Nouveau modèle pour un passage à l'échelle de la θ-subsomption

Hippolyte Léger, Dominique Bouthinon, Mustapha Lebbah, Hanane Azzag

Universite Paris 13, Sorbonne Paris Cite, L.I.P.N
UMR-CNRS 7030 F-93430, Villetaneuse, France
{leger, bouthinon, lebbah, azzag}@lipn.univ-paris13.fr

Résumé. Le test de θ-subsomption, opération fondamentale en Programmation Logique Inductive (PLI) pour tester la validité d'une hypothèse sur les exemples, est particulièrement coûteux. Ainsi, les systèmes d'apprentissage de PLI les plus récents ne passent pas à l'échelle. Nous proposons donc un nouveau modèle de θ-subsomption fondé sur un réseau d'acteurs, dans le but de pouvoir décider la subsomption sur de très grandes clauses.

1 Introduction

La θ-subsomption est utilisée dans de nombreux systèmes de Programmation Logique Inductive (PLI) pour tester la validité d'une hypothèse sur les exemples. Une clause C θ-subsume une clause D si et seulement si il existe une substitution θ telle que $C\theta \subseteq D$. Malheureusement, la complexité temporelle de pire cas de la θ-subsomption est $(O(|D|^{|C|}))$. De nombreuses recherches ont été menées pour créer des algorithmes de θ-subsomption efficaces (Ferilli et al. (2003); Kuzelka et Zelezný (2008); Santos et Muggleton (2010)) [1]. Cependant, le passage à l'échelle de la subsomption sur des plate-formes distribuées a reçu beaucoup moins d'attention et aucun système à notre connaissance ne se concentre sur la θ-subsomption. Notre but est de créer un modèle générique de θ-subsomption pouvant passer à l'échelle et être facilement intégré à des systèmes d'apprentissage relationnel en utilisant des plate-formes Big Data distribuées.

2 Preliminaires

Nous considérons ici la θ-subsomption entre deux clauses C et D où C et D sont des clauses de Horn définies sans fonctions, C contenant des variables et D des constantes. Une substitution est un ensemble fini $\{X_1/v_1, \ldots, X_n/v_n\}$ où X_i est une variable et v_i est une constante (une variable apparaît une seule fois dans une substitution). Deux substitutions θ_1 et θ_2 ne sont pas compatibles si elles assignent deux valeurs distinctes à une même variable, par exemple $\theta_1 = \{Y/a\}$ et $\theta_2 = \{Y/b\}$. En revanche l'union de deux substitutions compatibles est toujours valide. Nous présentons ci-dessous, un exemple θ-subsomption utilisé tout au long de l'article.

1. Voir Ferilli et al. (2003) pour une étude plus approfondie de ces travaux

Exemple 1

$$
\begin{array}{llllllllll}
C & = & t(X) & \leftarrow & p(X,Y,Z) & \wedge & q(Z,T) & \wedge & r(T,T,U). & \\
D & = & t(a) & \leftarrow & p(a,b,c) & \wedge & q(c,e) & \wedge & r(e,e,g) & \wedge \\
 & & & & p(a,b,d) & \wedge & q(d,f) & \wedge & r(f,f,g) & \wedge \\
 & & & & & & & & r(e,f,g). &
\end{array}
$$

L'exemple 1 montre que C θ-subsume D pour $\theta = \{X/a, Y/b, Z/c, T/e, U/g\}$ et $\theta = \{X/a, Y/b, Z/d, T/f, U/g\}$. Les propriétés suivantes seront utilisées dans notre modèle :

Propriété 1 *C θ-subsume D si et seulement si 1)] il existe une substitution α ne référant que des variables de $head(C)$, telle que $head(C)\alpha = head(D)$ et, 2) il existe une substitution μ ne référant que des variables de $body(C)\alpha$, telle que $body(C)\alpha\mu \subseteq body(D)$. (La preuve est évidente.)*

Propriété 2 *Soit $A = \{a_1, \ldots, a_n\}$ une conjonction de littéraux et B une conjonction de littéraux clos. Alors, A θ-subsume B si et seulement si il existe un ensemble de substitutions compatibles $\{\mu_1, \ldots, \mu_n\}$ tel que $a_i\mu_i \in B$ ($1 \leq i \leq n$).*

Preuve *$A\mu \subseteq B \Leftrightarrow \{a_1\mu, \ldots, a_n\mu\} \subseteq B \Leftrightarrow$ il existe un ensemble de substitutions compatible $\{\mu_1, \ldots, \mu_n\}$ avec $\mu = \mu_1 \cup \ldots \cup \mu_n$ et où μ_i ne réfère que des variables de a_i, tel que $\{a_1\mu_1, \ldots, a_n\mu_n\} \subseteq B \Leftrightarrow a_i\mu_i \in B$ ($1 \leq i \leq n$). $\square$*

Considérons $A = body(C)\alpha$ et $B = body(D)$ sus-mentionnés. On remarque que $A\mu \subseteq B$ avec $\mu = \mu_1 \cup \mu_2 \cup \mu_3$ où $\mu_1 = \{Y/b, Z/c\}$, $\mu_2 = \{Z/c, T/e\}$ et $\mu_3 = \{T/e, U/g\}$.

