Technical Communications of the International Conference on Logic Programming, 2010 (Edinburgh), pp. 172–181
http://www.floc-conference.org/ICLP-home.html

SUBSUMER: A PROLOG θ-SUBSUMPTION ENGINE

JOSÉ SANTOS [1] AND STEPHEN MUGGLETON [1]

[1] Department of Computing, Imperial College London
E-mail address: {jcs06,shm}@doc.ic.ac.uk

Abstract. State-of-the-art θ-subsumption engines like Django (C) and Resumer2 (Java) are implemented in imperative languages. Since θ-subsumption is inherently a logic problem, in this paper we explore how to efficiently implement it in Prolog.

θ-subsumption is an important problem in computational logic and particularly relevant to the Inductive Logic Programming (ILP) community as it is at the core of the hypotheses coverage test which is often the bottleneck of an ILP system. Also, since most of those systems are implemented in Prolog, they can immediately take advantage of a Prolog based θ-subsumption engine.

We present a relatively simple ($\approx$ 1000 lines in Prolog) but efficient and general θ-subsumption engine, Subsumer. Crucial to Subsumer's performance is the dynamic and recursive decomposition of a clause in sets of independent components. Also important are ideas borrowed from constraint programming that empower Subsumer to efficiently work on clauses with up to several thousand literals and several dozen distinct variables.

Using the notoriously challenging Phase Transition dataset we show that, cputime wise, Subsumer clearly outperforms the Django subsumption engine and is competitive with the more sophisticated, state-of-the-art, Resumer2. Furthermore, Subsumer's memory requirements are only a small fraction of those engines and it can handle arbitrary Prolog clauses whereas Django and Resumer2 can only handle Datalog clauses.

1. Introduction and motivation

Current state-of-the-art ILP systems are usually developed in Prolog, e.g. Aleph [Sri07] and ProGolem [Mug09], mainly because many of the algorithms needed for an ILP system are already built-in in a Prolog engine (e.g. unification, backtracking, SLD-resolution).

However, for complex learning problems where predicates are highly non-determinate and the target concept size is large ($>$ 10 literals), the Prolog's built-in SLD-resolution is inadequate. In these situations there is a combinatorial explosion of alternative variable bindings and consequently it will often take too long for the Prolog engine to decide whether the given goal succeeds. This is unacceptable for an ILP system as there will be, typically, tenths to hundredths of thousands such complex goals (i.e. putative hypothesis) that need to be evaluated before a final theory is proposed.

The subsumption problem at the culprit of the ILP bottleneck has not received much attention because, for many ILP applications, Prolog's built-in resolution seems to suffice. However, due to the non-determinism explosion highlighted above, ILP researchers often

1998 ACM Subject Classification: I.2.3 Deduction and Theorem Proving.
Key words and phrases: Theta-subsumption, Prolog, Inductive Logic Programming.

Technical Communications of the 26th International Conference on Logic Programming, Edinburgh, July, 2010
Editors: Manuel Hermenegildo, Torsten Schaub
LIPIcs - Leibniz International Proceedings in Informatics. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, Germany
Digital Object Identifier: 10.4230/LIPIcs.ICLP.2010.172

have to bound the maximum hypotheses length and recall (i.e. number of solutions per predicate) to relatively small values, which may be preventing better theories to be found.

In the last few years two efficient subsumption engines, Django [Mal04] and Resumer2 [Kuz08], were developed. However these are complex engines, around 10.000 lines of source code each, implemented in C and Java respectively, making them unpractical to use within a Prolog based ILP system. More importantly, both those engines require substantial amounts of memory, sometimes 10x more memory than the ILP system itself for the same data. This limits considerably their applicability given that, for challenging problems, the ILP system already consumes a sizeable portion of the system's resources.

The motivation for Subsumer was to develop a simple, lightweight, fully general Prolog subsumption engine that could be easily integrated from any Prolog application and, in particular, Prolog implementations of ILP systems.

2. The θ-subsumption problem

θ-subsumption [Rob65] is an approximation to logical implication. While implication is undecidable in general θ-subsumption is a NP-complete problem [Kap86]. A clause $C\theta$-subsumes a clause D ($C \vdash_\theta D$) if and only if there exists a substitution θ such that $C\theta \subseteq D$.

