A new Model for Scalable θ-subsumption

Hippolyte Leger, Dominique Bouthinon, Mustapha Lebbah, and Hanane Azzag

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

Abstract. The θ-subsumption test is known to be a bottleneck in Inductive Logic Programming. The state-of-the-art learning systems in this field are hardly scalable. So we introduce a new θ-subsumption algorithm based on an Actor Model, with the aim of being able to decide subsumption on very large clauses. We use Akka, a powerful tool to build distributed actor systems based on the JVM and the Scala language.

Keywords: θ-subsumption, Inductive Logic Programming, Actor Model, distributed computing, Akka

1 Introduction

θ-subsumption has been introduced by Robinson [16] to replace the logic implication which is undecidable [11]: a clause C θ-subsumes a clause D if and only if there exists a substitution θ such that $C\theta \subseteq D$. Most of the Inductive Logic Programming (ILP) systems ([18, 14]) use θ-subsumption to check that a hypothesis covers an example. θ-subsumption is decidable and equivalent to logical implication when C is not self-resolving and D is not tautological [7]. Unfortunately, the worst case time complexity of θ-subsumption is $(O(|D|^{|C|}))$ even when C and D are Horn clauses and D is fixed and ground. The basic θ-subsumption algorithm, based on SLD-resolution used in Prolog, is inefficient when the predicates are not determinate [13, 9]. Many researches have achieved to design efficient θ-subsumption algorithms [4, 12, 10, 17]. Although researches have been conducted to make ILP capable of dealing with large data in a parallel environment (for instance [5, 6, 3, 19]), as far as we know, no system focuses on θ-subsumption under modern framework and paradigms. Our motivation is to design a simple, general model for a scalable subsumption engine that could easily be integrated into relational machine learning systems using distributed platforms. In this paper, we show how to model θ-subsumption than can be run on cloud computers, and present preliminary results.

2 Preliminaries

We consider here the θ-subsumption between two function-free definite Horn clauses C and D, where D is variable-free (ground). In our context a (ground) substitution is a finite set $\{X_1/v_1, \ldots, X_n/v_n\}$ where X_i is a variable and v_i is a constant. A variable appears only once in a substitution, which is applied to a first order formula to substitute variables with constants. Two substitutions are not compatible if they assign two distinct values to the same variable, otherwise they are said compatible. Let us introduce an example of θ-subsumption that will be used throughout this paper:

Example 1

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

Example 1 shows that $C\theta \subseteq D$ both for $\theta = \{X/a, Y/b, Z/c, T/e, U/g\}$ and $\theta = \{X/a, Y/b, Z/d, T/f, U/g\}$. Let us consider the following properties that will be used in our model:

Property 1. C θ-subsumes D if and only if

1) there exists a substitution α only referring to the variables of $\mathit{head}(C)$, such that $\mathit{head}(C)\alpha = \mathit{head}(D)$ and,
2) there exists a substitution μ only referring to the variables of $\mathit{body}(C)\alpha$, such that $\mathit{body}(C)\alpha\mu \subseteq \mathit{body}(D)$.

(The proofs of all the properties presented in this paper can be found here : `https://lipn.univ-paris13.fr/~leger/ilp2016.html`)

Example 1 we have $\mathit{head}(C)\alpha = \mathit{head}(D) = t(a)$ with $\alpha = \{X/a\}$, and $\mathit{body}(C)\alpha = \{p(a,Y,Z),\ q(Z,T),\ r(T,T,U)\}$. For $\mu = \{Y/b,\ Z/c,\ T/e,\ U/g\}$ we have $\mathit{body}(C)\alpha\mu \subseteq \mathit{body}(D)$. Thus C θ-subsumes D with $\theta = \alpha \cup \mu$.

Property 2. Let $A = \{a_1, \ldots, a_n\}$ be a conjunction of literals and B be a conjunction of ground literals. Then A θ-subsumes B if and only if there exists a set of compatible substitutions $\{\mu_1, \ldots, \mu_n\}$ such that $a_i\mu_i \in B$ $(1 \le i \le n)$.

