Apprentissage relationnel polynomial pour la classification d'arbres

Jean Decoster, Sławomir Staworko, Fabien Torre

Mostrare (INRIA Lille Nord Europe et CNRS LIFL)
Université Lille Nord de France

Résumé : Après avoir rappelé le cadre général de la programmation logique inductive, nous proposons une sous-famille des clauses de Horn nommée MQD. Visant des applications de classification de document XML, nous définissons un langage de clauses $\mathcal{L}_{fcns}$ permettant de représenter des arbres et des motifs d'arbres. Ce langage nous fournit exemples et hypothèses. Nous montrons que ce langage est inclus dans les MQD et proposons des algorithmes dédiés pour les opérations de base nécessaires à l'apprentissage, à savoir les calculs de θ-subsomption et de moindre généralisé. Nos algorithmes étant polynomiaux et non exponentiels comme dans le cas général des clauses de Horn (Plotkin, 1970), ils peuvent participer à la classification supervisée d'arbres, l'apprentissage relationnel devenant alors de complexité polynomiale.

Mots-clés : Programmation logique inductive, θ-subsomption polynomiale, moindre généralisé, clauses déterminées, codages d'arbres, classification d'arbres.

Introduction

Nous sommes motivés dans nos travaux par le traitement automatique des arbres, en particulier des documents XML et nous souhaitons *apprendre* à réaliser ces traitements. Sachant qu'un arbre peut facilement devenir un graphe (par exemple en le saturant de ses axes XPATH ou plus simplement en y ajoutant des liens d'égalité entre nœuds), nous nous intéressons aux représentations relationnelles et aux techniques d'apprentissage de la *programmation logique inductive* (PLI). Cependant, un arbre restant un graphe particulier nous espérons éviter la complexité exponentielle de la PLI classique. Plus formellement, nous sommes encouragés à chercher dans cette voie par les résultats de complexité polynomiale obtenus pour l'évaluation de requêtes monadiques Datalog portant sur des arbres (Gottlob *et al.*, 2002).

Dans cet article, nous nous concentrons sur la tâche de classification d'arbres, explorons leurs représentations logiques et proposons une réécriture plus efficace des opérations de base de l'apprentissage relationnel que sont la θ-subsomption et le calcul de moindre généralisé. Nous nous inscrivons dans une démarche que l'on pourrait désigner comme celle des *clauses déterminées* (Muggleton & Feng, 1990; Quinlan, 1991; Kietz & Lübbe, 1994). Dans ce cadre, nous proposons une nouvelle famille de

clauses, particulièrement adaptée à la représentation des arbres et permettant des gains de complexité algorithmique.

Le plan est le suivant. Section 1 nous rappelons le cadre de la PLI, ses opérations de base et leur complexité exponentielle. Puis, section 2, nous posons des contraintes sur les clauses de Horn qui simplifient les opérations qui nous intéressent pour l'apprentissage et permettent cette fois une complexité polynomiale. Nous montrons ensuite, section 3, que ces clauses particulières permettent de représenter des arbres et que, par suite, nous sommes capables d'apprendre à classer des documents XML en temps polynomial. Finalement, nous faisons le bilan de ce travail et en proposons des perspectives à la section 4.

1 Éléments de programmation logique inductive

Nous nous plaçons dans le cadre classique de la PLI (Muggleton, 1991) en utilisant des clauses de Horn pour représenter nos exemples et nos hypothèses durant l'apprentissage. Cependant, considérant que nous visons une tâche de classification, et par souci de simplicité, nous allons manipuler dans la suite des clauses sans tête. Nous représentons donc une clause comme l'ensemble des littéraux de son corps et négligeons le fait qu'ils sont négatifs. De plus, et toujours sans perte de généralité, nous limiterons les termes utilisés à des constantes et des variables, sans autre symbole de fonction.

Nous serons intéressés par la notion de *connexion*. Deux littéraux ℓ et ℓ' d'une clause sont dits *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 dite *connectée* si ses littéraux sont tous connectés deux à deux.

Nous allons supposer que les exemples comme les hypothèses manipulées proviennent d'un *langage*, c'est-à-dire d'un ensemble (éventuellement infini) de clauses. Typiquement, un tel langage est défini par un ensemble fini de symboles de prédicats, un ensemble de constantes et des contraintes syntaxiques supplémentaires pour préciser les compositions valides de termes au sein des littéraux.

Nous chercherons à apprendre avec comme opérations de base la θ-subsomption et le calcul de moindre généralisé, suivant en cela Plotkin (1970) dont les résultats sont exposés ci-dessous.