Example 2.1 (θ-subsumption)**.**
$C : h(X_0) \leftarrow l1(X_0, X_1), l1(X_0, X_2), l1(X_0, X_3), l2(X_1, X_2), l2(X_1, X_3)$
$D : h(c_0) \leftarrow l1(c_0, c_1), l1(c_0, c_2), l2(c_1, c_2)$
$C\theta$ 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.

The standard algorithm for θ-subsumption is based on Prolog's SLD-resolution [Kow71]. 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. Note that the order of the literals in C has a significant impact on SLD-resolution (in)efficiency.

2.1. θ-subsumption time complexity

Let N and M be the lengths of clauses C and D. The standard θ-subsumption algorithm has complexity $O(M^N)$ as we need to map each literal of C (ranging from $1..N$) to a literal in D (ranging from $1..M$).

In practice, since SLD-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.

Let V be the set of distinct variables in C, and T the set of distinct terms in D. The θ-subsumption problem is then equivalent to do a mapping from V to T. This approach has complexity $O(|T|^{|V|})$ which is generally better than $O(M^N)$ since usually the clauses we are interested have $|T| << M$ and $|V| << N$. Django, Resumer2 and Subsumer all use this latter mapping.

```
1.  solve_component(VarsInComp, VarsConstr)
2.    if VarsInComp is empty then
3.      return true
4.    Let V = most_promising_free_variable(VsInComp, VsConstr)
5.    Let SubVsInComps = decomp_comp(VsInComp, VsConstr, V)
6.    Let V_Neighbours = free vars sharing a literal with V
7.    for each value Val in V's domain
8.    do
9.      V=Val;
9.      Let NVsConstr = update_vars_domains(V_Neighbours, VsConstr)
10.     for each component VComp in SubVsInComps
11.     do
12.       solve_component(VComp, NVsConstr)
13.     done
14.   done
15.   return false
16. end solve
```

Figure 1: Pseudo-code for Subsumer's main algorithm

3. Subsumer: A Prolog θ-subsumption engine

Subsumer is a publicly available (`http://ilp.doc.ic.ac.uk/Subsumer`), simple ($\approx$ 1000 lines of Prolog) fully general θ-subsumption engine with the expected behaviour from a Prolog implementation as it does not need to keep state. The Subsumer library exports a predicate, $theta\_subsumes(+subsumer, +subsumee)$, that either fails or succeeds. In case of success the variables in the subsumer clause are bound with the corresponding terms/variables of the subsumee and all possible solutions are returned by backtracking.

3.1. Main algorithm

Subsumer's main algorithm (Fig 1) works by at each iteration finding the most "promising" free (i.e still unbound) variable, V, to bound from the current component. Note that a component is defined solely by the variables appearing on it. The current heuristic is to pick the variable with smallest domain. Then the current component is decomposed assuming V has been bound (line 5). The components are returned in increasing order of their number of variables. In that way smaller components, which in principle are easier to test, are evaluated before longer ones. This can speed up the overall subsumption test significantly in case no solution is found for those smaller components.

In line 7 we iterate over the possible values for V's domain and in line 9 update its neighbour variables domain. This neighbour variable domain update is the most expensive part of Subsumer's algorithm but, due to space restrictions, we will be not be able to go into detail here. Essentially, it is implemented with a sophisticated indexing and back-indexing datastructure, that allows efficient assignment of a value to a variable and respective propagation of its new value to its direct interacting variables.

Each time the domain for a neighbour of V becomes inconsistent we have to backtrack and assign a different value to V. Although this can be particularly lengthy and get to several levels of deep recursion before a backtracking occurs, it works well in practice.

Also note that this algorithm is natural to parallelize. The natural place is the "for each loop" in line 8 where we could evaluate several components in parallel. This type of

parallelization has the peculiar property of possibly achieving superlinear (in the number of cores) speedups in case the subsumption test fails. This is because if a thread evaluating a component fails, all the other component evaluation threads running in parallel can stop immediately as there will be no solution for the whole clause. Unfortunately, however, implementing this parallel algorithm is not easy with current Prolog compilers [1].