Let us consider $A = \mathit{body}(C)\alpha$ and $B = \mathit{body}(D)$ mentioned above. We notice that $A\mu \subseteq B$ with $\mu = \mu_1 \cup \mu_2 \cup \mu_3$ where $\mu_1 = \{Y/b, Z/c\}$, $\mu_2 = \{Z/c, T/e\}$ and $\mu_3 = \{T/e, U/g\}$. So, according to properties 1 and 2, the subsumption problem of two clauses C and D can be modelled as:

1. seek a substitution α only referring the variables of $\mathit{head}(C)$ such that $\mathit{head}(C)\alpha = \mathit{head}(D)$,
2. if step1 succeeds: (let $\mathit{body}(C)\alpha = \{a_1, \cdots, a_n\}$) find a set of compatible substitutions $\{\mu_1, \cdots, \mu_n\}$, where μ_i only refers the variables of a_i, such that $a_i\mu_i \in \mathit{body}(D)$ $(1 \le i \le n)$,
3. if step 2 succeeds: output the substitution $\theta = \alpha \cup \mu_1 \cup \cdots \cup \mu_n$.

Step 1 is very easy to check, so the new model of θ-subsumption we introduce in the next section focuses on step 2.

3 θ-subsumption based on an Actor Model

Given a (usually non-ground) conjunction $A = \{a_1, \ldots, a_n\}$ and a ground conjunction B, we seek for a set of compatible substitutions $\{\mu_1, \ldots, \mu_n\}$ such that μ_i refers only the variables of a_i and $a_i\mu_i$ belongs to B ($1 \leq i \leq n$). To solve this problem we introduce an original subsumption procedure based on an Actor Model. The Actor Model was motivated by the prospect of highly parallel computing machines communicating via a high-performance communications network [8], [1]. An actor is a computational entity linked to some other actors forming a graph, or network. Actors may modify private state, but can only affect each other through asynchronous messages. The message-driven framework of the Actor Model is well adapted to represent the θ-subsumption process. To solve our problem we start by building an actor network from A. Then we send the atoms of B to the network which outputs the first (or all) substitution(s) ensuring the θ-subsumption. If there is no possible substitution it will only produce the message "end". The network (actually a directed graph) is made of four types of actors as illustrated in Figure 1:

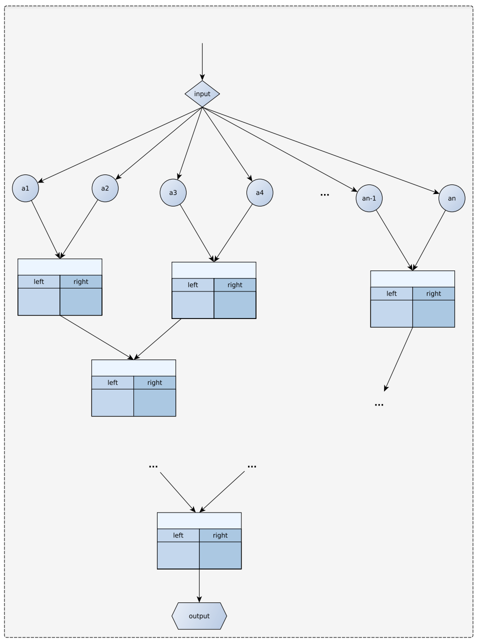

Fig. 1. Actors network built from a conjunction $A = \{a_1, \ldots, a_n\}$.

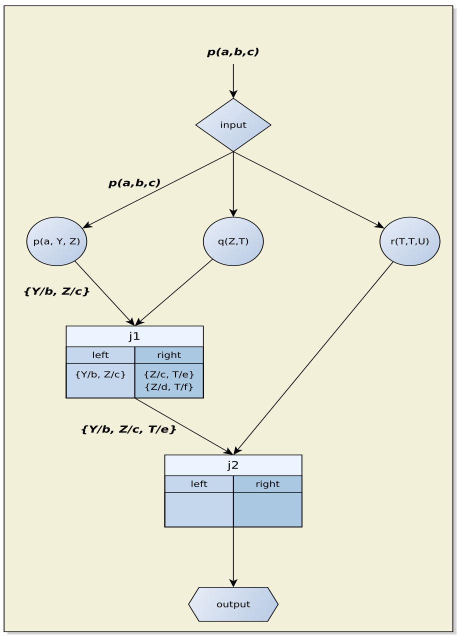

Fig. 2. Actors network built from the conjunction $A = \{p(a, Y, Z), q(Z, T), r(T, T, U)\}$. It illustrates how the message *p(a,b,c)* circulates in the network.