Définition 1 (θ-subsomption)
Une clause C θ-subsume une clause C' (noté $C \succeq_\theta C'$) si et seulement il existe une substitution θ telle que $C\theta \subseteq C'$.

Décider si une clause en θ-subsume une autre est un problème NP-complet et cela est visible sur le test de subsomption correspondant qui est donné à travers les algorithmes 1 (sub_atom, sub_clause et sub) : pour chaque littéral de la première clause il est possible d'avoir à considérer tous les littéraux de la seconde et donc, finalement, toutes les assignations possibles des littéraux de la première clause vers les littéraux de la seconde.

Beaucoup de travaux du domaine visent à contourner ce résultat négatif. En particulier, Muggleton & Feng (1990) comme Quinlan (1991) puis Kietz & Lübbe (1994) ont

Algorithm 1 θ-subsomption.

function sub_atom$(\ell, \ell', \textbf{var } \theta)$
let $\ell = R(t_1, \ldots, t_n)$;
let $\ell' = P(s_1, \ldots, s_m)$;
if $R \neq P$ **or** $n \neq m$ **then**
  return $false$;
$\theta' \leftarrow \theta$;
for $i \in \{1, \ldots, n\}$ **do**
  if $t_i = s_i$ **then**
    continue ;
  if $t_i = X$ **and** $\nexists v\ v/X \in \theta$ **then**
    $\theta \leftarrow \theta \cup \{s_i/X\}$;
  if $t_i = X$ **and** $s_i/X \in \theta$ **then**
    continue ;
  $\theta \leftarrow \theta'$;
  return $false$;
end for
return $true$;
end function

function sub_clause(C, C', θ)
if $C = \varnothing$ **then**
  return $true$;
choose $\ell \in C$;
for $\ell' \in C'$ **do**
  $\theta' \leftarrow \theta$;
  if sub_atom(ℓ, ℓ', θ') **and**
    sub_clause$(C \backslash \{\ell\}, C', \theta')$
  then return $true$;
end for
return $false$;
end function

function sub(C, C')
return sub_clause$(C, C', \varnothing)$;
end function

défini des notions de littéraux et de clauses *déterminées* ; ils ont montré que se restreindre à de telles clauses amène une complexité polynomiale pour le test de θ-subsomption.

Définition 2 (clause déterminée)
$C = (l_1, \cdots, l_n)$ est déterminée par rapport à C' ssi pour tout $1 \leqslant i \leqslant n$, si il existe une substitution θ telle que $\{l_j\}_{j<i}\theta \subseteq C'$ alors il existe au plus une substitution θ' compatible avec θ telle que $\{l_j\}_{j \leqslant i}\theta\theta' \subseteq C'$. On dit que C est déterminée par rapport à un langage $\mathcal{L}$ ssi elle est déterminée vis-à-vis de toute clause de $\mathcal{L}$.

Autrement dit, les littéraux des clauses sont ordonnés et l'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 C', étant donnés les choix faits pour les littéraux précédents. Par rapport à l'algorithme général de subsomption entre clauses de Horn, nous sommes donc dispensés de remettre nos choix en question. C'est ce que traduit l'algorithme 2 (sub_D) qui précise le test de subsomption dédié aux clauses déterminées.

La θ-subsomption, déterminée ou pas, permet de décider si une hypothèse couvre ou non un exemple, et également de savoir si une hypothèse est plus générale qu'une autre. Pour apprendre ces hypothèses, il nous faut maintenant un opérateur de généralisation.

Définition 3 (moindre généralisé entre clauses de Horn sous θ-subsomption)
Étant donné un ensemble de clauses $\mathscr{E}$, une clause G est dite moindre généralisée *de $\mathscr{E}$ si et seulement si :*
- *$\forall C_i \in \mathscr{E} : G \succeq_\theta C_i$;*
- *il n'existe pas G' strictement plus spécifique que G telle que : $\forall C_i \in \mathscr{E} : G' \succeq_\theta C_i$.*

Algorithm 2 θ-subsomption pour clauses déterminées.

function $\mathsf{sub}_{\mathsf{D}}(C, C', \theta)$
1: **while** $C \neq \varnothing$ **do**
2: $\quad \ell \leftarrow \mathsf{first}(C)$;
3: $\quad$ **for** $\ell' \in C'$ **do**
4: $\quad\quad$ **if** $\mathsf{sub\_atom}(\ell, \ell', \theta)$ **then**
5: $\quad\quad\quad C \leftarrow C \backslash \{\ell\}$;
6: $\quad\quad\quad$ **continue outer loop** ;
7: $\quad$ **end for**
8: $\quad$ **return** false ;
9: **end while**
10: **return** true ;
end function

Plotkin (1970) a montré (modulo θ-équivalence) l'unicité de ce moindre généralisé pour les clauses de Horn et a proposé son calcul par les algorithmes 3 (lgg_atom et lgg). Cet algorithme n'est pas de complexité exponentielle, mais il peut donner comme résultat une clause qui n'est pas *réduite*, c'est-à-dire présentant un ou plusieurs *littéraux redondants*.