D'après les propriétés 1 et 2, le problème de subsomption entre deux clauses C et D peut être modélisé comme suit :

1. trouver une substitution α ne référant que des variables de $head(C)$ tel que $head(C)\alpha = head(D)$,
2. si l'étape 1 a réussi : (soit $body(C)\alpha = \{a_1, \cdots, a_n\}$) trouver un ensemble de substitutions compatibles $\{\mu_1, \cdots, \mu_n\}$, μ_i ne référant que des variables de a_i, telle que $a_i\mu_i \in body(D)$ ($1 \leq i \leq n$),
3. si l'étape 2 a réussi : la subsomption est assuré par la substitution $\theta = \alpha \cup \mu_1 \cup \cdots \cup \mu_n$.

L'étape 1 est très facile à vérifier. Le modèle de θ-subsomption que nous présenterons dans la section suivante se concentrera donc sur l'étape 2.

3 θ-subsomption basée sur un réseau d'acteurs

Étant donné une conjonction (généralement non close) $A = \{a_1, \ldots, a_n\}$ et une conjonction close B, nous cherchons un ensemble $\{\mu_1, \ldots, \mu_n\}$ de substitutions compatibles tel que μ_i ne réfère que des variables de a_i, et $a_i\mu_i$ appartient à B ($1 \leq i \leq n$). Le schéma de θ-subsomption que nous proposons est fondé sur un modèle d'acteurs (Hewitt et al. (1973)). Un acteur est une entité indépendante, liée à d'autres acteurs via un système de messagerie asynchrone. Les acteurs forment un graphe orienté, que l'on appellera un réseau d'acteurs.

Afin de résoudre le problème de θ-subsomption entre les conjonctions A et B on construit d'abord un réseau d'acteurs à partir de A (cf. algorithme 1). Ensuite, on envoie les atomes de B au réseau qui retournera la première (ou toutes les) substitution(s) vérifiant la θ-subsomption

ainsi qu'un message *fin*. Si le problème n'a aucune solution, le réseau ne retourne que le message *fin*.

Algorithm 1 Construction du réseau d'acteurs.

buildNetwork(A) /* $A = \{a_1, \ldots, a_n\}$ est une conjonction de n littéraux */
begin
  create the output actor out ; buildTree(out, n) ; /* création de l'arbre d'acteurs */
  create the input actor in ;
  pour chaque acteur de substitution s_i (feuille de l'arbre de racine *out*) **faire**
    link s_i with in ; set a_i as internal label of s_i ;
  return in ;
end.
buildTree(j**,** n**)** /* j : acteur de jointure (ou de sortie) du niveau précédent. n : le nombre de feuilles restant à considérer */
begin
  si $n = 1$ **alors** create a substitution actor a and store it ; /* nouvelle feuille de l'arbre */
  sinon create a join actor a ; buildTree(a, $n/2 + n$ mod 2) ; buildTree(a, $n/2$) ; **finSi**
  link a to j ;
end.

Construction du réseau Le réseau (un graphe orienté) est constitué de quatre types d'acteurs, comme le montre la figure 1.

L'*acteur d'entrée* est l'unique point d'entrée du réseau. Chaque message qu'il reçoit est un atome de B. L' *acteur de substitution* (représenté par un cercle) : chacun est associé à un atome a_i de A et a pour rôle de construire des substitutions à partir des atomes clos qu'il reçoit. L'*acteur de jointure* (représenté par un rectangle) a deux parents et a pour rôle d'unir (si possible) les substitutions qu'il reçoit de ses parents. Cet acteur possède deux mémoires internes (*gauche* et *droite*) afin de stocker les substitutions fournies par ses parents. L'*acteur de sortie* est le seul point de sortie du réseau. Il reçoit les substitutions (si il en existe) qui établissent la θ-subsomption entre A et B.