the input actor is the single input of the network. Each message it receives is a ground atom from B.

the substitution actor (represented by a circle): each one is associated with an atom a_i of A and is devoted to build substitutions from ground atoms it receives.

the join actor (represented by a rectangle) has two parents and is devoted to join the compatible substitutions provided by his parents. It has two internal memories (*left* and *right*) to store the substitutions provided by its left and right parents.

the output actor is the single output of the network. It receives the final substitutions (if they exist) establishing the θ-subsumption between A and B.

If we do not consider the input and output actors the network is a binary tree where the substitution actors are the leaves, the other nodes being join actors. A single join actor, the root of the tree, is linked with the output actor (Algorithm 1 presents the way we build the whole network). The network contains $2n + 1$ nodes so the time complexity to build the network is $O(n)$.

Algorithm 1 Building the actors network

```
buildNetwork($A$)                    /* $A = \{a_1, \ldots, a_n\}$ is a conjunction of $n$ literals */
begin
  create the output actor $out$ ;
  buildTree($out$, $n$) ;            /* build the tree of join and substitution actors */
  $S = \{s_1, \cdots, s_n\} \leftarrow$ leaves($out$) ;   /* get the substitution actors of the tree of root $root$ */
  create the input actor $in$ ;
  for $i \leftarrow 1$ to $n$ do
        link $s_i$ with $in$ ; set $a_i$ as internal label of $s_i$ ;
  end for
  return $in$ ;
end.

buildTree($j$, $n$)                  /* $j$: join (or output) actor of the preceding level */
begin
  if $n = 1$ then
        create a substitution actor $a$ and store it ;        /* a new leaf for the tree */
  else
        create a join actor $a$ ; buildTree($a$, $n/2 + n$ mod 2) ; buildTree($a$, $n/2$) ;
  end if
  link $a$ to $j$ ;
end.
```

Let us illustrate how such a network is used to check the subsumption through the network presented in Figure 2 built from the conjunction $A = body(C)\alpha = \{p(a, Y, Z), q(Z, T), r(T, T, U)\}$ where C is the clause given in Example 1. We also assume that the atoms of $B = body(D) = \{p(a, b, c), p(a, b, d), q(c, e), q(d, f), r(e, e, g), r(f, f, g), r(e, f, g)\}$ are sent to the network. Note that all the operations of the actors are made concurrently:

- When a ground atom b of B is provided to the *input actor*, this actor sends b to all *substitution actor*s associated with the atoms of A built from the same predicate as b. ex) the *input actor* receives $b = p(a, b, c)$, then it sends $p(a, b, c)$ to the actor $p(a, Y, Z)$.
- When a *substitution actor* associated with an atom a_i of A receives a ground atom b it checks if there exists a substitution μ_i such that $a_i\mu_i = b$. If μ_i exists the actor sends it to the single *join actor* with which it is linked. ex) $b = p(a, b, c)$ and $a_i = p(a, Y, Z)$ then the substitution $\mu_i = \{Y/b, Z/c\}$ is sent to the *join actor* j_1, if $b = p(e, b, c)$ no substitution is sent. The worst case complexity to check if there exists μ_i with $a_i\mu_i = b$ is $O(v.ln(v))$ where v is the common arity of a_i and b ($ln(v)$ is the worst case complexity to access any argument of a_i and b).
- When a *join actor* receives a substitution μ from its left (right) parent it first stores it in its left (right) internal memory. Then, it joins μ with each compatible substitution δ found in its right (left) memory and sends $\mu \cup \delta$ to its single successor. ex) the *join actor* j_1 receives $\mu = \{Y/b, Z/c\}$ from its left parent, then it stores it in its left memory. Assume the right memory of j_1 contains the substitutions $\delta_1 = \{Z/c, T/e\}$ and $\delta_2 = \{Z/d, T/f\}$. Thus $\mu \cup \delta_1 = \{Y/b, Z/c, T/e\}$ is sent to the successor of j_1, while $\mu \cup \delta_2 = \{Y/b, Z/c, Z/d, T/f\}$ is not considered because it is not a valid substitution. The complexity to check that μ and δ are compatible is $O(|\mu|.ln(|\mu)|)$ (we assume that μ and δ have approximately the same size). So the number of operations made by a join actor when it receives a message μ is in $O(m.|\mu|.ln(|\mu|))$ where m is the current size of the right (left) memory.
- When the *output actor* receives a substitution μ it displays μ as a solution ($A\mu \subseteq B$). If we do not want any other solutions, the process terminates, otherwise the actor waits for other substitutions.