Algorithm 3 Moindre généralisé et réduction.

function $\mathsf{lgg\_atom}(\ell, \ell', \mathbf{var}\ \Theta)$
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: $\quad\quad t_i^\star \leftarrow t_i$;
8: $\quad$ **else if** $\exists t_i/s_i/t \in \Theta$ **then**
9: $\quad\quad t_i^\star \leftarrow t$;
10: $\quad$ **else**
11: $\quad\quad t_i^\star \leftarrow$ **new** variable ;
12: $\quad\quad \Theta \leftarrow \Theta \cup \{t_i/s_i/t_i^\star\}$;
13: **end for**
14: **return** $\{R(t_1^\star, \ldots, t_n^\star)\}$;
end function

function $\mathsf{lgg}(C, C')$
15: $G \leftarrow \varnothing$;
16: $\Theta \leftarrow \varnothing$;
17: **for** $\ell \in C$ **do**
18: $\quad$ **for** $\ell' \in C'$ **do**
19: $\quad\quad G \leftarrow G \cup \mathsf{lgg\_atom}(\ell, \ell', \Theta)$;
20: $\quad$ **end for**
21: **end for**
22: **return** G ;
end function

function $\mathsf{reduce}(C)$
23: $R \leftarrow C$;
24: **for** $\ell \in C$ **do**
25: $\quad$ **if** $\mathsf{sub}(R \backslash \{\ell\}, R)$ **then**
26: $\quad\quad R \leftarrow R \backslash \{\ell\}$;
27: **end for**
28: **return** R ;
end function

Définition 4 (Littéral redondant)
Soit une clause C et l un littéral de C. l est un littéral redondant de C ssi $C \geq_\theta C\backslash\{l\}$.

Dans un tel cas, il est préférable de nettoyer la clause de ses littéraux redondants : pour limiter la taille des clauses manipulées et donc le temps nécessaire aux calculs de θ-subsomption et aussi pour bénéficier d'une réelle unicité du moindre généralisé. Gottlob & Fermüller (1993) ont démontré que décider si une clause est réduite ou non est un problème co-NP-complet. Ils ont aussi montré que trouver la réduction d'une clause nécessite au moins un nombre linéaire d'appel à un NP-oracle. Cela s'explique en présentant la réduction comme une succession de plusieurs tests de θ-subsomption, comme le montre l'algorithme 3 (reduce).

Exemple 1
Considérons les clauses C et C' définies comme suit :
C : $fc(n_1, n_2), ns(n_2, n_3), fc(n_4, n_5)$.
C' : $fc(m_1, m_2), ns(m_2, m_3)$.
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, sa suppression donne la clause : G' : $fc(X_1, X_2), ns(X_2, X_3)$ qui est bien équivalente à G puisque $G \geq_\theta G'$ avec $\theta = \{X_1/X_4, X_2/X_5\}$. Après réduction, c'est donc G' qui est fournie.

Dans la suite, nous allons chercher des cas particuliers pour lesquels les algorithmes sub, lgg et reduce peuvent être réécrits sous une forme plus efficace. Ces algorithmes seront comparés vis-à-vis de leur complexité, elle-même mesurée en nombre d'appels aux algorithmes sub_atom et lgg_atom, ceux-ci servant systématiquement de briques de base. Ces complexités seront fonction de la taille des clauses manipulées C et C', c'est-à-dire de leurs nombres de littéraux respectifs notés $|C|$ et $|C'|$.

Ainsi, dans le cas des algorithmes de Plotkin (1970) présentés, la subsomption est en $|C'|^{\mathcal{O}(|C|)}$ (exponentielle) et le calcul de moindre généralisé en $\mathcal{O}(|C| \times |C'|)$ (quadratique). La complexité de la subsomption entre clauses déterminées est elle en $\mathcal{O}(|C| \times |C'|)$ (quadratique). La complexité de reduce dépend directement de la complexité de subsomption : pour les clauses générales de Horn elle est en $|C|^{\mathcal{O}(|C|)}$ (exponentielle) et pour les clauses déterminées en $O(|C|^3)$ (cubique).

