t>$
$<t><a><c></c><d></d><e></e><f></f>\cdots</t>$
$<t><a><b_1></b_1><c></c><d></d><e></e><f></f>\cdots</t>$
$<t><a><c></c><d></d><e></e><f></f>\cdots</t>$
$<t><a><c_1></c_1><d></d><e></e><f></f>\cdots</t>$

Nous retrouvons ainsi la première partie du mot $abcdefabcdef$ *adaptée au cas des* Nested Words. *Le symbole* $\cdots$ *désigne la deuxième partie de ce motif qui a été omis par faute de place. Ainsi, la taille du moindre généralisé de cet ensemble est exponentielle dans le nombre d'exemples, plus précisément il est en* $\mathcal{O}(2^{n/2})$.

Nous avons donc un moindre généralisé dont la taille est exponentielle dans le nombre d'exemples. On peut remarquer que le mécanisme employé pour assurer l'explosion du moindre généralisé de mots est similaire à celui employé dans le cas des arbres. Il s'agit toujours d'un motif commun à tous les exemples de départ qui est rattaché à d'autres atomes, ces derniers subissant une légèrement variation en fonction des exemples.

A.4 Exemple d'explosion de la taille du moindre généralisé de clauses de $\mathcal{L}_{fcns}$

L'exemple suivant illustre la construction donnée dans le chapitre 5 à la sous-section 5.3.8 entrainant l'explosion de la taille d'un moindre généralisé pour un nombre d'exemple fini de $\mathcal{L}_{fcns}$.

Exemple 69. *Nous nous plaçons dans le cas où* $n = 3$. *On a donc un ensemble* E *contenant* 3 *clauses. Chacune de ces clauses contient deux composantes indépendantes. Dans ces composantes, on retrouve un motif commun de la forme (à un renommage de variable près) :*

$$fc(Y_1, Y_2), fc(Y_2, Y_3), fc(Y_3, Y_4), fc(Y_4, Y_5), fc(Y_5, Y_6)$$

En fonction de l'exemple et de la composante, chaque variable de ce motif sera connectée ou non à un ensemble de littéraux étant le même pour chaque composante de chaque clause.

Nous représenterons chaque exemple sous la forme d'une forêt d'arbres. Chaque composante d'une clause est ainsi un arbre respectant le codage first-child next-sibling. *Dans les figures ci-dessous, un lien oblique dénote de la notion de frère-fils (fc) tandis qu'un lien horizontal représente la notion de fraternité (ns).*

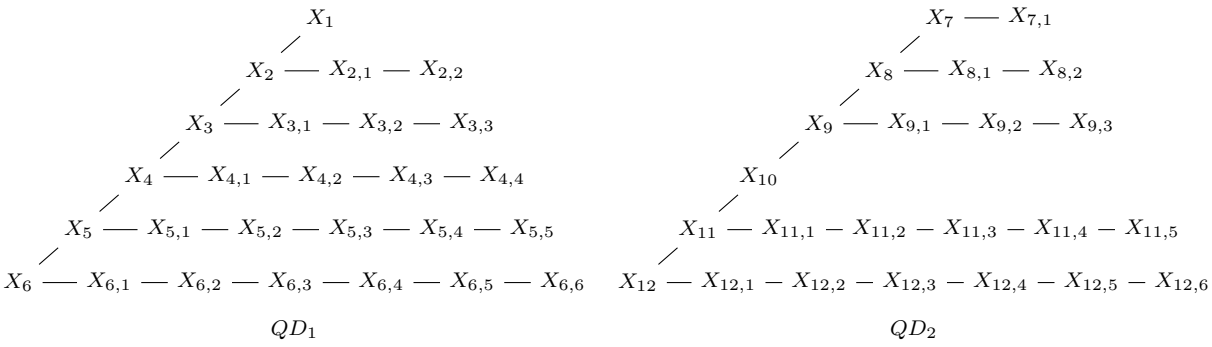


FIGURE A.5 – Exemple C_1

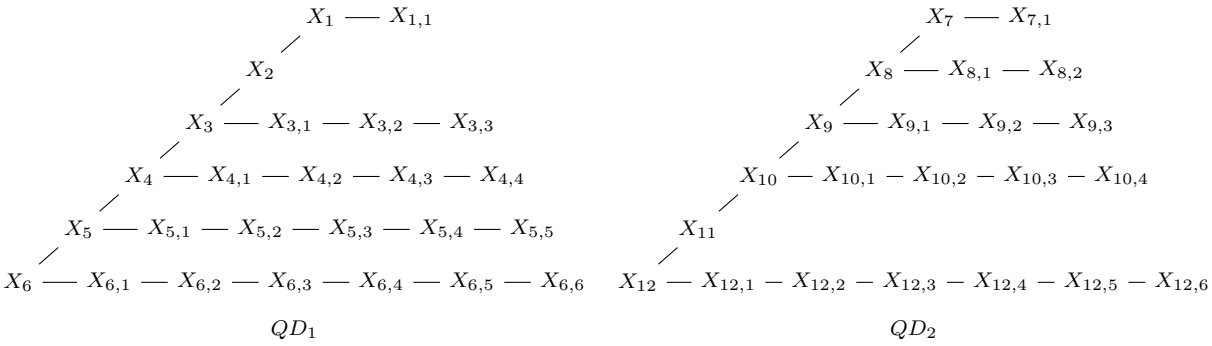


FIGURE A.6 – Exemple C_2

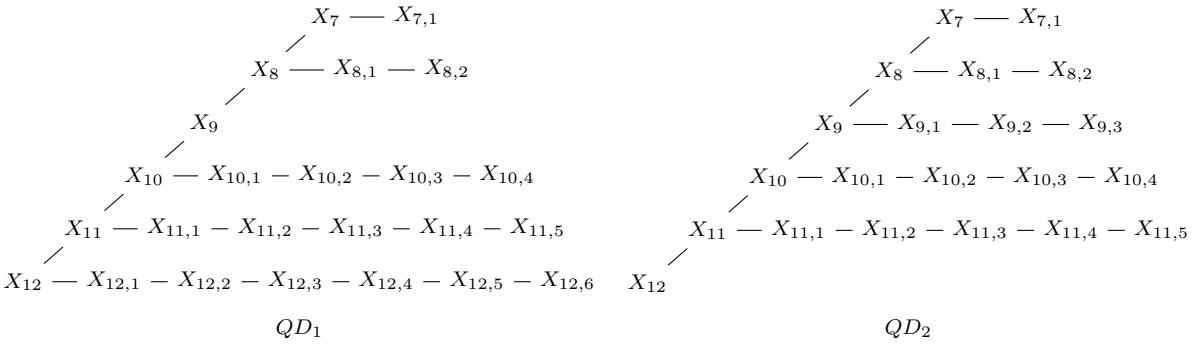


FIGURE A.7 – Exemple C_3

On donne maintenant le moindre généralisé obtenu pour les clauses C_1 et C_2.