3.2. Datastructures

The subsumer clause, $C = h \leftarrow b_1, .., .., b_n$ is represented as a list of literals. The hypothesis is preprocessed to gather all the distinct (upon variable renaming) calling patterns for the existing predicate symbols. E.g. $l1(X_0), X_1$ and $l1(X_1, X_2)$ have the same calling pattern but $l1(X_0, X_1)$ and $l1(X_0, X_0)$ are distinct.

The subsumee clause, $D = e \leftarrow g_1, .., .., g_n$ is given as a list of ground literals representing everything known to be true about e (it is the ground bottom clause of e with recall set to infinity). The example is preprocessed so that we just keep for each distinct predicate symbol $p_{s/a}$ (i.e. PredicateName/Arity) its available list of values $Val(p_{s/a})$, that is the predicate symbol domain. For instance, we would compactly represent clause D in Example 2.1, as $\{l1/2 : [\langle c_0, c_1\rangle, \langle c_0, c_2\rangle], l2/2 : [\langle c_0, c_2\rangle]\}$.

The space needed to store clause's D related information is thus: $O(\sum_1^N Val(p_{s/a_i}))$ where N is the number of distinct predicate symbols in D. A necessary condition for subsumption is that all distinct predicate symbols in C also exist in D.

The variables are extracted from C and their initial domain is computed. The initial domain for a variable is the intersection of its individual domains in each of the unique calling patterns it occurs. For instance, we the initial domains for clause C when subsuming clause D in Example 2.1, is $X_0 \in \{c_0\}, X_1 \in \{c_1\}, X_2 \in \{c_2\}, X_3 \in \{c_2\}$.

All direct pairwise variable interactions are also stored. A variable v_1 directly interacts with another variable v_2 iff they share the same literal in C. For instance, we have the following variable interactions for clause C in Example 2.1: $X_0 : \{X_1, X_2, X_3\}, X_1 : \{X_0, X_2, X_3\}, X_2 : \{X_0, X_1\}, X_3 : \{X_0, X_3\}$.

We also have a datastructure that, for each variable, holds the indexs of the literals where the variable occurs in the clause (clause's head being index 1). For the same clause C from Example 2.1 we then have $X_0 : [1, 2, 3, 4], X_1 : [2, 5, 6], X_2 : [3, 5], X_3 : [4, 6]$.

3.3. Clause decomposition

The dominant factor for reduced time complexity in Subsumer is clause decomposition. Let $H = h \leftarrow b_1, .., b_i, .., b_N$ and suppose literal b_i succeeds $k_i > 0$ times. The worst case number of predicate calls is $\prod_1^N k_i$ which, assuming an average branching factor, b, of solutions per literal leads to a $O(b^N)$ time complexity. For non-determinate clauses (i.e. clauses having literals with $b > 1$) this becomes untractable for relatively small N.

However, when the clause is decomposable in K groups of independent literals the complexity drops from $O(b^N)$ to $\sum_1^K O(b^{N_{g_i}})$, which is $O(b^{maxN_{g_i}})$. The worst case is now only exponential in the size of the longest group rather than the whole clause size.

[1] There are two problems: efficiency and transparency. From our experience, managing the threads explicitly in YAP is inefficient and also obfuscates the structure of the algorithm underneath. The ideal situation would be for Prolog compilers to have native parallel versions of list processing libraries (predicate checklist/2 in library(apply_macros) is the relevant one here).

The reasoning is then applied recursively to the newly found subcomponents. This idea, named once-transformation, was initially presented in [Cos03]. In Subsumer we implement a variant of it with several important differences. In the once-transformation the clause was transformed and independent literals were embedded in once/1 calls. The transformed clause was then called by the Prolog engine. In our approach, the clause is not transformed and our unit of evaluation are the distinct logical variables in a component, not a literal.

Two clause components are independent if, and only if, they do not share any (free) variable. Note that a clause is only satisfiable if all its components are. Thus if one component has no solutions then there is no solution for the whole clause. Equally importantly, the different solutions (θ-substitutions) of a component do not impact the solutions of the remaining components meaning that we can safely skip to the next component as soon as a solution for the current component has been found.

Example 3.1. $h(X) \leftarrow a(X,Y), b(X,Z), c(Y,A), d(Y,B), e(Z,C), f(Z,D)$

In Example 3.1 all variables are connected and thus the whole clause is a single component. However, when variable X becomes bound, literals $a(x,Y), c(Y,A), d(Y,B)$ belong to one component and literals $b(x,Z), e(Z,C), f(Z,D)$ to another. They are independent of each other as they do not share any common variable. This type of decomposition, when the head variables are assumed ground, is called the cut-transformation in [Cos03]. Resumer2 does this level of decomposition whereas Django does not do any form of clause decomposition.