2 Clauses *Multi-Quasi-Determinate*

Nous reprenons l'idée des clauses déterminées et considérons dorénavant que les clauses sont représentées par des séquences de littéraux.

Définition 5 (clause quasi-déterminée)
$C = (l_1, \cdots, l_n)$ est quasi-déterminée (QD) par rapport à C' ssi pour toute substitution θ telle que $\{l_1\}\theta \subseteq C'$, la clause $(l_2, \cdots, l_n)\theta$ est déterminée par rapport à C'. On dit que C est QD par rapport à un langage $\mathcal{L}$ ssi elle est QD vis-à-vis de toute clause de $\mathcal{L}$.

La définition de clause quasi-déterminée utilise celle de clause déterminée (définition 2) : seul le premier littéral peut faire l'objet de choix multiples, mais une fois ce choix fait, le reste de la clause est déterminé. Ce principe se retrouve dans le test de subsomption que nous proposons pour les clauses QD à l'algorithme 4 ($\mathsf{sub}_{\mathsf{QD}}$) : il s'agit de trouver une correspondance pour le premier littéral, puis de poursuivre de manière déterminée à l'aide de l'algorithme 2 ($\mathsf{sub}_{\mathsf{D}}$) ; en cas d'échec, on revient alors sur le choix de *matching* du premier littéral. La complexité d'un test de subsomption entre deux clauses QD est en $\mathcal{O}(|C| \times |C'|)$ (quadratique).

Pour le calcul de moindre généralisé entre clauses QD, il nous semble impossible de tirer parti du quasi-déterminisme des clauses. C'est pourquoi nous utilisons simplement l'algorithme 3 (lgg) de Plotkin (1970) dont la complexité est en $\mathcal{O}(|C| \times |C'|)$ (quadratique).

Exemple 2
Considérons les deux clauses suivantes supposées QD :
C : $fc(n_1, n_2), ns(n_2, n_5), fc(n_2, n_3), fc(n_3, n_4), td(n_4)$.
C' : $fc(m_1, m_2), ns(m_2, m_3), td(m_2)$.
Le moindre généralisé de C et C' 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 $\Theta = \{n_1/m_1/X_1, n_2/m_2/X_2, n_2/m_1/X_3, n_3/m_2/X_4, n_3/m_1/X_5, n_4/m_2/X_6, n_5/m_3/X_7\}$.

Dans cet exemple, la clause obtenue même si elle est réduite n'est pas connectée et l'on peut se demander s'il s'agit encore d'une clause QD. On ne peut pas trancher sans connaître la sémantique des prédicats utilisés et l'on ne peut donc pas exclure la nécessité de séparer ces littéraux pour obtenir plusieurs clauses QD. De la même manière, il est délicat de discuter de la réduction d'un telle clause : plutôt que de confier l'ensemble des littéraux obtenus à l'algorithme reduce de Plotkin (1970), il est sans doute préférable de considérer la clause comme plusieurs clauses QD et de définir une réduction appropriée.

Ainsi, nous sommes amenés à généraliser la notion de clause QD pour capturer des clauses ayant de composantes indépendantes multiples.

Définition 6 (multi-clause)
Une multi-clause *est un ensemble de clauses $\mathscr{C} = \{C_1, C_2, \ldots, C_k\}$ telles que les clauses prises deux à deux ne partagent aucune variable. Une multi-clause est interprétée comme l'union de ses composantes et sa taille est la somme des tailles de ses composantes.*

Définition 7 (clause MQD)
$\mathscr{C} = \{C_1, C_2, \ldots, C_k\}$ est une clause multi-quasi-determinate *(MQD) par rapport à une clause C' ssi chaque composante C_i de $\mathscr{C}$ est une clause QD maximale par rapport à C', c'est-à-dire que $C_i \cup \{\ell\}$ n'est pas QD quelle que soit $\ell \in C_{j \neq i}$. On dit que C est MQD par rapport à un langage $\mathcal{L}$ ssi elle est MQD vis-à-vis de toute clause de $\mathcal{L}$.*