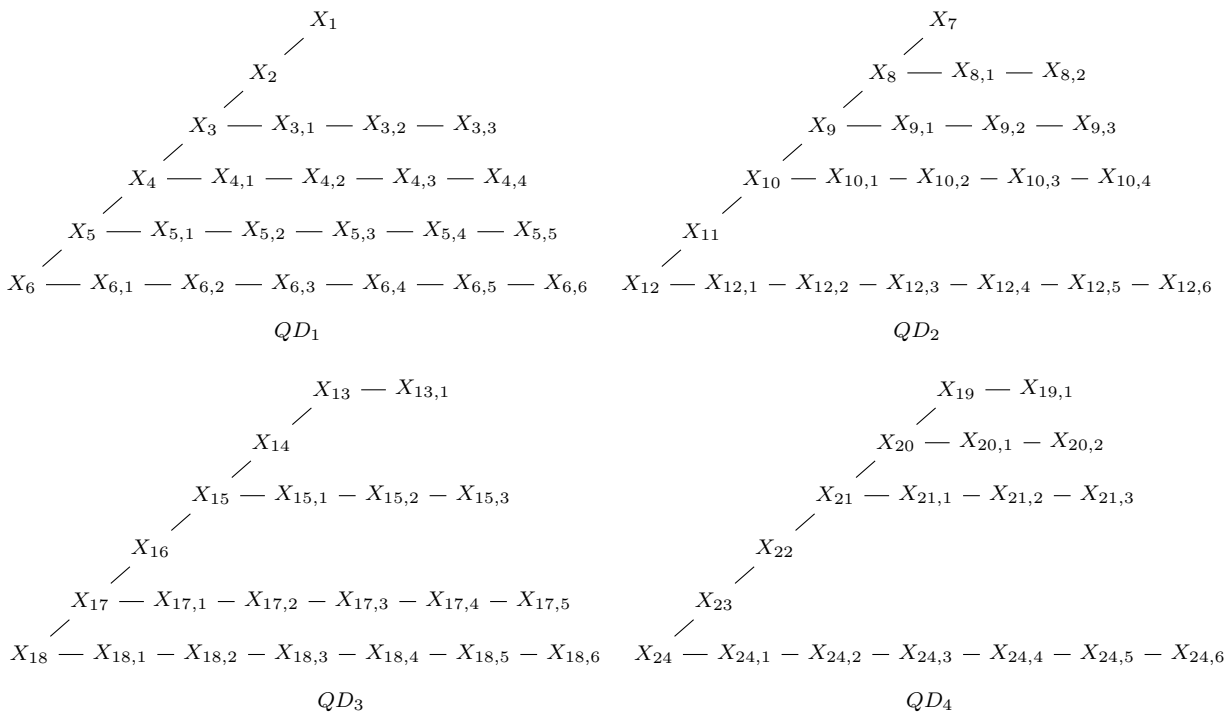
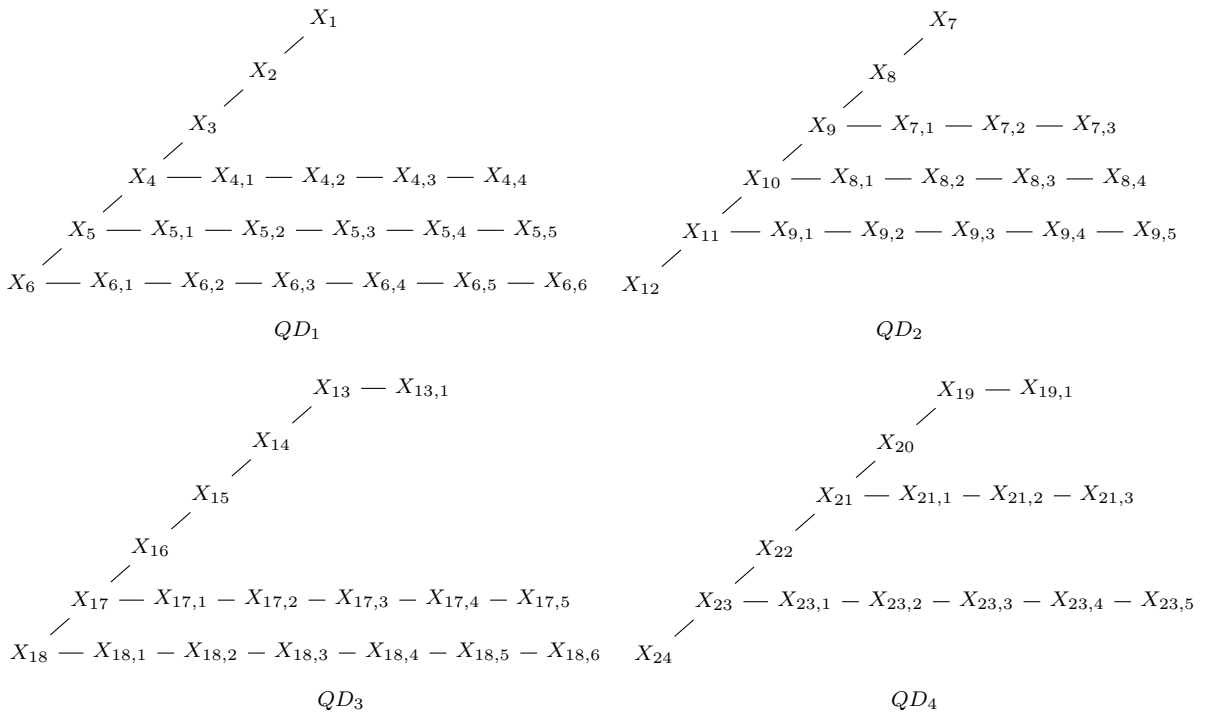


FIGURE A.8 – Moindre généralisé de C_1 et C_2

On remarque que ce moindre généralisé contient, dans chacune de ces composantes, le motif présent dans les clauses C_1 et C_2. Ainsi le phénomène d'explosion commence.