In Subsumer this decomposition is applied recursively. If variable Y becomes bound next, then component $a(x,y), c(y,A), d(y,B)$ can be further divided into two components $c(y,A)$ and $d(y,B)$. Literal $a(x,y)$ no longer appears as it is now fully ground and thus no longer belongs to a component.

Also significantly, in Subsumer the independent components are created dynamically rather than statically at the beginning of clause evaluation. Although this has an overhead, it allows to choose the variable with the smallest domain (or another promising heuristic) as the splitting variable rather than, as in the once-transformation, an arbitrary variable where no information about its goodness exist. The costs of doing the decomposition dynamically should be more than offset by minimizing early the domain of the variable used.

3.4. Related engines

There are only two other subsumption engines comparable with Subsumer in terms of the complexity of clauses they can handle: Django [Mal04] and Resumer2 [Kuz08].

Common to the three engines are algorithms inspired by the constraint satisfaction framework. All do some custom form of arc-consistency and propagate constraints. Django and Resumer2 require particularly large quantities of memory as they perform determinate matching between the literals in the subsumer clause and the literals in the subsumee prior of starting its normal non-determinate matching.

Determinate matching is an idea originally presented in [Kie94], where signatures (fingerprints) of a literal are computed taking into account its neighbours (i.e. variables and literals it interacts). If the same unique fingerprint exists on both clauses for a given pair of literals these can be safely matched. Django computes these signatures with second level neighbours whereas Resumer2 uses only first level neighbours. This explains partially why

Django requires even more memory than Resumer2. Subsumer does not perform any form of determinate matching.

Django default variable ordering heuristic is the minimal variable domain divided by the number of variable interactions. In Resumer2 each variable is assigned a weight equal to its number of interactions divided by its domain size and then variables are selected with probability proportional to their weight. Subsumer uses simply minimal variable domain. Django also has a meta layer where it tries to adapt its heuristics to the underlying dataset. Resumer2 main novelty on the other side is a randomized restart mechanism inspired by SAT solvers, where if it finds itself stuck for a long time in a subsumption test, it restarts subsumption with a different variable ordering. This is an interesting idea whose impact we will investigate in the next section.

Finally, Subsumer can deal with arbitrary Prolog clauses whereas both Resumer2 and Django can handle only Datalog clauses (i.e. Prolog clauses with no function symbols).

4. Empirical evaluation

In this section we extensively compare Django, Resumer2 and Subsumer. The goal is to compare running times and memory requirements for the three engines on a very challenging benchmark for θ-subsumption engines. In the sections below when we refer to examples we mean the subsumee clauses and by hypotheses we mean the subsumer clauses. This analogy is due to the direct translation of clauses' roles to an ILP system.

All the datasets, subsumption engines and scripts to replicate these experiments can be found at `http://ilp.doc.ic.ac.uk/Subsumer`.

4.1. Datasets

The datasets selected to compare the subsumption engines are instances of the Phase Transition (PT) problem [Gio00]. This artificial problem was originally developed to be a challenge for relational learners like ILP systems. In an ILP system the task is to induce a theory (i.e. target concept) that, together with provided background knowledge, entails a set of positive examples (of the target concept) but no negative examples.

The PT problem is a collection of noise free datasets of varying difficulty each characterized by two parameters, the target concept size, $M \in [5..30]$, and the distinct number of terms, $L \in [12..38]$, present in a subsumee clause. Furthermore each instance is highly non-determinate with 100 solutions per distinct predicate symbol/arity. For each instance there are 200 positive and 200 negative examples evenly divided between train and test and there exists at least one single clause (the target concept) that perfectly discriminates between the positive and negative examples (i.e. has 100% accuracy).

The instances belong to three major regions: Yes, No and Phase Transition. In the Yes region the probability that a randomly generated clause will cover an arbitrary example is close to 1, in the No region is close to 0 and in the narrow Phase Transition (PT) region the probability drops abruptly from 1 to 0.

