When does it payoff to use sophisticated entailment engines in ILP?

José Santos and Stephen Muggleton

Department of Computing, Imperial College London
Email: {jcs06,shm}@doc.ic.ac.uk

Abstract. Entailment is an important problem in computational logic particularly relevant to the Inductive Logic Programming (ILP) community as it is at the core of the hypothesis coverage test which is often the bottleneck of an ILP system. Despite developments in resolution heuristics and, more recently, in subsumption engines, most ILP systems simply use Prolog's left-to-right, depth-first search selection function for SLD-resolution to perform the hypothesis coverage test.
We implemented two alternative selection functions for SLD-resolution: smallest predicate domain (SPD) and smallest variable domain (SVD); and developed a subsumption engine, Subsumer. These entailment engines were fully integrated into the ILP system ProGolem.
The performance of these four entailment engines is compared on a representative set of ILP datasets. As expected, on determinate datasets Prolog's built-in resolution, is unrivalled. However, in the presence of even little non-determinism, its performance quickly degrades and a sophisticated entailment engine is required.

Keywords: Entailment engines, Coverage testing, SLD-resolution

1 Introduction and motivation

Inductive Logic Programming (ILP) systems construct hypotheses from a rich hypothesis language and thus have to traverse a large hypothesis search space. This search requires having to test some metric of the candidate hypothesis on the provided examples. A metric typically used is coverage: positive examples covered minus negative examples covered. Evaluating coverage of a single candidate hypothesis requires thus, potentially, testing the coverage of the candidate clause on all training examples. Moreover, each one of these coverage tests can be very expensive to compute as it is a query in first-order logic.

This problem is well known to ILP researchers and several techniques have been proposed to alleviate it. Just to name a few, these techniques range from combining queries in query packs [1] to take advantage of the similar structure of the candidate clauses, transforming the clause before execution [4] so that the transformed clause is more efficient to evaluate, improving the indexing mechanism [3] of the Prolog engine, to stochastic estimation of the clause coverage [14], [7].

Another approach to improve the coverage test efficiency is to use a custom resolution engine instead of Prolog's left-to-right, depth-first search implementation of SLD-resolution. In the 1980's there was extensive research on this subject. In [15] it is noted that Prolog's default evaluation order of goals in a clause can lead to intolerable inefficiencies. The authors, motivated by a AI planning application, propose a "cheapest-first" heuristic that is akin to ours smallest predicate domain, letting the resolution engine choose, during evaluation, the predicate that has fewer solutions. They also recognize the potential overhead of this re-ordering procedure Later, [10] even proposes a machine learning approach to automatically decide the goal order in a query.

In some scenarios a full resolution engine is not needed (see section 2.1 for a discussion) and one can do the coverage test with a θ-subsumption engines. Subsumption engines optimized to perform subsumption efficiently on complex non-determinate clauses, have been developed recently, e.g. Django [9], Resumer2 [8] and Subsumer [13].

In section 2.2 we present two alternative heuristics for SLD-resolution. These heuristics, together with the subsumption engine Subsumer and Prolog's built-in SLD-resolution were integrated in the ILP system ProGolem [11]. By empirically evaluating the performance of these entailment engines on a representative set of ILP problems, we try to characterize the properties an ILP problem must have for it to payoff to use sophisticated entailment engines.

This characterization has immediate applicability to ProGolem as it already implements these four entailment engines and could choose dynamically the most suitable algorithm for each pair $\langle hypothesis, example \rangle$.

2 The θ-subsumption problem

θ-subsumption is an incomplete approximation to logical implication [12]. While implication is undecidable in general, θ-subsumption is a NP-complete problem [5]. A clause C θ-subsumes a clause D ($C \vdash_\theta D$) if and only if there exists a substitution θ such that $C\theta \subseteq D$.

Example 1. $C : h(X_0) \leftarrow l1(X_0, X_1), l1(X_0, X_2), l1(X_0, X_3), l2(X_1, X_2), l2(X_1, X_3)$
$D : h(c_0) \leftarrow l1(c_0, c_1), l1(c_0, c_2), l2(c_1, c_2)$
$C\theta$ subsumes D with $\theta = \{X_0/c_0, X_1/c_1, X_2/c_2, X_3/c_2\}$.