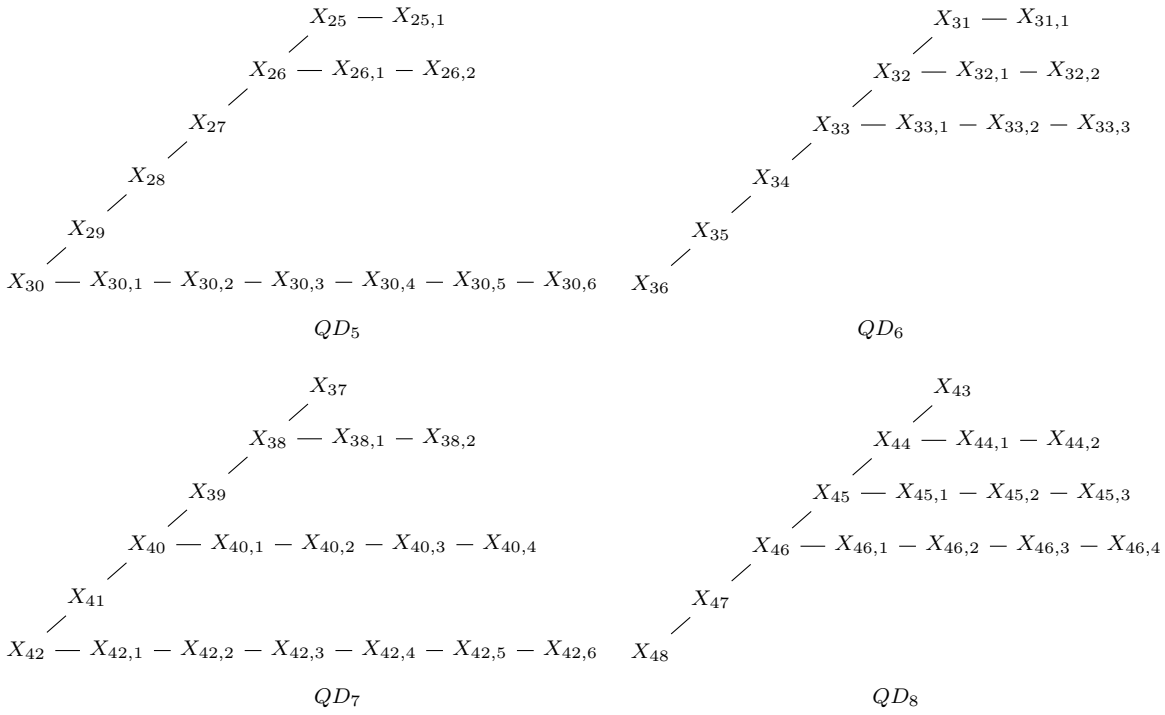


FIGURE A.9 – Moindre généralisé de C_1, C_2 et C_3

Nous nous intéressons maintenant au moindre généralisé des clauses C_1, C_2 et C_3. On remarque tout d'abord qu'il comporte 8 composantes. Chacune de ces composantes contient, comme c'était le cas pour le moindre généralisé de C_1 et C_2, le motif vu précédemment ainsi que des littéraux connectés aux variables de ce motif. Le motif présent dans chacune de ces clauses a donc été multiplié par 2^2 pour le moindre généralise de C_1 et C_2 et ensuite par 2^3 pour le moindre généralisé de C_1, C_2 et C_3. Ainsi, la taille exponentielle du moindre généralisé s'explique par la présence de ce motif invariant dans toutes les composantes de chaque clause de l'ensemble. Ce dernier est connecté à l'ensemble de littéraux partiellement communs à toutes les composantes et subissant des variations. Le moindre généralisé conserve ainsi la partie commune tout en réduisant au fur et à mesure l'ensemble des littéraux partiellement communs.

Bibliographie

[Alexander 2011] David Alexander, Paavo Arvola, Thomas Beckers, Patrice Bellot, Timothy Chappell, Christopher M. De Vries, Antoine Doucet, Norbert Fuhr, Shlomo Geva, Jaap Kamps, Gabriella Kazai, Marijn Koolen, Sangeetha Kutty, Monica Landoni, Véronique Moriceau, Richi Nayak, Ragnar Nordlie, Nils Pharo, Eric SanJuan, Ralf Schenkel, Andrea Tagarelli, Xavier Tannier, James A. Thom, Andrew Trotman, Johanna Vainio, Qiuyue Wang and Chen Wu. *Report on INEX 2010*. SIGIR Forum, vol. 45, no. 1, pages 2–17, 2011. 34, 122

[Alfred North Whitehead 1912] Bertrand Russell Alfred North Whitehead. Principia mathematica, volume 2. Cambridge University Press, 1912. 2, 7

[Alur 2012] Rajeev Alur and Loris D'Antoni. *Streaming Tree Transducers*. In Artur Czumaj, Kurt Mehlhorn, Andrew M. Pitts and Roger Wattenhofer, editeurs, ICALP (2), volume 7392 of *Lecture Notes in Computer Science*, pages 42–53. Springer, 2012. 151, 208

[Becker 2011] MoritzY. Becker and Jason Mackay. *Relaxed Safeness in Datalog-Based Policies*. In Frank Olken, Monica Palmirani and Davide Sottara, editeurs, Rule - Based Modeling and Computing on the Semantic Web, volume 7018 of *Lecture Notes in Computer Science*, pages 49–57. Springer Berlin Heidelberg, 2011. 15

[Benedikt 2009] Michael Benedikt and Christoph Koch. *XPath leashed*. ACM Comput. Surv., vol. 41, no. 1, pages 3 :1–3 :54, January 2009. 35

[Bergadano 1993a] F. Bergadano and D. Gunetti. *Functional inductive logic programming with queries to the user*. In PavelB. Brazdil, editeur, Machine Learning : ECML-93, volume 667 of *Lecture Notes in Computer Science*, pages 323–328. Springer Berlin Heidelberg, 1993. 5, 157, 207, 213

[Bergadano 1993b] Francesco Bergadano and Daniele Gunetti. *An Interactive System to Learn Functional Logic Programs*. In Proceedings of the International Joint Conference on Artificial Intelligence (IJCAI'93), 1993. 3, 5, 157, 207, 213

[Bille 2005] Philip Bille. *A survey on tree edit distance and related problems*. Theor. Comput. Sci., vol. 337, no. 1-3, pages 217–239, June 2005. 161

[Blum 1975] Lenore Blum and Manuel Blum. *Toward a Mathematical Theory of Inductive Inference*. Information and Control, vol. 28, no. 2, pages 125–155, 1975. 20

[Boiret 2012] Adrien Boiret, Aurélien Lemay and Joachim Niehren. *Learning Rational Functions*. In 16th International Conference on Developments of Language Theory, Taipee, Taïwan, Province De Chine, 2012. 208

[Boytcheva 2000] Svetla Boytcheva. *Least Generalization under Relative Implication*. In Proceedings of the 9th International Conference on Artificial Intelligence : Methodology, Systems, and Applications, AIMSA '00, pages 59–68, London, UK, UK, 2000. Springer-Verlag. 48, 59

[Boytcheva 2002] Svetla Boytcheva and Zdravko Markov. *An Algorithm for Inducing Least Generalization Under Relative Implication*. In Proceedings of the Fifteenth International Florida Artificial Intelligence Research Society Conference, pages 322–326. AAAI Press, 2002. 59

[Bray 2006] Tim Bray, Jean Paoli, C. M. Sperberg-McQueen, Eve Maler and François Yergeau. *Extensible Markup Language (XML) 1.0 (Fourth Edition)*, November 2006. 2, 150

[Bray 2008] Tim Bray, Jean Paoli, C. M. Sperberg-McQueen, Eve Maler and François Yergeau. *Extensible Markup Language (XML) 1.0 (Fifth Edition)*, November 2008. 1, 25, 33

[Breiman 1992] Leo Breiman and Philip Spector. *Submodel Selection and Evaluation in Regression. The X-Random Case.* International Statistical Review, vol. 60, no. 3, 1992. 42

[Buntine 1988] Wray Buntine. *Generalized subsumption and its applications to induction and redundancy.* Artificial Intelligence, vol. 36, no. 2, pages 149 – 176, 1988. 61, 80

[C Garriga 2012] Gemma C Garriga, Roni Khardon and Luc De Raedt. *Mining Closed Patterns in Relational, Graph and Network Data.* Annals of Mathematics and Artificial Intelligence, November 2012. 53

[Camacho 2004] Rui Camacho. *IndLog — Induction in Logic.* In JóseJúlio Alferes and João Leite, editeurs, Logics in Artificial Intelligence, volume 3229 of *Lecture Notes in Computer Science*, pages 718–721. Springer Berlin Heidelberg, 2004. 205, 206

[Caralp 2013] Mathieu Caralp, Emmanuel Filiot, Pierre-Alain Reynier, Frédéric Servais and Marc Jean-Talbot. *Expressiveness of Visibly Pushdown Transducers.* 2nd International Workshop on Trends in Tree Automata and Tree Transducers, 2013. 151, 208

[Ceri 1989] S. Ceri, G. Gottlob and L. Tanca. *What You Always Wanted to Know About Datalog (And Never Dared to Ask).* IEEE Trans. on Knowl. and Data Eng., vol. 1, no. 1, pages 146–166, march 1989. 15

[Chartrand 1977] Gary Chartrand. Introductory graph theory. Dover Publications, Inc., New York, NY, USA, 1977. 33

[Church 1936] Alonzo Church. *A Note on the Entscheidungsproblem.* J. Symb. Log., vol. 1, no. 1, pages 40–41, 1936. 13

[Comon 1997] H. Comon, M. Dauchet, R. Gilleron, C. Löding, F. Jacquemard, D. Lugiez, S. Tison and M. Tommasi. *Tree Automata Techniques and Applications.* Available on : `http://www.grappa.univ-lille3.fr/tata`, 1997. release 2007. 31, 71, 210