We selected 43 datasets from the set of 702 possible PT instances ($range(M) * range(L) = 26 * 27 = 702$) as they are good representatives of the three regions. 12 instances are from the Yes region, 15 from the No region and 16 from the PT region. These are the same instances that were used in [Bot03] but there to highlight the difficulty of learning concepts from the PT and No regions for a relational learning system.

4.2. Subsumees/Examples

Each example is a single (saturated) clause with all facts known to be true about it.

All the 400 examples per dataset instance were used. From the subsumption engine perspective all examples are equal, there is no distinction between positive or negative examples. However, since our hypotheses are biased to cover positive examples, it is a better challenge if subsumee clauses that are less likely to be covered are also included.

Due to the nature of the PT dataset all the examples for a particular instance have the same size (i.e. number of literals) and the number of distinct predicate symbols is equal to the concept size, M. The number of distinct terms in an example is L. The arity of all predicate symbols is three with the first argument being always the term from the head. All terms in the examples are constants with no function symbols.

Below is a small excerpt of an example for dataset id=3 (m=18,l=16). The full example has 801 literals.

$$p(d0) \leftarrow br0(d0, d0\_9, d0\_5), br0(d0, d0\_9, d0\_3), br0(d0, d0\_9, d0\_2), \ldots,$$
$$br3(d0, d0\_0, d0\_11), br3(d0, d0\_0, d0\_1), br4(d0, d0\_9, d0\_6), \ldots,$$
$$br7(d0, d0\_0, d0\_3), br7(d0, d0\_0, d0\_15), br7(d0, d0\_0, d0\_13).$$

The examples length range from 501 literals (m=5, l=15) to 2921 literals (m=29, l=24). These instances are from the Yes and No region respectively.

4.3. Subsumers/Hypotheses

The clauses used as subsumers (i.e. hypotheses) were generated using the concept of assymmetric relative minimal generalizations (ARMG) [Mug09]. Essentially the ARMG algorithm receives a clause C and an example e as input and returns a reduced clause R_c, where all literals from C responsible for not entailing e are pruned.

The hypotheses generation algorithm employed receives a list of positive examples and computes the iterative ARMG of all of them. The iterative ARMG of a list of examples is found by computing the (variablized) bottom clause for the first example and then, using it as the start clause, iteratively apply the ARMG algorithm to the remaining examples.

The more examples used to construct an ARMG the smaller (and more general) it will be. Furthermore an ARMG will at least entail all the examples used in its construction.

In order to create the ARMGs we used 10 randomly selected lists of 6, 7, 8, 9 and 10 positive only examples [2] , yielding 50 varying length hypotheses (10 hypotheses are ARMGs with 6 positive examples, ..., 10 hypotheses are ARMGs with 10 positives).

Below is a small excerpt of an hypothesis, an ARMG of 6 positive examples, for dataset id=3 (m=8,l=16). The full hypothesis has 59 literals.

$$p(A) \leftarrow br0(A, B, C), br0(A, B, D), br0(A, E, F), br0(A, E, G), br0(A, E, H), \ldots,$$
$$br1(A, E, O), br1(A, E, N), br1(A, E, L), br1(A, E, J), br2(A, J, K), \ldots,$$
$$br4(A, H, Q), br4(A, D, F), br4(A, D, C), br5(A, I, N), br6(A, J, Q).$$

[2]We did not want to mix positive and negative examples in the ARMG. The reason is that we know this dataset is noise free and since an ARMG, by construction, covers the examples used to create it, the resulting clauses would be shorter and thus less difficult to test for subsumption.

Note that, since our hypotheses are not random -they are biased towards covering positive examples- in the Yes, No and Phase transition regions the probabilities for subsumption are not necessarily close to 1, 0 and 0.5. Nevertheless, it is still relevant to divide the dataset in these three regions as the subsumption tests have a region related difficulty (e.g. longer clauses with more terms in examples and variables in hypotheses).

The hypotheses length vary significantly within each instance (e.g. from 59 to 121 literals for m=17,l=14) but the extremes are 29 literals (m=14, l=24) and 626 literals (m=26, l=12). These instances are both from the PT region. The length of the examples and hypotheses is just a rough indication of the subsumption problem difficulty. Other important factors are: ratio between those lengths, distinct terms in the examples, distinct variables in the hypotheses, distinct predicate symbols.