The θ-subsumption problem is thus, given two clauses, C and D, find a substitution θ such that all literals of C can be mapped into a subset of the literals of D.

Prolog performs entailment using SLD-resolution [6] which is, in general, stronger than pure subsumption (see 2.1). Within SLD-resolution all mappings from the literals in C onto the literals in D (for the same predicate symbol) are constructed left-to-right in a depth-first search manner.

As all Prolog programmers know, the order of the literals in C has a significant impact on the (in)efficiency of the query evaluation.

2.1 Subsumption versus resolution

Selective Linear Definite clause resolution (SLD-resolution) is the inference rule in logic programming. It allows the Prolog interpreter to derive all logical consequences of a query. In order to use subsumption to decide if an example e is covered by a clause C, one needs to encode all literals related to that example in a single saturated clause S_e (see below). When used to implement ILP's coverage test, θ-subsumption generates the same solutions as SLD-resolution when the underlying prolog program (i.e. background knowledge in the ILP setting) is pure Prolog.

If the background knowledge contains non-pure Prolog constructs (e.g. non-constructive arithmetic operators, cuts, ...) subsumption will only find a subset, usually empty, of the solutions that SLD-resolution finds.

Unfortunately many real-world ILP datasets express their background knowledge in non-pure Prolog. Often the problem lies with real number arithmetic. For instance, consider the program in Figure 1.

```
:- modeh(1, active(+molecule)).          active(mol1).  logp(mol1, 3.14).
:- modeb(1, logp(+molecule,-real)).      gteq(X, X):- !.
:- modeb(1, gteq(+real,#real)).          gteq(X, Y):- X>=Y.
```

Fig. 1. Simple ILP program with non-pure background knowledge

The saturated clause for $active(mol1)$ is $active(mol1) \leftarrow logp(mol1, 3.14)$, $gteq(3.14, 3.14)$. Suppose now we have an hypothesis $active(X) \leftarrow logp(X, Y)$, $gteq(Y, 3.05)$. This hypothesis does not subsume the ground bottom clause as there is no literal $gteq(3.14, 3.05)$ in it. However, were we to use SLD-resolution we would be able to prove the hypothesis with the binding $X = mol1, Y = 3.14$.

The culprit of the problem is that the ground bottom clause did not capture the full information available in the $gteq/2$ clause. There are two problems, the cut in the first $gteq/2$ clause prevents retrieving more solutions, to the ground bottom clause of $mol1$, when the second argument is unbound or equal to the first argument. The most serious problem is that the $>=$ comparison operator is not constructive. That is, $>= /2$ requires both arguments to be instantiated, not returning in backtracking numbers that verify the condition when one or both of the arguments are unbound.

In situations like these one cannot use a θ-subsumption engine but need instead a resolution engine.

Throughout this paper we sometimes use the terms entailment, subsumption and resolution almost interchangeably although they are not equivalent. This abuse of terminology is justified because, for the purpose of our experiments, those expressions are equivalent. From the perspective of an ILP system, what matters is whether a clause covers (i.e. entails) an example or not. Both subsumption and resolution engines perform this entailment test with the same result as long as the background knowledge is pure Prolog.

2.2 Entailment algorithms in ProGolem

ProGolem implements four entailment engines. Three are variants of SLD-resolution and one is the subsumption engine Subsumer described in [13]. The three variants of SLD-resolution are Prolog's built-in left-to-right depth-first search heuristic for SLD-resolution (hereafter Left-to-right) and two alternative selection functions for SLD-resolution. Smallest Predicate Domain (SPD-resolution for simplicity) and Smallest Variable Domain (SVD-resolution).

In SPD-resolution the literal with fewest number of solutions at each moment is picked. Note that in Prolog the literals are always picked left-to-right in the order given in the clause. This is the same as the "cheapest-first" heuristic described in [15].

SVD-resolution is more sophisticated, it computes the consistent domains of each variable and at each moment binds the variable with smallest domain with one of its possible values.

Subsumer improves upon SVD-resolution by decomposing a clause dynamically in independent components but is no longer a resolution engine. It is a θ-subsumption engine requiring the subsumee clause (the example) to be given as a ground bottom clause. Note that in this case the background knowledge is only used once to create the ground bottom clauses and is never called during a subsumption test (see Section 2.1).