[Cornuéjols 2010] Antoine Cornuéjols and Laurent Miclet. Apprentissage artificiel : Concepts et algorithmes. Eyrolles, June 2010. 41

[Cornuéjols 2003] A. Cornuéjols and L. Miclet. Apprentissage artificiel - concepts et algorithmes. Eyrolles Editions, 2003. 7, 36, 42

[de la Higuera 2010] Colin de la Higuera. Grammatical inference : Learning automata and grammars. Cambridge University Press, New York, NY, USA, 2010. 39

[De Raedt 1992] Luc De Raedt. Interactive theory revision : an inductive logic programming approach. Academic Press Ltd., London, UK, UK, 1992. 39

[Decoster 2010] Jean Decoster, Sławomir Staworko and Fabien Torre. *Apprentissage relationnel polynomial pour la classification d'arbres.* In Engelbert Mephu Nguifo, editeur, 12ème Conférence francophone sur l'Apprentissage automatique (CAp'2010), pages 189–200, Clermont-Ferrand, may 2010. PUG. 57

[Demaine 2009] Erik D. Demaine, Shay Mozes, Benjamin Rossman and Oren Weimann. *An optimal decomposition algorithm for tree edit distance.* ACM Trans. Algorithms, vol. 6, no. 1, pages 2 :1–2 :19, December 2009. 161, 184, 204

[Douar 2011] Brahim Douar, Michel Liquiere, Cherif Chiraz Latiri and Yahya Slimani. *FG-MAC : Frequent subgraph mining with Arc Consistency.* In CIDM, pages 112–119. IEEE, 2011. 54, 56, 147

[Douar 2012] Brahim Douar, Michel Liquiere, Chiraz Latiri and Yahya Slimani. *Graph-Based relational learning with a polynomial time projection algorithm.* In Proceedings of the 21st international conference on Inductive Logic Programming, ILP'11, pages 98–112, Berlin, Heidelberg, 2012. Springer-Verlag. 53, 54, 147

[Duboc 2009] AnaLuísa Duboc, Aline Paes and Gerson Zaverucha. *Using the bottom clause and mode declarations in FOL theory revision from examples.* Machine Learning, vol. 76, no. 1, pages 73–107, 2009. 179

[E.F. Moore 1956] E.F. Moore. *Gedanken-Experiments on Sequential Machines.* In C.E. Shannon and J. MacCarthy, editeurs, Automata Studies, pages 129–153. Princeton University Press, 1956. 207

[Elgot 1965] C. C. Elgot and J. E. Mezei. *On relations defined by generalized finite automata.* IBM J. Res. Dev., vol. 9, no. 1, pages 47–68, January 1965. 208

[Engelfriet 1977] Joost Engelfriet. *Top-down Tree Transducers with Regular Look-ahead.* Mathematical Systems Theory, vol. 10, pages 289–303, 1977. 209

[Esposito 1994] Floriana Esposito, Donato Malerba, Giovanni Semeraro, Clifford Brunk and Michael Pazzani. *Traps and pitfalls when learning logical definitions from relations.* In ZbigniewW. Raś and Maria Zemankova, editeurs, Methodologies for Intelligent Systems, volume 869 of *Lecture Notes in Computer Science*, pages 376–385. Springer Berlin Heidelberg, 1994. 52

[Esposito 1996] Floriana Esposito, Angela Laterza, Donato Malerba and Giovanni Semeraro. *Refinement of Datalog Programs*, 1996. 15, 52, 62

[Esposito 2000] Floriana Esposito, Giovanni Semeraro, Nicola Fanizzi and Stefano Ferilli. *Multistrategy Theory Revision : Induction and Abduction in INTHELEX.* Machine Learning, vol. 38, no. 1-2, pages 133–156, 2000. 57

[Esposito 2004] F. Esposito, S. Ferilli, N. Fanizzi, T. M. A. Basile and N. Di Mauro. *Incremental learning and concept drift in INTHELEX.* Intell. Data Anal., vol. 8, no. 3, pages 213–237, August 2004. 62

[Faroult 2006] Stephane Faroult. The art of sql. O'Reilly Media, 1 édition, 2006. 73

[Ferilli 2002] Stefano Ferilli, Nicola Fanizzi, Nicola Di Mauro and Teresa M. A. Basile. *Efficient theta-Subsumption under Object Identity.* In In Atti del Workshop AI*IA su Apprendimento Automatico, 2002. 52, 53, 145

[Filiot 2012] Emmanuel Filiot and Frédéric Servais. *Visibly Pushdown Transducers with Look-Ahead.* In Mária Bieliková, Gerhard Friedrich, Georg Gottlob, Stefan Katzenbeisser and György Turán, editeurs, SOFSEM, volume 7147 of *Lecture Notes in Computer Science*, pages 251–263. Springer, 2012. 151, 208

[Freund 1997] Yoav Freund and Robert E. Schapire. *A Decision-Theoretic Generalization of on-Line Learning and an Application to Boosting*, 1997. 136

[Freund 1999] Yoav Freund and Robert E. Schapire. *A Short Introduction to Boosting*, 1999. 136

[Geibel 1997] Peter Geibel and Fritz Wysotzki. *A Logical Framework for Graph Theoretical Decision Tree Learning.* In In International Workshop on Inductive Logic Programming, pages 173–180. Springer. LNAI, 1997. 3, 52, 61

[Gold 1967] E. Mark Gold. *Language identification in the limit.* Information and Control, vol. 10, no. 5, pages 447–474, 1967. 4, 5, 6, 25, 39, 70, 106

[Gottlob 1987] Georg Gottlob. *Subsumption and implication.* Inf. Process. Lett., vol. 24, no. 2, pages 109–111, January 1987. 50

[Gottlob 1993] Georg Gottlob and Christian G. Fermüller. *Removing redundancy from a clause.* Artificial Intelligence, vol. 61, no. 2, pages 263–289, 1993. 51

[Gottlob 2003] Georg Gottlob and Christos H. Papadimitriou. *On the complexity of single-rule datalog queries.* Inf. Comput., vol. 183, no. 1, pages 104–122, 2003. 48

[Guyon 2003] Isabelle Guyon and André Elisseeff. *An introduction to variable and feature selection.* J. Mach. Learn. Res., vol. 3, pages 1157–1182, March 2003. 41

[Hamming 1950] R. W. Hamming. *Error detecting and error correcting codes.* Bell System Technical Journal, vol. 29, no. 2, pages 147–160, 1950. 183

[Hart 1968] Peter E. Hart, Nils J. Nilsson and Bertram Raphael. *A formal basis for the heuristic determination of minimum cost paths.* IEEE Transactions on Systems Science and Cybernetics, vol. SSC-4(2), pages 100–107, 1968. 181

[Haussler 1989] David Haussler. *Learning Conjunctive Concepts in Structural Domains.* Machine Learning, vol. 4, no. 1, pages 7–40, 1989. 62

[Heijenoort 1967] J. Van Heijenoort. From frege to gödel—a source book in mathematical logic. Harvard University Press, 1967. 2, 7

[Hell 2004] P. Hell and J. Nešetřil. Graphs and homomorphisms, volume 23. Oxford University Press, 2004. 33

[Hernández-orallo 1999] J. Hernández-orallo and M. J. Ramírez-quintana. *A Strong Complete Schema for Inductive Functional Logic Programming.* In in Dzeroski, S.; Flach, P. (eds) "Inductive Logic Programming" Lecture Notes in Artificial Intelligence (LNAI) series, pages 116–127. Springer, 1999. 5, 157, 207, 213

[Hirata 1999] Kouichi Hirata. *Flattening and implication.* In in : Proc. 10th Internat. Conf. on Algorithmic Learning Theory, LNAI 1720, pages 157–168. Springer, 1999. 22, 23