Le test de θ-subsumption entre deux clauses MQD $\mathscr{C}$ et $\mathscr{C}'$ est précisé par l'algorithme 4 ($\mathsf{sub}_{\mathsf{MQD}}$) : comme les composantes ne partagent pas de variable, la méthode consiste à utiliser $\mathsf{sub}_{\mathsf{QD}}$ pour trouver des correspondances indépendantes entre les composantes de $\mathscr{C}$ et $\mathscr{C}'$. La taille d'une clause MQD $\mathscr{C} = \{C_1, \ldots, C_n\}$ étant la somme des tailles de ses composants, i.e. $|\mathscr{C}| = |C_1| + \ldots + |C_n|$, la complexité de $\mathsf{sub}_{\mathsf{MQD}}$ est en $\mathcal{O}(\sum_{C \in \mathscr{C}} \sum_{C' \in \mathscr{C}'} |C| \times |C'|)$ et, par suite, en $\mathcal{O}(|\mathscr{C}| \times |\mathscr{C}'|)$ (quadratique).

Algorithm 4 θ-subsumption pour clauses QD et MQD.

function $\mathsf{sub}_{\mathsf{QD}}(C, C')$
1: **if** $C = \varnothing$ **then**
2: **return** $true$;
3: $\ell \leftarrow \mathsf{first}(C)$;
4: **for** $\ell' \in C'$ **do**
5: $\theta \leftarrow \varnothing$;
6: **if** $\mathsf{sub\_atom}(\ell, \ell', \theta)$ **and**
7: $\mathsf{sub}_{\mathsf{D}}(C \backslash \{\ell\}, C', \theta)$
8: **then**
9: **return** $true$;
10: **return** $false$;
end function

function $\mathsf{sub}_{\mathsf{MQD}}(\mathscr{C}, \mathscr{C}')$
11: **for** $C \in \mathscr{C}$ **do**
12: **for** $C' \in \mathscr{C}'$ **do**
13: **if** $\mathsf{sub}_{\mathsf{QD}}(C, C')$ **then**
14: **continue outer loop** ;
15: **end for**
16: **return** $false$;
17: **end for**
18: **return** $true$;
end function

Intuitivement, le calcul de lgg entre deux clauses MQD $\mathscr{C}$ et $\mathscr{C}'$ va suivre la même logique : nous allons généraliser deux à deux les composantes de $\mathscr{C}$ et $\mathscr{C}'$, en sachant que les variables produites au cours d'une généralisation de deux composantes ne pourront pas apparaître ailleurs. Cependant, nous avons vu que les littéraux ainsi obtenus nécessitent un partitionnement pour retrouver une clause MQD. Cette opération et donc le calcul de lgg, tout comme la réduction de clauses MQD, sont impossibles à donner dans une version générale car il faut pour cela préciser les prédicats utilisés et leurs sémantiques.

C'est l'objet de la section suivante : nous allons définir ces opérations particulières pour des clauses MQD dédiées à la représentation d'arbres.

3 Application aux arbres

Pour ce codage, nous cherchons à représenter les arbres sous forme de clauses MQD et utilisons pour cela un codage classique nommé *first-child/next-sibling*.

Définition 8 (langage $\mathcal{L}_{fcns}$)
Le langage $\mathcal{L}_{fcns}$ est basé sur l'utilisation des symboles de prédicats 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 immédiatement x ;*
- *$a(x)$ indique que x porte le label a avec $a \in \Sigma$.*

L'ensemble des constantes permises dans les clauses de $\mathcal{L}_{fcns}$ est l'ensemble des identifiants de nœuds D. De plus, l'apparition d'une constante ou d'une variable au sein

d'une clause est sujette aux règles ci-dessous :

- *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 ;*
- *elle ne peut apparaître qu'au plus une fois comme second argument de fc ou ns ;*
- *son occurrence comme second argument de fc ou ns doit précéder son utilisation comme premier argument de fc et ns.*

Un arbre à n nœuds est ainsi représenté par une clause dont le nombre de littéraux est en $\mathcal{O}(n)$ et nos complexités se lisent directement en fonction de la taille des arbres.

L'intuition de l'aspect QD des clauses de $\mathcal{L}_{fcns}$ est facile à appréhender : à partir d'un point d'ancrage dans l'arbre cible, il y a au plus une manière de suivre un motif défini à l'aide des prédicats fc et ns.

Nous pouvons noter que si une clause de $\mathcal{L}_{fcns}$ utilise une constante, alors cette clause a au moins une composante déterminée (car la constante offre un point d'ancrage unique dans la clause à subsumer). Par souci de simplicité, nous allons négliger cette possibilité et nous consacrer uniquement au traitement de clauses qui n'ont pas de partie déterminée. Cette hypothèse est purement technique et n'est pas restrictive : notre approche peut aisément intégrer une partie déterminée, par exemple à la manière de Kietz & Lübbe (1994) pour les clauses *clauses k-locales*.