2.3 Time complexity

Let N be the length of an hypothesis H and M be the length of an example E. The worst case complexity of SLD/SPD-resolution is $O(M^N)$ as we need to map each literal of H (ranging from $1..N$) to a literal in E (ranging from $1..M$).

In practice, since the SLD/SPD-resolution tests the consistency of the matching while constructing the substitution (thus bounding other variables) and not just at the end, for clauses C with too many literals (i.e. $M \approx N$) the subsumption problem may become overconstrained and thus be easier than when M is a fraction of N.

An alternative way, employed by SVD-resolution and Subsumer, to tackle the subsumption problem is to map variables of H to terms in E rather than literals to literals. Let V be the set of distinct variables in H and T the set of distinct terms in E. We can map the θ-subsumption problem to the problem of finding a mapping from V to T. This approach has worst case complexity $O(|T|^{|V|})$.

However, it is not easy in practice to antecipate whether the literal or variable mapping works better as the average case complexity depends essentially on how constrained the search gets when a literal or a variable is bound. Note that when we map a literal all its variables get bound at the same time.

3 Empirical evaluation

In this section we extensively compare the four entailment engines described in Section 2.2. We have not used Django or Resumer2 as it would not be practi-

cal and, as shown in [13], Subsumer is a good representative of sophisticated subsumption engines. It outperforms Django and is competitive with Resumer2. ProGolem with all the datasets and scripts to replicate these experiments can be found at: http://www.doc.ic.ac.uk/∼jcs06/papers/ilp10.

3.1 Materials and Methods

The ProGolem ILP system [11] was used with a representative set of well-known ILP datasets to generate hypotheses. Datasets PT.02, PT.15 and PT.31 are less known. These are problems 02, 15 and 31 of the Phase Transition (PT) framework [2], representing instances from the Yes, No and PT regions.

ProGolem is a bottom-up ILP engine that, among many other settings, allows the user to choose which entailment engine to use. Since we wanted to use a θ-subsumption engine, only pure Prolog was allowed in the background knowledge. That meant removing or disabling cuts and non-constructive arithmetic operators in some of datasets' (e.g. mutagenesis) background knowledge.

For the resolution algorithms the examples are provided in the background knowledge as usual in ILP. For the subsumption engine each example is a single (saturated) clause with all facts known to be true about it. Below is a small excerpt of a ground bottom clause for the mutagenesis dataset. The full clause has 77 literals.

$$\begin{aligned} active(d112) \leftarrow\ & atm(d112, d112\_9, h, 3, 0.136), atm(d112, d112\_8, h, 3, 0.136), \ldots, \\ & atm(d112, d112\_1, c, 22, -0.125), bond(d112, d112\_6, d112\_9, 1), \ldots, \\ & bond(d112, d112\_1, d112\_7, 1), bond(d112, d112\_1, d112\_2, 7). \end{aligned}$$

Table 1 summarizes important statistics on the datasets used. The columns are: number of examples, average example length, average number of distinct predicate symbols per example, average number of solutions per predicate symbol (assuming its input variables are bound) and average number of distinct terms per example. The latter four columns have the respective standard error associated. The figures in Table 1 were generated by computing the full ground clauses for each example in each dataset.Recall is the maximum number of alternative solutions a predicate may return.

As can be seen from Table 1, from the eight datasets selected, three are highly non-determinate (PT.XX) with exactly 100 solutions per distinct predicate symbol. Datasets Alzheimers-amine, Proteins and Pyrimidines are non-determinate with each predicate symbol having at most one solution. Carcinogenesis and Mutagenesis have a medium degree of non-determinism.

ProGolem was also used to induce theories for these datasets with all the intermediate hypotheses being collected to be later evaluated by the different entailment engines. When inducing theories, ProGolem's recall was set to 20. This is to limit the complexity of the hypotheses generated.