[Hirschberg 1975] D. S. Hirschberg. *A linear space algorithm for computing maximal common subsequences.* Commun. ACM, vol. 18, no. 6, pages 341–343, June 1975. 183

[Horváth 2010] Tamás Horváth, Gerhard Paass, Frank Reichartz and Stefan Wrobel. *A Logic-Based Approach to Relation Extraction from Texts.* In Luc Raedt, editeur, Inductive Logic Programming, volume 5989 of *Lecture Notes in Computer Science*, pages 34–48. Springer Berlin Heidelberg, 2010. 146

[hwei Nienhuys-cheng 1995] Shan hwei Nienhuys-cheng and Ronald de Wolf. *The Subsumption Theorem in Inductive Logic Programming : Facts and Fallacies.* In Advances in Inductive Logic Programming. IOS, pages 265–276. Press, 1995. 19

[Karp 1972a] R. Karp. *Reducibility among combinatorial problems.* In R. Miller and J. Thatcher, editeurs, Complexity of Computer Computations, pages 85–103. Plenum Press, 1972. 173

[Karp 1972b] Richard M. Karp. *Reducibility Among Combinatorial Problems.* In R. E. Miller and J. W. Thatcher, editeurs, Complexity of Computer Computations, pages 85–103. Plenum, NY, 1972. 50

[Kietz 1992] Jörg-Uwe Kietz and Stefan Wrobel. *Controlling the Complexity of Learning in Logic through Syntactic and Task-Oriented Models.* In INDUCTIVE LOGIC PROGRAMMING, pages 335–359. Academic Press, 1992. 39

[Kietz 1993] Jörg-Uwe Kietz. *A Comparative Study of Structural Most Specific Generalisations Used In Machine Learning.* In ILP '93 : Proceedings of the Third International Workshop on Inductive Logic Programming, pages 149–164, 1993. 128

[Kietz 1994a] J-U. Kietz and M. Lübbe. *An Efficient Subsumption Algorithm for Inductive Logic Programming.* In W. W. Cohen and H. Hirsh, editeurs, Proceedings 11th

International Conference on Machine Learning, pages 130–138. Morgan Kaufmann, 1994. 3, 37, 50, 64, 65, 70, 77, 86, 105, 143, 153

[Kietz 1994b] Jörg-Uwe Kietz and Sašo Džeroski. *Inductive logic programming and learnability.* SIGART Bull., vol. 5, no. 1, pages 22–32, January 1994. 37, 38, 49, 50

[King 1995] Ross D. King, Ashwin Srinivasan and Michael J. E. Sternberg. *Relating chemical activity to structure : an examination of ILP successes*, 1995. 14, 54

[King 2001] Ross D. King, Ashwin Srinivasan and Luc Dehaspe. *Warmr : a data mining tool for chemical data.* Journal of Computer-Aided Molecular Design, vol. 15, no. 2, pages 173–181, 2001. 205, 206

[Koenig 2004] Sven Koenig, Maxim Likhachev, Yaxin Liu and David Furcy. *Incremental heuristic search in AI.* AI Mag., vol. 25, no. 2, pages 99–112, June 2004. 181

[Kohavi 1995] Ron Kohavi. *A Study of Cross-Validation and Bootstrap for Accuracy Estimation and Model Selection.* pages 1137–1143. Morgan Kaufmann, 1995. 42

[Kowalski 1970] Robert Kowalski. *The case for using equality axioms in automatic demonstration.* In M. Laudet, D. Lacombe, L. Nolin and M. Schützenberger, editeurs, Symposium on Automatic Demonstration, volume 125 of *Lecture Notes in Mathematics*, pages 112–127. Springer Berlin / Heidelberg, 1970. 10.1007/BFb0060628. 19

[Kowalski 1971] R. Kowalski and D. Kuehner. *Linear Resolution with Selection Function.* Artificial Intelligence, vol. 2, 1971. 176

[Kowalski 1974] Robert Kowalski. *Predicate Logic as Programming Language.* In IFIP Congress, pages 569–574, 1974. 7, 9

[Krogel 2003] Mark-A. Krogel, Simon Rawles, Filip Zelezny, Peter A. Flach, Nada Lavrac and Stefan Wrobel. *Comparative Evaluation of Approaches to Propositionalization*, 2003. 21

[Kuželka 2009] Ondřej Kuželka and Filip Železný. *A Restarted Strategy for Efficient Subsumption Testing.* Fundam. Inf., vol. 89, no. 1, pages 95–109, January 2009. 51

[Kuwabara 2005] Megumi Kuwabara, Takeshi Ogawa, Kouichi Hirata and Masateru Harao. *On Generalization and Subsumption for Ordered Clauses.* In Takashi Washio, Akito Sakurai, Katsuto Nakajima, Hideaki Takeda, Satoshi Tojo and Makoto Yokoo, editeurs, JSAI Workshops, volume 4012 of *Lecture Notes in Computer Science*, pages 212–223. Springer, 2005. 144

[Kuznetsov 2005] SergeiO. Kuznetsov and MikhailV. Samokhin. *Learning Closed Sets of Labeled Graphs for Chemical Applications.* In Stefan Kramer and Bernhard Pfahringer, editeurs, Inductive Logic Programming, volume 3625 of *Lecture Notes in Computer Science*, pages 190–208. Springer Berlin Heidelberg, 2005. 53

[Kuželka 2013] Ondřej Kuželka, Andrea Szabóová and Filip Železný. *Bounded Least General Generalization.* In Fabrizio Riguzzi and Filip Železný, editeurs, Inductive Logic Programming, volume 7842 of *Lecture Notes in Computer Science*, pages 116–129. Springer Berlin Heidelberg, 2013. 146

[Larson 1977] J. Larson and R. S. Michalski. *Inductive inference of VL decision rules.* SIGART Bull., no. 63, pages 38–44, June 1977. 14

[Lavrac 1991] Nada Lavrac, Saso Dzeroski and Marko Grobelnik. *Learning Nonrecursive Definitions of Relations with LINUS.* In Proceedings of the European Working Session on Machine Learning, EWSL '91, pages 265–281, London, UK, UK, 1991. Springer-Verlag. 205, 206

[Lee 1967] Char-Tung Lee. *A completeness theorem and a computer program for finding theorems derivable from given axioms.* PhD thesis, 1967. AAI6810359. 19

[Lee 2004] Sau Dan Lee and Luc De Raedt. *Constraint based mining of first order sequences in SeqLog.* In In Database Support for Data Mining Applications : Discovering Knowledge with Inductive Queries, pages 154–173. Springer, 2004. 144

[Lemay 2010] Aurélien Lemay, Sebastian Maneth and Joachim Niehren. *A learning algorithm for top-down XML transformations.* In Jan Paredaens and Dirk Van Gucht, editeurs, PODS, pages 285–296. ACM, 2010. 209

[Levenshtein 1966] V. Levenshtein. *Binary Codes Capable of Correcting Deletions, Insertions, and Reversals.* Soviet Physics Doklady, vol. 10, no. 8, pages 707–710, 1966. 183

[Liquiere 1998] Michel Liquiere and Jean Sallantin. *Structural machine learning with Galois lattice and Graphs.* In Proc. of the 1998 Int. Conf. on Machine Learning (ICML'98, pages 305–313. Morgan Kaufmann, 1998. 63

[Liquiere 2007] Michel Liquiere. *Arc Consistency Projection : A New Generalization Relation for Graphs.* In Uta Priss, Simon Polovina and Richard Hill, editeurs, ICCS, volume 4604 of *Lecture Notes in Comput. Sci.*, pages 333–346. Springer, 2007. 54, 56, 147