4.4. Subsumption engines

We used Subsumer, Django [Mal04] and Resumer2 [Kuz08]. Older subsumption engines based on determinate matching [Kie94] and maximal clique search [Sch96] were not tested as we could no longer find them publicly available. However, in [Mal04] they were tested against Django and it clearly outperformed those older engines by several orders of magnitude (speedups between 150x to 1200x).

As for Resumer2, we will also test a variant, which we name Resumer1, that has randomized restarts turned off. This experiment is interesting because it directly tests the importance of randomized restarts in this benchmark. Furthermore, by comparing the relative performance of Resumer1 to Resumer2, we can roughly estimate the gains we would obtain if we were to implement randomized restarts in top of Subsumer.

We compiled Django with gcc 4.1.2, Resumer2 (and Resumer1) with Sun's Java 1.6 and Subsumer with YAP6 Prolog [dS06], all with full optimizations enabled. All experiments were performed in a Athlon Opteron processor 1222 running at 3.0 GHz with 4 GB RAM and a 64 bit build of Linux.

4.5. Results and discussion

A first point to mention is that the four subsumption engines returned the same list of subsumed examples for each instance. This was expected as otherwise there would be at least one faulty implementation. Nevertheless, this is strong evidence that all engines correctly implement θ-subsumption. Notice that each instance consists of 50 (hypotheses) * 400 (examples) = 20.000 subsumption tests.

Analyzing Table 1[3] the first conclusion is that Django consumes too much memory. It consumes so much memory that in only 14 of the 43 datasets it did not crash for exceeding the 4Gb memory limit. It could not solve a single dataset from the No region, the most difficult one. Also, from a CPU time perspective, Django is clearly behind Resumer1/2 and Subsumer by up to 2 orders of magnitude for the few datasets it managed to finish.

The interesting comparison is between Resumer1/2 and Subsumer. Resumer1 is faster than Subsumer but the difference is merely, on average, 5%. Also relevant, standard deviation in Subsumer's running times are about half of Resumer1's. More importantly, Subsumer's memory requirements are only a small fraction (1/8 to 1/10) of either Resumer.

[3]Note that the PT and Overall columns favor Django as, naturally, we can just take into account the datasets where Django successfully finished.

Table 1: Average CPU times (seconds) and memory (megabytes) for problems in each region of the Phase Transition dataset

	Phase Transition dataset region							
	Yes		No		PT		Overall	
Engine	CPU	RAM	CPU	RAM	CPU	RAM	CPU	RAM
Django	4,404	2,248	N/A	N/A	78,736	3,037	15,023	2,361
Resumer1	99	608	544	1,167	225	749	301	855
Resumer2	75	578	154	1,136	120	875	119	883
Subsumer	190	75	442	141	292	92	316	105

Resumer2 is clearly best on all regions. It is followed by Resumer1 though Subsumer manages to outperform Resumer1 in the No region by 23%. Notice that randomized restarts are particularly helpful in this region and are solely responsible for the almost 4 times speedup that Resumer2 has over Resumer1. In the easiest Yes region, Subsumer is about 2 times slower than either Resumer and randomized restarts have almost no impact. In the PT region Resumer1 outperforms Subsumer by 30% and Resumer2 outperforms both by about 2 times showing that, again, randomized restarts are important. Randomized restarts are more helpful as the difficulty of the subsumption test increases. This is as expected as, for simple instances, randomized restarts do not have time to occur.

Overall, Resumer2 clearly outperforms Resumer1 being, on average, 2.5 times faster than it. Also the standard deviation for a subsumption test in Resumer2 decreased considerably comparing with Resumer1. Notably, this is achieved without increasing the memory footprint. This result is further evidence to Resumer2's authors claim in [Kuz08] that randomized restarts are helpful to reduce expected subsumption time.

We did a further experiment to test to which extent dynamic clause decomposition is important to Subsumer. We disabled it and analyzed how Subsumer performed using only the cut transformation. Although for the Yes and PT regions dynamic clause decomposition turned out to be mainly overhead (10%-25% slower), for all the problems in the No region it proved essential. Without it Subsumer would get stuck. It is in the No region that Subsumer outperforms Resumer1 and this is due to dynamic clause decomposition.