Autre observation qui va nous être utile : toute clause décrivant une arbre n'a que des constantes propres, qu'elle ne partage avec aucune autre et, par suite, la généralisation de deux clauses représentant deux arbres distincts est complètement variabilisée.

Le lemme suivant formalise les relations entre les clauses MQD et le langage $\mathcal{L}_{fcns}$.

Lemme 1

1. *Toute clause de $\mathcal{L}_{fcns}$ représentant un arbre est QD.*
2. *Si l'on a une telle clause C, alors $\{C\}$ est une clause MQD.*
3. *Une clause de $\mathcal{L}_{fcns}$ sans constante est une clause QD ssi elle est connectée.*

Le dernier point nous permet de revenir sur l'exemple 2 pour lequel nous connaissons maintenant la sémantique des prédicats. Cet exemple démontre que la clause résultat d'une généralisation de clauses QD n'est pas nécessairement une clause QD (car la clause obtenue n'est pas connectée et donc pas QD d'après le dernier point du lemme 1). Cependant, le moyen de revenir à une clause MQD après généralisation de deux clauses de $\mathcal{L}_{fcns}$ apparaît clairement : on obtient une union de clauses QD maximales simplement en regroupant ensemble les littéraux connectés.

C'est sur ce principe que nous définissons la fonction split propre à $\mathcal{L}_{fcns}$: elle est présentée à l'algorithme 5 (split_{fcns}). Bien qu'elle utilise trois boucles, cette fonction a une complexité en $\mathcal{O}(|G|^2)$ (quadratique) : la deuxième boucle parcourt F qui sert de frontière aux éléments de l'ensemble G et ainsi, tout élément ajouté à F est ensuite supprimé de G et tout élément est ajouté à F exactement une seule fois. Cette fonction est utilisée par lgg_{fcns}pour produire une clause MQD : pour toute paire de clauses QD celui-ci effectue un appel à l'algorithme de moindre généralisé de Plotkin (lgg de l'algorithme 3). La complexité de ce calcul est donc en $\mathcal{O}(|\mathscr{C}| \times |\mathscr{C}'|)$ (quadratique).

Algorithm 5 split, lgg, et reduce pour les clauses de $\mathcal{L}_{fcns}$.

function $\mathsf{split}_{fcns}(G)$
1: $\mathscr{C} \leftarrow \varnothing$;
2: **while** $G \neq \varnothing$ **do**
3: **choose** $\ell \in G$;
4: $G \leftarrow G \backslash \{\ell\}$;
5: $C \leftarrow \{\ell\}$; *// indep. component*
6: $F \leftarrow \{\ell\}$; *// frontier set*
7: **while** $F \neq \varnothing$ **do**
8: **choose** $R(t_1, \ldots, t_n) \in F$;
9: $F \leftarrow F \backslash \{R(t_1, \ldots, t_n)\}$;
10: **for** $P(s_1, \ldots, s_m) \in G$ **do**
11: **if** $\exists i, j \; t_i = s_j$ **then**
12: $G \leftarrow G \backslash \{P(s_1, \ldots, s_m)\}$;
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

function $\mathsf{lgg}_{fcns}(\mathscr{C}, \mathscr{C}')$
20: $\mathscr{G} \leftarrow \varnothing$;
21: **for** $C \in \mathscr{C}$ **do**
22: **for** $C' \in \mathscr{C}'$ **do**
23: $\mathscr{G} \leftarrow \mathscr{G} \cup \mathsf{split}_{fcns}(\mathsf{lgg}(C, C'))$;
24: **end for**
25: **end for**
26: **return** $\mathscr{G}$;
end function

function $\mathsf{reduce}_{fcns}(\mathscr{G})$
27: $\mathscr{R} \leftarrow \mathscr{G}$;
28: **for** $C \in \mathscr{G}$ **do**
29: **for** $C' \in \mathscr{G}$ **do**
30: **if** $\mathsf{sub}_{\mathsf{QD}}(C, C')$ **then**
31: $\mathscr{R} \leftarrow \mathscr{R} \backslash \{C\}$;
32: **end for**
33: **end for**
34: **return** $\mathscr{R}$;
end function

Reste à discuter de la redondance des clauses MQD ainsi obtenues. Le lemme suivant indique une bonne propriété des MQD nous dispensant de rechercher des littéraux redondants au sein de chaque composante.