[Lloyd 1987] John W. Lloyd. Foundations of logic programming, 2nd edition. Springer, 1987. 7, 11

[Malerba 2001] Donato Malerba and FrancescaA. Lisi. *Discovering Associations between Spatial Objects : An ILP Application.* In Céline Rouveirol and Michéle Sebag, editeurs, Inductive Logic Programming, volume 2157 of *Lecture Notes in Computer Science*, pages 156–163. Springer Berlin Heidelberg, 2001. 3, 53

[Maletti 2009] Andreas Maletti, Jonathan Graehl, Mark Hopkins and Kevin Knight. *The Power of Extended Top-Down Tree Transducers.* SIAM J. Comput., vol. 39, no. 2, pages 410–430, June 2009. 151, 210

[Maloberti 2001] Jérôme Maloberti and Michèle Sebag. *Theta-Subsumption in a Constraint Satisfaction Perspective.* In Proceedings of the 11th International Conference on Inductive Logic Programming, ILP '01, pages 164–178, London, UK, UK, 2001. Springer-Verlag. 146

[Maloberti 2004] Jérôme Maloberti and Michèle Sebag. *Fast Theta-Subsumption with Constraint Satisfaction Algorithms.* Machine Learning, vol. 55, no. 2, pages 137–174, 2004. 51

[Marcinkowski 1992] Jerzy Marcinkowski and Leszek Pacholski. *Undecidability of the Horn-Clause Implication Problem.* In FOCS, pages 354–362. IEEE, 1992. 3, 47

[Mealy 1955] George H. Mealy. *A Method for Synthesizing Sequential Circuits.* Bell System Technical Journal, vol. 34, no. 5, pages 1045–1079, 1955. 207

[Mitchell 1982] Tom M. Mitchell. *Generalization as Search.* Artif. Intell., vol. 18, no. 2, pages 203–226, 1982. 37

[Mitchell 1997] Tom M. Mitchell. Machine learning. McGraw-Hill, New York, 1997. 36, 38

[Morik 1999] Katharina Morik. *Tailoring Representations to Different Requirements.* In In Proceedings of the 10th International Conference on Algorithmic Learning Theory (ALT, pages 1–12, 1999. 66

[Muggleton 1988] Stephen Muggleton and Wray L. Buntine. *Machine Invention of First Order Predicates by Inverting Resolution.* In John E. Laird, editeur, ML, pages 339–352. Morgan Kaufmann, 1988. 39

[Muggleton 1990] Stephen Muggleton and Cao Feng. *Efficient Induction Of Logic Programs*. In New Generation Computing. Academic Press, 1990. 3, 11, 20, 39, 51, 52, 60, 64, 66, 70, 77, 143, 153, 206

[Muggleton 1991] Stephen Muggleton. *Inductive Logic Programming*. New Generation Computing Journal, vol. 8, no. 4, pages 295–318, 1991. 2, 3, 5, 25, 37, 70

[Muggleton 1992] Stephen Muggleton. *Inverting Implication*. In Artificial Intelligence Journal, 1992. 50

[Muggleton 1994a] S. Muggleton and C.D. Page. *Self-saturation of Definite Clauses*. In S. Wrobel, editeur, ILP94, volume 237 of *GMD-Studien*, pages 161–174. Gesellschaft für Mathematik und Datenverarbeitung MBH, 1994. 58

[Muggleton 1994b] Stephen Muggleton and Luc De Raedt. *Inductive Logic Programming : Theory and Methods*. JOURNAL OF LOGIC PROGRAMMING, vol. 19, no. 20, pages 629–679, 1994. 2, 3, 37, 38, 46, 70, 152

[Muggleton 1995] Stephen Muggleton. *Inverse entailment and Progol*, 1995. 3, 39, 153, 179, 205, 206

[Muggleton 2010] Stephen Muggleton, José Santos and Alireza Tamaddoni-Nezhad. *ProGolem : A System Based on Relative Minimal Generalisation*. In Luc Raedt, editeur, Inductive Logic Programming, volume 5989 of *Lecture Notes in Computer Science*, pages 131–148. Springer Berlin Heidelberg, 2010. 3, 39, 51, 68, 153, 179, 206

[Neven 2002] Frank Neven. *Automata theory for XML researchers*. SIGMOD Rec., vol. 31, no. 3, pages 39–46, September 2002. 34

[Niblett 1988] Tim Niblett. *A Study of Generalisation in Logic Programs*. In EWSL, pages 131–138, 1988. 60

[Nienhuys-Cheng 1996a] S-H. Nienhuys-Cheng and R. de Wolf. *The Subsumption Theorem in Inductive Logic Programming : Facts and Fallacies*. In L. De Raedt, editeur, Advances in Inductive Logic Programming, pages 265–276. IOS, 1996. 48

[Nienhuys-Cheng 1996b] Shan-Hwei Nienhuys-Cheng and Ronald de Wolf. *Least Generalizations and Greatest Specializations of Sets of Clauses*. J. Artif. Intell. Res. (JAIR), vol. 4, pages 341–363, 1996. 47, 48, 58

[Nienhuys-Cheng 1997a] Shan-Hwei Nienhuys-Cheng and Ronald de Wolf. Foundations of inductive logic programming. Springer-Verlag New York, Inc., Secaucus, NJ, USA, 1997. 11, 18, 22, 23

[Nienhuys-Cheng 1997b] Shan-Hwei Nienhuys-Cheng and Ronald de Wolf. Foundations of inductive logic programming. Springer-Verlag New York, Inc., Secaucus, NJ, USA, 1997. 60

[Nijssen 2003] Siegfried Nijssen and Joost N. Kok. *Efficient frequent query discovery in FARMER*. In In Proc. of the 7th PKDD, volume 2838 of LNCS, pages 350–362. Springer, 2003. 53

[Ohlbach 1998] Hans Jürgen Ohlbach. *Elimination of Self-Resolving Clauses*, 1998. 23

[Pankowski 2003] Tadeusz Pankowski. *Transformation of XML data using an unranked tree transducer*. In In : E-Commerce and Web Technologies, 4th International Conference, EC-Web, pages 259–269, 2003. 151

[Paulson 1991] Lawrence C. Paulson, Lawrence C. Paulson, Andrew W. Smith and Andrew W. Smith. *Logic Programming, Functional Programming, and Inductive Definitions*. In In Extensions of Logic Programming, volume 475 of LNCS, pages 283–310. Springer-Verlag, 1991. 5, 157, 207, 213

[Pawlik 2011] Mateusz Pawlik and Nikolaus Augsten. *RTED : a robust algorithm for the tree edit distance*. Proc. VLDB Endow., vol. 5, no. 4, pages 334–345, December 2011. 161, 184, 204

[Plotkin 1970] Gordon D. Plotkin. *A note on inductive generalization*. Machine Intelligence, vol. 5, pages 153–163, 1970. 3, 49, 51, 59, 60, 70, 74, 81, 95, 142, 144

[Plotkin 1971a] G.D. Plotkin. *Automatic Methods of Inductive Inference*. PhD thesis, Edinburgh University, 1971. 59, 60, 61, 74

[Plotkin 1971b] Gordon D. Plotkin. *A further note on inductive generalization*. Machine Intelligence, vol. 6, pages 101–124, 1971. 51, 59, 60, 61, 74

[Porter 1990] Bruce W. Porter and Raymond J. Mooney, editeurs. Machine learning, proceedings of the seventh international conference on machine learning, austin, texas, usa, june 21-23, 1990. Morgan Kaufmann, 1990. 21