Since ProGolem is a bottom-up ILP system it is biased towards generating longer clauses. However, because some of these datasets are rather simple and all hypotheses were collected (including ones after negative reduction), many short

Dataset	$\|Ex\|$	Examples Len.	Pred. Symb.	Sols per P. S.	Terms per Ex.
Alz-amine	686	31±0	20±0	1±0	23±0
Carcinogenesis	298	115±4	11±0	5±1	54±1
Mutagenesis	188	83±2	2±0	41±2	48±1
Proteins	2028	287±1	42±0	1±0	36±0
Pyrimidines	2788	50±0	10±0	1±0	22±0
PT.02	400	701±0	7±0	100±0	20±0
PT.15	400	1503±1	15±0	100±0	39±0
PT.31	400	804±0	8±0	100±0	28±0

Table 1. Relevant statistics for the examples used per dataset

hypotheses were generated as well. Many of those could have been generated by a classical top-down ILP system like Aleph or Progol. For instance, one of the simpler hypothesis generated for the mutagenesis dataset was $active(A) \leftarrow bond(A, B, C, 1), bond(A, C, D, 2)$.

Table 2 summarizes the information on the hypotheses collected. The columns have an identical meaning to Table 1 except that column "Literals per Predicate Symbol" is the average number of times a given (distinct) predicate symbol appears on the hypothesis. Note that in a hypothesis the terms are usually variables and not just constants or function symbols.

Dataset	$\|Hyps\|$	Hypotheses Len.	Pred. Symb.	Lits per P. S.	Terms per Hyp.
Alz-amine	328	28±1	18±0	1±0	21±0
Carcinogenesis	161	43±3	6±0	4±0	29±2
Mutagenesis	382	43±1	2±0	21±1	33±0
Proteins	464	75±3	19±0	3±0	21±0
Pyrimidines	1730	42±0	10±0	4±0	32±0
PT.02	444	131±8	5±0	24±0	20±0
PT.15	68	163±32	7±1	25±1	36±1
PT.31	156	119±13	5±0	23±0	27±0

Table 2. Relevant statistics for the hypothesis used per dataset

3.2 Results and discussion

Each entailment engine was used to test the Boolean coverage of a random sample of 20.000 pairs $\langle hypothesis, example \rangle$ from each dataset. Table 3 presents the average times, with respective standard errors, in milliseconds, per subsumption test. Whenever the subsumption test required more than 5 seconds it was aborted. The "Timeout" column has the percentage of subsumption tests in these circumnstances. To compute the average time all the timed out tests were ignored.

Dataset	Entailment engines							
	Left-to-right		SPD-resolution		SVD-resolution		Subsumer	
	Avg time	Timeout	Avg time	Timeout	Avg time	Timeout	Avg time	Timeout
Alz-amine	0.0±0.0	0.00%	0.1±0.0	0.00%	0.3±0.0	0.00%	0.9±0.0	0.00%
Carcinogenesis	3.2±0.5	0.45%	0.5±0.1	0.01%	0.8±0.1	0.00%	1.8±0.3	0.01%
Mutagenesis	224±5.0	36.9%	19±1.4	0.27%	35±1.8	0.74%	9.9±0.8	0.03%
Proteins	0.1±0.0	0.00%	0.4±0.0	0.00%	21±0.1	0.00%	8.8±0.0	0.00%
Pyrimidines	0.1±0.0	0.00%	0.2±0.0	0.00%	0.3±0.0	0.00%	1.9±0.0	0.00%
PT.02	1987±9.6	98.8%	721±8.0	25.8%	421±6.3	8.53%	26±0.3	0.00%
PT.15	771±8.9	97.7%	360±6.2	64.3%	327±6.1	60.3%	142±2.5	0.37%
PT.31	2289±9.9	98.8%	405±6.1	52.1%	543±7.6	43.3%	76±1.4	0.03%

Table 3. Entailment average times with respective standard error, in ms, per dataset per entailment engine

Table 3 shows large differences in the entailment test costs on the non-determinate datasets. On the determinate datasets Prolog's left-to-right implementation of SLD-resolution is unrivalled but the time required by SPD-resolution is still competitive. As the degree of non-determinism grows, so does the advantage of Subsumer compared with the other entailment engines. It is important to note that Subsumer rarely timed out. However Subsumer's main drawback is its overhead on the determinate datasets and being unable to handle non-pure background knowledge.

4 Conclusions and future directions

Prolog's built-in left-to-right, depth-first search selection function for SLD-resolution is unrivalled on determinate datasets. However, when the dataset is even mildly non-determinate, Prolog's built-in resolution should not be used as the performance rapidly degrades and a large fraction of the entailment tests time out. For medium to highly non-determinate datasets Subsumer should be used. However, Subsumer is only applicable if the background knowledge is pure Prolog. If that is not the case then SPD-resolution should be employed.