We send all the atoms of B to the input actor. To ensure that the activity of the network stops we then send it a specific end-message. The end message is broadcasted through the network to the output actor which terminates the process if there is no solution. The model is correct and complete: each substitution reaching the output actor is a solution, and every possible solution can be outputed if the user wishes to (see `https://lipn.univ-paris13.fr/~leger/ilp2016.html` for proofs).

4 Experiments

Several programming languages implement the Actor Model. In this work we use the Akka [2] toolkit which is integrated to Scala [15], a multi-paradigms (mainly functional) programming language built on the Java Virtual Machine. The dataset was generated to hold a unique solution substitution. The subsumer consists of a single clause with 20 literals (20 distinct predicate symbols with an arity of 10). The subsumee holds 5 literals for each predicate symbol, giving a total of 100 literals. Note that we have run the test to only find the first possible solution. We have run the same subsumption test (with the same subsumer and subsumee) multiple times with a varying number of processor cores. This experiment was done using the Grid'5000 testbed, on an Intel Xeon E5-2660v2 CPU, with reservations ranging from 1 to 20 cores.

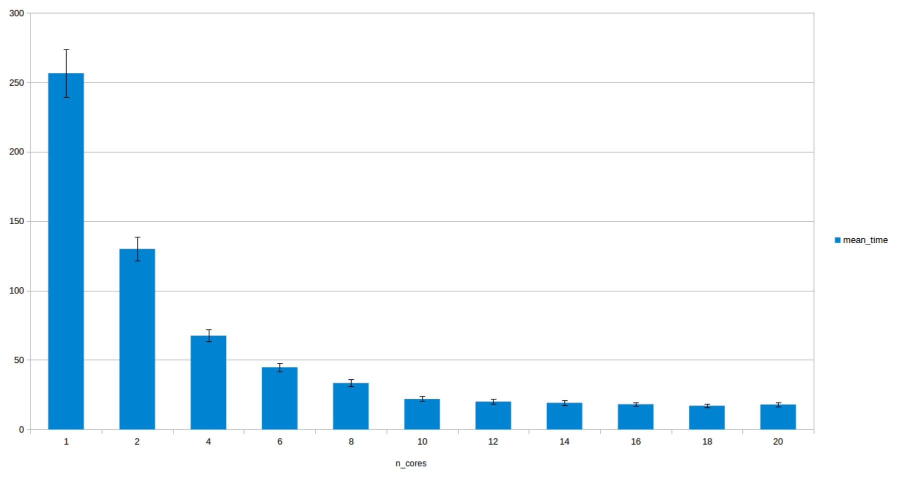

Fig. 3. Running time on different number of cores.

In Figure 3, we can see that the performance of the subsumption test increases along with the number of cores used. In this case the decrease in computation time eventually reaches a plateau, due to the data size. Please keep in mind that the number of actors is directly linked to the number of literals the hypothesis (subsumer) holds. This means that for a bigger clause, the cost-efficiency of parallelism would be higher.