To test the importance of the particular compilers used, in a separate experiment we compiled Resumer1 with GNU Java compiler (gcj 4.3.3) also with full optimizations enabled. This gcj version of Resumer1 took 2.5 times longer and required 25% more memory than Resumer1 compiled with Sun's JVM. Subsumer compiled with SWI-Prolog (5.6.59) takes 5.5 times longer than with YAP6. Compilers significantly influence running times.

5. Conclusions and future directions

Our subsumption engine comparison on the challenging PT problem showed that Subsumer clearly outperformed Django both in time and memory and that it is competitive with Resumer2 without randomized restarts. Furthermore, Subsumer requires only $\approx 1/8$ of Resumer2's memory and can handle arbitrary Prolog clauses. We also confirmed the importance of randomized restarts as previously pointed out in [Kuz08]. This is incentive to implement a randomized restart strategy in a future version of Subsumer.

Also worth investigating is the impact of our θ-subsumption engine embedded in a Prolog based ILP system. Could ILP systems then tackle problems they cannot now?

Besides the ILP community, we expect that other related research communities, e.g automated theorem proving, can also profit from our Prolog subsumption engine.

From a strict performance perspective, there would be gains in relaxing Subsumer's auto-imposed constraint of having no state. Namely, often hypotheses are related and have many identical literals, much of the datastructures could be computed once and, at the expense of some memory, running times could be significantly improved.

As for the θ-subsumption problem itself, it is worth verifying if it could be entirely mapped to a constraint satisfaction problem or a sub-graph isomorphism matching problem. If so, one can then use existing state-of-the-art solvers for those problems and check whether they are any better than custom engines like Resumer2 or Subsumer.

Acknowledgments

We thank Ondrej Kuzelka and Filip Zelezný for kindly providing Resumer2 and Django and, specially, for fruitful discussions that improved both Resumer2 and Subsumer. 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 also indebted to three anonymous referees for valuable comments.

References

[Bot03] 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.

[Cos03] 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.

[dS06] Anderson Faustino da Silva and Vítor Santos Costa. The design and implementation of the YAP compiler: An optimizing compiler for logic programming languages. In Sandro Etalle and Miroslaw Truszczynski (eds.), *ICLP*, *LNCS*, vol. 4079, pp. 461–462. Springer, 2006.

[Gio00] Attilio Giordana and Lorenza Saitta. Phase transitions in relational learning. *Machine Learning*, 41(2):217–251, 2000.

[Kap86] Deepak Kapur and Paliath Narendran. NP-completeness of the set unification and matching problems. In Jörg H. Siekmann (ed.), *CADE*, *LNCS*, vol. 230, pp. 489–495. Springer, 1986.

[Kie94] Jrg-Uwe Kietz and Marcus Lbbe. An efficient subsumption algorithm for Inductive Logic Programming. *Proc. 11th Int. Conf. on Machine Learning*, pp. 130–138. Morgan Kaufmann, 1994.

[Kow71] Robert A. Kowalski and Donald Kuehner. Linear resolution with selection function. *Artif. Intell.*, 2(3/4):227–260, 1971.

[Kuz08] Ondrej Kuzelka and Filip Zelezný. A restarted strategy for efficient subsumption testing. *Fundam. Inform.*, 89(1):95–109, 2008.

[Mal04] Jérôme Maloberti and Michèle Sebag. Fast theta-subsumption with constraint satisfaction algorithms. *Machine Learning*, 55(2):137–174, 2004.

[Mug09] Stephen Muggleton, Jose Santos, and Alireza Tamaddoni-Nezhad. Progolem: a system based on relative minimal generalisation. In Luc De Raedt (ed.), *Proceedings of the 19th International Conference on ILP*, *LNCS*, vol. 5989, pp. 131–148. Springer, 2009.

[Rob65] John Alan Robinson. A machine-oriented logic based on the resolution principle. *J. ACM*, 12(1):23–41, 1965.

[Sch96] Tobias Scheffer, Ralf Herbrich, and Fritz Wysotzki. Efficient theta-subsumption based on graph algorithms. In S. Muggleton (ed.), *ILP workshop*, *LNCS*, vol. 1314, pp. 212–228. Springer, 1996.

[Sri07] Ashwin Srinivasan. *The Aleph Manual*. University of Oxford, 2007.