These conclusions are not specific to ProGolem. They are valid for top-down ILP systems as well. Therefore it would be beneficial to integrate at least SPD-resolution and Subsumer in other ILP systems, e.g. Aleph.

It could be interesting to study if there are performance gains in using a specific entailment engine per pair $\langle hypothesis, example \rangle$ or whether looking at global properties of the dataset is enough to choose the best engine.

To fully take advantage of these powerful entailment engines on complex non-determinate problems such as the Phase Transition framework [2] one needs to improve the search control strategy of the ILP system. Being able to explore complex hypotheses is a necessary condition but is only half the way to enable ILP systems to learn theories on complex non-determinate domains.

Acknowledgments

The first author thanks Wellcome Trust for his Ph.D. scholarship. The second author thanks the Royal Academy of Engineering and Microsoft for funding his present 5 year Research Chair. We are indebted to three anonymous referees for valuable comments.

References

1. Hendrik Blockeel, Luc Dehaspe, Bart Demoen, Gerda Janssens, Jan Ramon, and Henk Vandecasteele. Improving the efficiency of Inductive Logic Programming through the use of query packs. *J. Artif. Intell. Res. (JAIR)*, 16:135–166, 2002.
2. Marco Botta, Attilio Giordana, Lorenza Saitta, and Michèle Sebag. Relational learning as search in a critical region. *Journal of Machine Learning Research*, 4:431–463, 2003.
3. Vítor Santos Costa, Konstantinos F. Sagonas, and Ricardo Lopes. Demand-driven indexing of prolog clauses. In Verónica Dahl and Ilkka Niemelä, editors, *ICLP*, volume 4670 of *Lecture Notes in Computer Science*, pages 395–409. Springer, 2007.
4. Vítor Santos Costa, Ashwin Srinivasan, Rui Camacho, Hendrik Blockeel, Bart Demoen, Gerda Janssens, Jan Struyf, Henk Vandecasteele, and Wim Van Laer. Query transformations for improving the efficiency of ILP systems. *Journal of Machine Learning Research*, 4:465–491, 2003.
5. Deepak Kapur and Paliath Narendran. Np-completeness of the set unification and matching problems. In Jörg H. Siekmann, editor, *CADE*, volume 230 of *Lecture Notes in Computer Science*, pages 489–495. Springer, 1986.
6. Robert A. Kowalski and Donald Kuehner. Linear resolution with selection function. *Artif. Intell.*, 2(3/4):227–260, 1971.
7. Ondrej Kuzelka and Filip Zelezný. Fast estimation of first-order clause coverage through randomization and maximum likelihood. In William W. Cohen, Andrew McCallum, and Sam T. Roweis, editors, *ICML*, volume 307 of *ACM International Conference Proceeding Series*, pages 504–511. ACM, 2008.
8. Ondrej Kuzelka and Filip Zelezný. A restarted strategy for efficient subsumption testing. *Fundam. Inform.*, 89(1):95–109, 2008.
9. Jérôme Maloberti and Michèle Sebag. Fast theta-subsumption with constraint satisfaction algorithms. *Machine Learning*, 55(2):137–174, 2004.
10. Shaul Markovitch and Paul D. Scott. Automatic ordering of subgoals - a machine learning approach. In *NACLP*, pages 224–240, 1989.
11. Stephen Muggleton, Jose Santos, and Alireza Tamaddoni-Nezhad. ProGolem: a system based on relative minimal generalisation. In Luc De Raedt, editor, *Proceedings of the 19th International Conference on ILP*, Lecture Notes in Computer Science, pages 131–148, Leuven, Belgium, 2009. Springer.
12. John Alan Robinson. A machine-oriented logic based on the resolution principle. *J. ACM*, 12(1):23–41, 1965.
13. Jose Santos and Stephen Muggleton. Subsumer: A Prolog theta-subsumption engine. In *Technical communications of the 26th Int. Conference on Logic Programming, Leibniz International Proc. in Informatics*, Edinburgh, Scotland, 2010.
14. Michèle Sebag and Céline Rouveirol. Tractable induction and classification in first order logic via stochastic matching. In *IJCAI (2)*, pages 888–893, 1997.
15. David E. Smith and Michael R. Genesereth. Ordering conjunctive queries. *Artif. Intell.*, 26(2):171–215, 1985.