5 Conclusion and perspectives

We have shown that Actor Modeling is indeed effective at reducing the running time of the θ-subsumption problem. Due to a lack of time we did not compare the implementation of our model with classical state-of-the-art solutions like Subsumer [17] or Resumer[10]. We must also refine our model by applying the major ILP optimizations, like clause partitioning and linked variables analysis. We have just implemented a distributed version of our model and started our first distributed experiment using the Grid'5000 testbed, supported by a scientific interest group hosted by Inria and including several Universities as well as other organizations (see https://www.grid5000.fr). Finally, we are investigating another Actor Model for scalable θ-subsumption where the actors are the subsumee's literals. This would lead to an increasing number of actors but an important reduction of the workload for each actor.

References

1. Gul Agha. An overview of actor languages. In *Proceedings of the 1986 SIGPLAN Workshop on Object-oriented Programming*, OOPWORK '86, pages 58–67, New York, NY, USA, 1986. ACM.
2. Jamie Allen. *Effective Akka*. O'Reilly Media, Inc., 2013.
3. Annalisa Appice, Michelangelo Ceci, Antonio Turi, and Donato Malerba. A parallel, distributed algorithm for relational frequent pattern discovery from very large data sets. *Intell. Data Anal.*, 15(1):69–88, 2011.
4. Stefano Ferilli, Nicola Mauro, Teresa M. A. Basile, and Floriana Esposito. *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, pages 1–13. Springer Berlin Heidelberg, Berlin, Heidelberg, 2003.
5. Nuno A. Fonseca, Fernando Silva, and Rui Camacho. *Strategies to Parallelize ILP Systems*, pages 136–153. Springer Berlin Heidelberg, Berlin, Heidelberg, 2005.
6. Nuno A. Fonseca, Ashwin Srinivasan, Fernando Silva, and Rui Camacho. Parallel ilp for distributed-memory architectures. *Machine Learning*, 74(3):257–279, 2009.
7. Georg Gottlob. Subsumption and implication. *Information Processing Letters*, 24(2):109 – 111, 1987.
8. Carl Hewitt, Peter Bishop, and Richard Steiger. A universal modular actor formalism for artificial intelligence. In *Proceedings of the 3rd International Joint Conference on Artificial Intelligence*, IJCAI'73, pages 235–245, San Francisco, CA, USA, 1973. Morgan Kaufmann Publishers Inc.
9. Jrg-Uwe Kietz and Marcus Lübbe. An efficient subsumption algorithm for inductive logic programming. In *Proceedings of the 11th International Conference on Machine Learning*, pages 130–138, 1994.
10. Ondrej Kuzelka and Filip Zelezn. A restarted strategy for efficient subsumption testing. *Fundam. Inform.*, 89(1):95–109, 2008.
11. John W. Lloyd. *Foundations of Logic Programming, 1st Edition*. Springer, 1984.
12. Jérôme Maloberti and Michèle Sebag. Fast theta-subsumption with constraint satisfaction algorithms. *Machine Learning*, 55(2):137–174, 2004.
13. S Muggleton and C Feng. Efficient induction of logic programs. pages 368–381, 1990.
14. Stephen Muggleton, José Santos, and Alireza Tamaddoni-Nezhad. *Inductive Logic Programming: 19th International Conference, ILP 2009, Leuven, Belgium, July 02-04, 2009. Revised Papers*, chapter ProGolem: A System Based on Relative Minimal Generalisation, pages 131–148. Springer Berlin Heidelberg, Berlin, Heidelberg, 2010.
15. Martin Odersky and al. An Overview of the Scala Programming Language. Technical Report IC/2004/64, EPFL, Lausanne, Switzerland, 2004.
16. J. A. Robinson. A machine-oriented logic based on the resolution principle. *J. ACM*, 12(1):23–41, January 1965.
17. Jose Santos and Stephen Muggleton. Subsumer: A Prolog theta-subsumption engine. In Manuel Hermenegildo and Torsten Schaub, editors, *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.
18. Ashwin Srinivasan. The aleph system. 1987.
19. Ashwin Srinivasan, Tanveer A. Faruquie, and Sachindra Joshi. Data and task parallelism in ilp using mapreduce. *Machine Learning*, 86(1):141–168, 2012.