[Quinlan 1990] J. R. Quinlan. *Learning logical definitions from relations*. MACHINE LEARNING, vol. 5, pages 239–266, 1990. 5, 39, 206, 207, 213

[Quinlan 1991] J. R. Quinlan. *Determinate literals in inductive logic programming*. In Proceedings of the 12th international joint conference on Artificial intelligence - Volume 2, IJCAI'91, pages 746–750, San Francisco, CA, USA, 1991. Morgan Kaufmann Publishers Inc. 3, 64, 70, 77, 153

[Quinlan 1993] J. R. Quinlan and R. M. Cameron-jones. *FOIL : A Midterm Report*. In In Proceedings of the European Conference on Machine Learning, pages 3–20. Springer-Verlag, 1993. 5, 39, 206, 207, 213

[Quinlan 1995] J. R. Quinlan and R. M. Cameron-jones. *Induction of Logic Programs : FOIL and Related Systems*. New Generation Computing, vol. 13, pages 287–312, 1995. 5, 39, 205, 206, 207, 213

[Quinlan 1996] J. R. Quinlan. *Boosting First-Order Learning*. In Lecture Notes in Computer Science, pages 143–155. Springer, 1996. 3, 5, 157, 207, 213

[Rabin 1969] Michael O. Rabin. *Decidability of Second Order Theories and Automata on Infinite Trees*. Transactions of the American Mathematical Society, vol. 141, pages 1–35, 1969. 31, 71

[Raedt 1996] Luc De Raedt, Peter Idestam-Amquist and Gunther Sablon. *Theta-subsumption for structural matching*, 1996. 51

[Raedt 1997] Luc De Raedt and Luc Dehaspe. *Clausal Discovery*. In Machine Learning, pages 1058–1063. Morgan Kaufmann, 1997. 39, 67

[Raskin 2008] Jean-François Raskin and Frédéric Servais. *Visibly Pushdown Transducers*. In Luca Aceto, Ivan Damgård, Leslie Ann Goldberg, Magnús M. Halldórsson, Anna Ingólfsdóttir and Igor Walukiewicz, editeurs, ICALP (2), volume 5126 of *Lecture Notes in Computer Science*, pages 386–397. Springer, 2008. 208

[Ray 2008] Oliver Ray and Katsumi Inoue. *Mode-directed inverse entailment for full clausal theories*. In Proceedings of the 17th international conference on Inductive logic programming, ILP'07, pages 225–238, Berlin, Heidelberg, 2008. Springer-Verlag. 67

[Razmara 2011] Majid Razmara. *Application of Tree Transducers in Statistical Machine Translation*. Rapport technique, Simon Fraser University, 2011. 151, 210

[Rifkin 2004] Ryan Rifkin and Aldebaro Klautau. *In Defense of One-Vs-All Classification*. J. Mach. Learn. Res., vol. 5, pages 101–141, December 2004. 43

[Rijsbergen 1979] C. J. Van Rijsbergen. Information retrieval. Butterworth-Heinemann, Newton, MA, USA, 2nd édition, 1979. 43

[Robinson 1965] J. A. Robinson. *A Machine-Oriented Logic Based on the Resolution Principle.* J. ACM, vol. 12, no. 1, pages 23–41, January 1965. 15, 49

[Rodrigues 2013] Christophe Rodrigues. *Apprentissage incrémental de modèles d'action relationnels.* PhD thesis, Université Paris 13, LIPN, 2013. 62

[Rounds 1970] William C. Rounds. *Mappings and Grammars on Trees.* Mathematical Systems Theory, vol. 4, no. 3, pages 257–287, 1970. 207, 209

[Rouveirol 1992] C. Rouveirol. *Extensions of Inversion of Resolution Applied to Theory Completion.* In S. Muggleton, editeur, Inductive Logic Programming, pages 63–92. Academic Press, London, 1992. 21, 178, 179

[Rouveirol 1994] Céline Rouveirol. *Flattening and Saturation : Two Representation Changes for Generalization.* Machine Learning, vol. 14, pages 219–232, 1994. 21, 22, 178, 179

[Santos 2003] Vitor Santos, Nuno Fonseca, Nuno Fonseca, O Silva, O Silva, Rui Camacho, Rui Camacho and Vitor Santos Costa. *Induction with April - A preliminary report.* Rapport technique, DCC-FC & LIACC, Universidade do, 2003. 3, 153, 155

[Santos 2009] JoséC.A. Santos, Alireza Tamaddoni-Nezhad and Stephen Muggleton. *An ILP System for Learning Head Output Connected Predicates.* In LuísSeabra Lopes, Nuno Lau, Pedro Mariano and LuísM. Rocha, editeurs, Progress in Artificial Intelligence, volume 5816 of *Lecture Notes in Computer Science*, pages 150–159. Springer Berlin Heidelberg, 2009. 3, 5, 153, 206, 213

[Santos 2010] Jose Santos and Stephen Muggleton. *Subsumer : A Prolog theta-subsumption engine.* In Manuel Hermenegildo and Torsten Schaub, editeurs, Technical Communications of the 26th International Conference on Logic Programming, volume 7 of *Leibniz International Proceedings in Informatics (LIPIcs)*, pages 172–181, Dagstuhl, Germany, 2010. Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik. 51

[Santos 2011] José Santos and Stephen Muggleton. *When Does It Pay Off to Use Sophisticated Entailment Engines in ILP ?* In Paolo Frasconi and FrancescaA. Lisi, editeurs, Inductive Logic Programming, volume 6489 of *Lecture Notes in Computer Science*, pages 214–221. Springer Berlin Heidelberg, 2011. 176

[Schaefer 1978] Thomas J. Schaefer. *The complexity of satisfiability problems.* In Proceedings of the tenth annual ACM symposium on Theory of computing, STOC '78, pages 216–226, New York, NY, USA, 1978. ACM. 115, 173

[Schmidt-Schauß 1988] Manfred Schmidt-Schauß. *Implication of Clauses is Undecidable.* Theor. Comput. Sci., vol. 59, pages 287–296, 1988. 3, 47

[Schützenberger 1975] Marcel Paul Schützenberger. *Sur les relations rationnelles.* In H. Barkhage, editeur, Automata Theory and Formal Languages, volume 33 of *Lecture Notes in Computer Science*, pages 209–213. Springer, 1975. 208

[Sebag 1997a] Michèle Sebag and Céline Rouveirol. *Subsomption Stochastique et Apprentissage Polynomial en Logique des Prédicats*, 1997. 53

[Sebag 1997b] Michèle Sebag and Céline Rouveirol. *Tractable Induction and Classification in First Order Logic Via Stochastic Matching*, 1997. 47, 53, 54

[Selkow 1977] Stanley M. Selkow. *The Tree-to-Tree Editing Problem.* Inf. Process. Lett., vol. 6, no. 6, pages 184–186, 1977. 205, 213

[Semeraro 1994] Giovanni Semeraro, Floriana Esposito, Donato Malerba, Clifford Brunk and Michael J. Pazzani. *Avoiding Non-Termination when Learning Logical Programs : A Case Study with FOIL and FOCL.* In LOPSTR '94/META '94 : Proceedings of the 4th International Workshops on Logic Programming Synthesis and

Transformation - Meta-Programming in Logic, pages 183–198, London, UK, 1994. Springer-Verlag. 3, 52, 53, 61, 62

[Semeraro 1996] Giovanni Semeraro, Floriana Esposito and Donato Malerba. *Ideal refinement of Datalog programs.* In Maurizio Proietti, editeur, Logic Program Synthesis and Transformation, volume 1048 of *Lecture Notes in Computer Science*, pages 120–136. Springer Berlin Heidelberg, 1996. 53