La procédure de θ-subsomption Illustrons la procédure de θ-subsomption à travers le réseau présenté Figure 1, construit à partir de la conjonction $A = body(C)\alpha = \{p(a, Y, Z), q(Z, T), r(T, T, U)\}$ où C est la clause donnée dans l'exemple 1. Supposons que les atomes de $B = body(D) = \{p(a, b, c), p(a, b, d), q(c, e), q(d, f), r(e, e, g), r(f, f, g), r(e, f, g)\}$ sont envoyés au réseau :

— Lorsqu'un atome clos b de B est fourni à l'acteur d'entrée, ce dernier envoie b à tous les acteurs de substitution associés aux atomes de A construits à partir du même prédicat que b. ex) l'acteur d'entrée reçoit $b = p(a, b, c)$, puis envoie $p(a, b, c)$ à l'acteur $p(a, Y, Z)$.

— Lorsqu'un acteur de substitution associé à un atome a_i de A reçoit un atome clos b, il vérifie s'il existe une substitution μ_i telle que $a_i\mu_i = b$. Si μ_i existe, l'acteur envoie cette substitution au seul acteur de jointure auquel il est lié. ex) si $b = p(a, b, c)$ et $a_i = p(a, Y, Z)$, la substitution $\mu_i = \{Y/b, Z/c\}$ est envoyée à l'acteur de jointure j_1. Si $b = p(e, b, c)$, rien n'est envoyé.

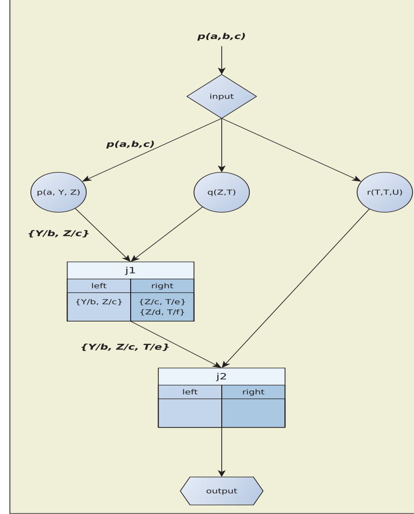

FIG. 1: Réseau d'acteurs construit à partir d'une conjonction $A = \{p(a, Y, Z),\ q(Z, T),\ r(T, T, U)\}$.

— Lorsqu'un acteur de jointure reçoit une substitution μ de son parent gauche (droit), il la stocke dans sa mémoire gauche (droite). Puis, il tente d'unir μ avec chaque substitution δ de la mémoire droite (gauche). Pour chaque substitution compatible δ, l'acteur envoie $\mu \cup \delta$ à son successeur. ex) l'acteur de jointure j_1 reçoit $\mu = \{Y/b, Z/c\}$ de son parent gauche, puis stocke cette substitution dans sa mémoire gauche. Supposons que la mémoire droite de j_1 contienne les substitutions $\delta_1 = \{Z/c, T/e\}$ et $\delta_2 = \{Z/d, T/f\}$. Ainsi, $\mu \cup \delta_1 = \{Y/b, Z/c, T/e\}$ est envoyé au successeur de j_1, alors que $\mu \cup \delta_2 = \{Y/b, Z/c, Z/d, T/f\}$ n'est pas considéré (substitution non valide).

— Lorsque l'acteur de sortie reçoit une substitution μ, il l'affiche comme solution ($A\mu \subseteq B$). Si on ne veut qu'une seule solution, le test se termine. Sinon, l'acteur attend les autres solutions.

Afin d'assurer que le réseau s'arrête, on envoie un *message de fin* à l'acteur d'entrée lorsque le dernier atome de B a été envoyé. Le *message de fin* est propagé à travers le réseau jusqu'à l'acteur de sortie, qui termine l'exécution.

4 Expérimentations

Dans cette section nous décrivons l'implémentation du réseau d'acteurs introduit dans la section 3.2, puis nous présentons nos expériences et les résultats obtenus.

4.1 Implémentation

Nous utilisons Akka (Allen (2013)), qui est intégré à Scala (Odersky et al. (2004)) pour implémenter notre modèle acteur de θ-subsomption. Dans Akka, le parallélisme est géré par

throughput	pmax	temps (s)
1	1 [1]	3296
	2 [3]	619
	3 [7]	499
	4 [10]	605
10	1 [1]	3214
	2 [3]	427
	3 [7]	269
	4 [10]	329
100	1 [1]	3620
	2 [3]	650
	3 [7]	317
	4 [10]	148
1000	1 [1]	7371
	2 [3]	1271
	3 [7]	403
	4 [10]	411

FIG. 2: Temps de calcul moyen pour chaque configuration du *Dispatcher*.