Lemme 2
Pour toute clause MQD $\mathscr{C} = \{C_1, \cdots, C_n\}$ moindre généralisée obtenue par lgg_{fcns} et split_{fcns} à partir de clauses de $\mathcal{L}_{fcns}$ réduites, chaque C_i est réduite.

Par contre, l'exemple 3 montre qu'une redondance entre composantes peut apparaître.

Exemple 3
Après application de la fonction split_{fcns}, la clause calculée dans l'exemple 2 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 est redondante et peut être supprimée.

Finalement, nous traitons la redondance dans les clauses MQD de $\mathcal{L}_{fcns}$ à l'aide de l'algorithme 5 (reduce_{fcns}) qui manipule uniquement des clauses QD ce qui 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. Sa complexité est en $\mathcal{O}(|\mathscr{C}|^2)$ (quadratique).

En résumé, nous sommes capables d'apprendre à partir d'arbres représentés par des clauses MQD : tout arbre est représenté par une clause MQD (deuxième point du lemme 1), l'algorithme lgg_{fcns} généralise de tels exemples, fournit toujours des clauses MQD et tous les algorithmes nécessaires sont polynomiaux.

4 Conclusion

Travaux apparentés. Tout d'abord, Muggleton & Feng (1990) 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. En tout état de cause, les clauses ij-déterminées, avec i et j fixés, forment une sous-classe des clauses déterminées et 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.

Kietz & Lübbe (1994) ont eux proposé un algorithme de θ-subsomption efficace pour les *clauses k-locales*. Ces clauses peuvent avoir une partie déterminée mais aussi et surtout des parties non déterminées appelées *locales*, qui seraient donc l'équivalent de nos clauses QD au sein des MQD. Cependant, là où nous exigeons un quasi-déterminisme, les *locales* sont contraintes de ne pas avoir plus de k littéraux ou k variables propres. Gérer des parties non déterminées ramène à la complexité exponentielle de Plotkin (1970) et 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.

Dans le cadre de la tâche d'extraction d'informations, Thomas (2005) 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 pour reformuler les clauses et obtenir des descriptions où chaque symbole de prédicat n'est utilisé qu'une seule fois. Autrement dit, il s'agit d'une restriction des clauses déterminées.

Citons également les travaux de Ferilli *et al.* (2002) sur l'*identité d'objets* (OI). 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 clause. Cette contrainte a pour nous 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.

Les travaux de Maloberti & Sebag (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 conséquence, nous jugeons préférable d'utiliser nos algorithmes lorsque la nature des clauses est connue comme étant MQD, et de réserver Django aux cas où les clauses manipulées n'ont pas de propriétés exploitables.

Terminons avec Liquière (2007) qui a proposé une nouvelle relation de généralité pour les graphes, cette relation n'étant pas toujours équivalente à la θ-subsomption. Nous pensons que ce cadre est assez général pour représenter les clauses de $\mathcal{L}_{fcns}$ et que dans ce cas précis la subsomption proposée est bien équivalente à la θ-subsomption. Cependant, nous avons établi que nos algorithmes dédiés bénéficiaient d'une meilleure complexité que les algorithmes généraux pour les graphes proposés par Liquière (2007).

Bilan. Nous avons proposé une sous-famille des clauses de Horn destinées à la classification en récrivant pour elle les algorithmes de θ-subsomption et de moindre généralisé avec une complexité polynomiale. Ces algorithmes sont suffisants pour permettre l'utilisation de nombreuses méthodes d'apprentissage qui pour le coup seront elles aussi polynomiales dans le cas des arbres.

Notre démarche a consisté à suivre la piste des clauses *déterminées* : les QD contiennent les déterminées mais est une famille strictement plus large, les MQD ferment les QD pour l'opération de moindre généralisé. Nous avons également vu les clauses du langage $\mathcal{L}_{fcns}$, sous-classe des MQD dédiée aux arbres.

La table 1 récapitule toutes les complexités établies dans cet article.

TABLE 1 – Récapitulatif des complexités des opérations en fonction des clauses.