[Semeraro 1998] Giovanni Semeraro, Floriana Esposito, Donato Malerba, Nicola Fanizzi and Stefano Ferilli. *A Logic Framework for the Incremental Inductive Synthesis of Datalog Theories.* In LOPSTR '97 : Proceedings of the 7th International Workshop on Logic Programming Synthesis and Transformation, pages 300–321, London, UK, 1998. Springer-Verlag. 15, 52, 57

[Shapiro 1981] Ehud Y. Shapiro. *The model inference system.* In Proceedings of the 7th international joint conference on Artificial intelligence - Volume 2, IJCAI'81, pages 1064–1064, San Francisco, CA, USA, 1981. Morgan Kaufmann Publishers Inc. 20

[Shapiro 1983] Ehud Y. Shapiro. Algorithmic program debugging. MIT Press, Cambridge, MA, USA, 1983. 67

[Shapiro 1991] Ehud Y. Shapiro. *Inductive Inference of Theories from Facts.* In Computational Logic - Essays in Honor of Alan Robinson, pages 199–254, 1991. 51

[Srinivasan 2004] A. Srinivasan. *The Aleph Manual*, 2004. http ://www.comlab.ox.ac.uk/activities/machinelearning/Tang2003h/. 3, 39, 46, 153, 154, 179, 205, 206

[Stolle 2005] Christian Stolle, Andreas Karwath and Luc De Raedt. *CLASSIC'CL : an integrated ILP system.* In Proceedings of the 8th International Conference of Discovery Science, volume 3735 of *Lecture Notes in Artificial Intelligence*, pages 354–362, 2005. URL : http ://www.cs.kuleuven.ac.be/cgi-bin-dtai/publ_info.pl ?id=42979. 67

[Suciu 2002] Dan Suciu. *Typechecking for Semistructured Data.* In Giorgio Ghelli and Gösta Grahne, editeurs, Database Programming Languages, volume 2397 of *Lecture Notes in Computer Science*, pages 1–20. Springer Berlin Heidelberg, 2002. 34

[Tang 2003] Lap Poon Rupert Tang. *Integrating top-down and bottom-up approaches in inductive logic programming : applications in natural language processing and relational data mining.* PhD thesis, The University of Texas at Austin, 2003. AAI3122800. 179

[Tarski 1956] Alfred Tarski. *Logic, semantics, metamathematics : papers from 1923 to 1938.* 1956. 2, 7, 11

[Thatcher 1970] James W. Thatcher. *Generalized2 sequential machine maps.* J. Comput. Syst. Sci., vol. 4, no. 4, pages 339–367, August 1970. 207, 209

[Thomas 2005] Bernd Thomas. *Machine learning of information extraction procedures : an ILP approach.* PhD thesis, Universität Koblenz-Landau, Institut für Informatik, 2005. 143

[Torre 1999] F. Torre. *GloBo : un algorithme stochastique pour l'apprentissage supervisé et non-supervisé.* In M. Sebag, editeur, Actes de la Première Conférence d'Apprentissage, pages 161–168, 1999. 136

[Torre 2004] Fabien Torre. *GloBoost : Boosting de moindres généralisés.* In Michel Liquière et Marc Sebban, editeur, Actes de la Sixième Conférence Apprentissage CAp'2004, pages 49–64. Presses Universitaires de Grenoble, 2004. 135, 136

[Torre 2005] Fabien Torre. *GloBoost : Combinaisons de Moindres Généralisés.* Revue d'Intelligence Artificielle, vol. 19, no. 4-5, pages 769–797, 2005. 136

[Torre 2009] Fabien Torre and Alain Terlutte. *Méthodes d'ensemble en Inférence Grammaticale : une approche à base de moindres généralisés.* In Younès Bennani and Céline Rouveirol, editeurs, 11ème Conférence francophone sur l'Apprentissage automatique (CAp'2009), pages 33–48, Hammamet (Tunisie), may 2009. PUG. 136

[Tropashko 2005] Vadim Tropashko. *Nested intervals tree encoding in SQL.* SIGMOD Rec., vol. 34, no. 2, pages 47–52, June 2005. 73

[van der Laag 1995] P.R.J. van der Laag. *An analysis of refinement operators in inductive logic programming.* PhD thesis, Erasmus Universiteit, Rotterdam, the Netherlands, 1995. 144

[Van der Vlist 2002] Eric Van der Vlist. Xml schema - the w3c's object-oriented descriptions for xml. O'Reilly, 2002. 34, 150

[Van der Vlist 2004] Eric Van der Vlist. Relax ng : A simpler schema language for xml. O'Reilly, Beijing, 2004. 150

[Vries 2011] ChristopherM. Vries, Richi Nayak, Sangeetha Kutty, Shlomo Geva and Andrea Tagarelli. *Overview of the INEX 2010 XML Mining Track : Clustering and Classification of XML Documents.* In Shlomo Geva, Jaap Kamps, Ralf Schenkel and Andrew Trotman, editeurs, Comparative Evaluation of Focused Retrieval, volume 6932 of *Lecture Notes in Computer Science*, pages 363–376. Springer Berlin Heidelberg, 2011. 122

[W3C 1997] W3C. *The HyperText Markup Language (HTML) 4.0.* 1997. 34

[W3C 1999] W3C. *XML Path language (XPath) 1.0.* 1999. 35

[W3C 2000] W3C. *The Extensible HyperText Markup Language (XHTML) 1.0.* 2000. 35

[W3C 2002] W3C. *Really Simple Syndication (RSS) 2.0.* 2002. 35

[W3C 2007a] W3C. *XML Path language (XPath) 2.0.* 2007. 35

[W3C 2007b] W3C. *XML Query language (XQuery) 1.0.* 2007. 35

[W3C 2007c] W3C. *XSL Transformations (XSLT) 2.0.* 2007. 35

[W3C 2010] W3C. *The Extensible HyperText Markup Language (XHTML) 2.0.* 2010. 35

[W3C 2011] W3C. *The HyperText Markup Language (HTML) 5.0.* 2011. 34

[Webb 1991] G. I. Webb. *Einstein : An Interactive Inductive Knowledge-Acquisition Tool.* In Proceedings of the Sixth Banff Knowledge Acquisition for Knowledge-Based Systems Workshop, pages (36)1–16, 1991. 135

[Webb 1992] Geoffrey I Webb and John W. M Agar. *Inducing diagnostic rules for glomerular disease with the DLG machine learning algorithm.* Artif. Intell. Med., vol. 4, no. 6, pages 419–430, December 1992. 134

[Winston 1970] Patrick H. Winston. *Learning Structural Descriptions From Examples.* Rapport technique, Cambridge, MA, USA, 1970. 14

[Wu 2008] Xiaobing Wu. *An inductive learning system for XML documents.* In Proceedings of the 17th international conference on Inductive logic programming, ILP'07, pages 292–306, Berlin, Heidelberg, 2008. Springer-Verlag. 42

[Wu 2012] Jemma Wu. *A Framework for Learning Comprehensible Theories in XML Document Classification.* IEEE Trans. Knowl. Data Eng., vol. 24, no. 1, pages 1–14, 2012. 42

[Yan 2003] Xifeng Yan and Jiawei Han. *CloseGraph : mining closed frequent graph patterns.* In Proceedings of the ninth ACM SIGKDD international conference on Knowledge discovery and data mining, KDD '03, pages 286–295, New York, NY, USA, 2003. ACM. 53

[Zhang 1989] K. Zhang and D. Shasha. *Simple fast algorithms for the editing distance between trees and related problems.* SIAM J. Comput., vol. 18, no. 6, pages 1245–1262, December 1989. 161, 183