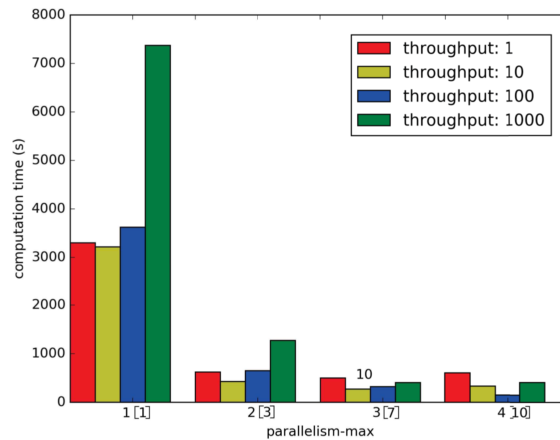

FIG. 3: Temps de calcul moyen pour chaque configuration du *Dispatcher*.

un *Dispatcher* dont les deux paramètres principaux sont *parallelism-max*, qui limite le nombre de *threads* disponibles pour l'exécution, et *throughput* qui limite le nombre de messages traités par un acteur avant de passer à l'acteur suivant. Dans nos expériences nous avons fait varier le paramètre *parallelism-max* entre 1 et 4 et le paramètre *throughput* entre 1 et 1000. Tous les tests ont été éxécutés sur une machine Linux 64 bits équipée d'un processeur Intel Core i7-5600U de deux coeurs (avec hyperthreading) à 2.6GHz, 8GO de mémoire avec Scala 2.11.7 et Akka 2.4.4.

4.2 Données

Nous avons exécuté notre implémentation sur un couple hypothèse/exemple afin d'observer comment se comporte le réseau d'acteurs sur différentes configurations de parallélisme. Les données sont issues de (Santos et Muggleton (2010)), et ont été engendrées à partir d'instances du problème de transition de phase (Giordana et Saitta (2000)). L'hypothèse est une clause de 30 littéraux, avec 4 symboles de prédicats d'arité 2 ou 3, avec en tout 8 symboles de variables. L'exemple a les mêmes propriétés, mais contient 200 littéraux.

4.3 Performances

On peut voir dans les figures 3 et 4 que chaque paramètre a un impact sur le temps de calcul (les nombres entre crochets sont les nombres effectifs de threads utilisés). En particulier, l'augmentation du parallélisme amène un réduction significative du temps de calcul. Notons

que le nombre d'acteurs dépend directement du nombre de littéraux présents dans l'hypothèse. Le rapport coût/efficacité du parallélisme augmente donc avec la taille de la clause hypothèse.

5 Conclusion

Nous avons créé un nouveau modèle de θ-subsumption, qui montre un certain potentiel de passage à l'échelle. Nous avons montré qu'un modèle à base d'acteurs est efficace pour réduire le temps de calcul. Nous venons de commencer les tests de ce modèle en environnement distribué (cluster).

Références

Allen, J. (2013). *Effective Akka*. O'Reilly Media, Inc.

Ferilli, S., N. Mauro, T. M. A. Basile, et F. Esposito (2003). *AI*IA 2003 : Advances in Artificial Intelligence : 8th Congress of the Italian Association for Artificial Intelligence, Pisa, Italy, September 2003. Proceedings*, Chapter A Complete Subsumption Algorithm, pp. 1–13. Berlin, Heidelberg : Springer Berlin Heidelberg.

Giordana, A. et L. Saitta (2000). Phase transitions in relational learning. *Machine Learning 41*(2), 217–251.

Hewitt, C., P. Bishop, et R. Steiger (1973). A universal modular actor formalism for artificial intelligence. In *Proceedings of the 3rd International Joint Conference on Artificial Intelligence*, IJCAI'73, San Francisco, CA, USA, pp. 235–245. Morgan Kaufmann Publishers Inc.

Kuzelka, O. et F. Zelezný (2008). A restarted strategy for efficient subsumption testing. *Fundam. Inform. 89*(1), 95–109.

Odersky, M. et al. (2004). An Overview of the Scala Programming Language. Technical Report IC/2004/64, EPFL, Lausanne, Switzerland.

Santos, J. et S. Muggleton (2010). Subsumer : A Prolog theta-subsumption engine. In M. Hermenegildo et T. Schaub (Eds.), *Technical Communications of the 26th International Conference on Logic Programming*, Volume 7 of *Leibniz International Proceedings in Informatics (LIPIcs)*, Dagstuhl, Germany, pp. 172–181. Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik.

Summary

The θ-subsumption test is known to be a bottleneck in Inductive Logic Programming (ILP). The state-of-the-art learning systems in this field are not scalable. We introduce a new model of θ-subsumption and an algorithm based on an actor model, with the aim of being able to decide subsumption on very large clauses.