	θ-subsomption	moindre-généralisé	moindre-généralisé réduit
Horn	$\lvert C'\rvert^{\mathcal{O}(\lvert C\rvert)}$	$\mathcal{O}(\lvert C\rvert \times \lvert C'\rvert)$	$(\lvert C\rvert \times \lvert C'\rvert)^{\mathcal{O}(\lvert C\rvert \times \lvert C'\rvert)}$
D	$\mathcal{O}(\lvert C\rvert \times \lvert C'\rvert)$	$\mathcal{O}(\lvert C\rvert \times \lvert C'\rvert)$	$\mathcal{O}((\lvert C\rvert \times \lvert C'\rvert)^3)$
QD	$\mathcal{O}(\lvert C\rvert \times \lvert C'\rvert)$	$\mathcal{O}(\lvert C\rvert \times \lvert C'\rvert)$	-
$\mathcal{L}_{fcns}$	$\mathcal{O}(\lvert \mathscr{C}\rvert \times \lvert \mathscr{C}'\rvert)$	$\mathcal{O}(\lvert \mathscr{C}\rvert \times \lvert \mathscr{C}'\rvert)$	$\mathcal{O}((\lvert \mathscr{C}\rvert \times \lvert \mathscr{C}'\rvert)^2)$
MQD	$\mathcal{O}(\lvert \mathscr{C}\rvert \times \lvert \mathscr{C}'\rvert)$	-	-

Premières expérimentations. À l'aide d'expériences préliminaires, nous avons montré que nos méthodes permettent effectivement l'apprentissage de classifieurs d'arbres et que cette tâche n'était pas possible avec les algorithmes généraux de Plotkin (1970) : nous apprenons une théorie en quelques heures là où l'on estime que les algorithmes de Plotkin (1970) mettront plusieurs années à répondre à un test de subsomption.

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 (en moyenne après le cinquième exemple généralisé).

Perspectives. Nous allons poursuivre les implémentations et expérimentations sur des données arborescentes réelles et nous comparer aux récentes méthodes de l'apprentissage relationnel dédiées aux graphes.

Sur le plan théorique, nous espérons d'une part qu'il existe d'autres objets que les arbres qui pourraient bénéficier de notre formalisme MQD et de nos algorithmes polynomiaux, de nombreuses classes de graphes sont candidates. Nous continuerons d'autre part de chercher des familles de clauses pour lesquelles le test de subsomption reste polynomial. En particulier, il reste à explorer l'espace des clauses MQD qui ne sont pas dans $\mathcal{L}_{fcns}$. Il conviendra alors de s'interroger sur les applications de telle ou telle famille de clauses, l'une de nos préoccupations restant la recherche de représentations toujours plus riches des arbres.

Références

FERILLI S., FANIZZI N., MAURO N. D. & BASILE T. M. A. (2002). Efficient theta-subsumption under object identity. In *Atti del Workshop AI*IA su Apprendimento Automatico*.

GOTTLOB G. & FERMÜLLER C. G. (1993). Removing redundancy from a clause. *Artificial Intelligence*, **61**(2), 263–289.

GOTTLOB G., KOCH C. & WIEN T. U. (2002). Monadic datalog and the expressive power of languages for web information extraction. *J. ACM*, **513**, 17–28.

KIETZ J.-U. (1993). 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*, p. 149–164.

KIETZ J.-U. & LÜBBE M. (1994). An efficient subsumption algorithm for inductive logic programming. In W. W. COHEN & H. HIRSH, Eds., *Proceedings 11th International Conference on Machine Learning*, p. 130–138 : Morgan Kaufmann.

LIQUIÈRE M. (2007). Arc consistency projection : A new generalization relation for graphs. In U. PRISS, S. POLOVINA & R. HILL, Eds., *15th International Conference on Conceptual Structures (ICCS 2007)*, volume 4604 of *Lecture Notes in Computer Science*, p. 333–346 : Springer.

MALOBERTI J. & SEBAG M. (2001). Theta-subsumption in a constraint satisfaction perspective. In *ILP '01 : Proceedings of the 11th International Conference on Inductive Logic Programming*, p. 164–178, London, UK : Springer-Verlag.

MUGGLETON S. (1991). Inductive logic programming. *New Generation Computing Journal*, **8**(4), 295–317.

MUGGLETON S. & FENG C. (1990). Efficient induction of logic programs. In *Proceedings of the 1st Conference on Algorithmic Learning Theory*, p. 368–381 : Ohmsma, Tokyo, Japan.

PLOTKIN G. (1970). A note on inductive generalization. In B. MELTZER & D. MITCHIE, Eds., *Machine Intelligence*, volume 5, p. 153–165. Edinburgh University Press.

QUINLAN J. R. (1991). Determinate literals in inductive logic programming. In *Proceedings of ICML*, p. 442–446.

THOMAS B. (2005). *Machine learning of information extraction procedures : an ILP approach*. PhD thesis, Universität Koblenz-Landau, Institut für Informatik.