Subsumer: A Prolog $\theta$-subsumption engine

José Santos and Stephen Muggleton
Department of Computing, Imperial College London
(e-mail: \{jcs06,shm\}@doc.ic.ac.uk)

submitted 1 January 2003; revised 1 January 2003; accepted 1 January 2003

Abstract

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

$\theta$-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 $\theta$-subsumption engine.

We present a relatively simple (≈ 1000 lines in Prolog) but efficient and general $\theta$-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, cpu-time 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.

KEYWORDS: Theta-subsumption engine, Inductive Logic Programming, Constraint Logic Programming

1 Introduction and motivation

$\theta$-subsumption is an important problem in computational logic and particularly relevant to the Inductive Logic Programming (ILP) (Muggleton and Raedt 1994) and Theorem Proving communities. $\theta$-subsumption is at the core of the coverage test (i.e. determine which examples a putative hypotheses entails) in an ILP system.

Current state-of-the-art ILP systems are usually developed in Prolog, e.g. Aleph (Srinivasan 2007) and ProGolem (Muggleton et al. 2009), 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.

ILP algorithms construct hypotheses from a very rich hypotheses language and thus have to traverse a large search space. This search requires having to test some metric of the candidate clauses. The metric typically used is compression: positive example coverage minus negative coverage minus clause length. Evaluating compression of a single candidate clause requires thus many subsumption tests, each one of those being potentially very expensive as they are 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 (Blockeel et al. 2002) to take advantage of the similar structure of the candidate clauses, transform the clause before execution (Costa et al. 2003) so that the transformed clause is more efficient to evaluate, improve the indexing mechanism (Costa et al. 2007) of the Prolog engine, to stochastic estimation of the clause coverage (Sebag and Rouveirol 1997).

The subsumption problem at the culprit of the ILP bottleneck received far less attention probably because, for many ILP applications, Prolog’s built-in resolution seemed to suffice. However, due to the non-determinism explosion highlighted above, ILP researchers often 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 (Maloberti and Sebag 2004) and Resumer2 (Kuzelka and Zelezny 2008), 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. In this paper we will show not only how that objective, and in particular low memory footprint, was achieved but also how its runtime performance is superior to Django and competitive with Resumer2.

The rest of the paper is organized as follows. In the next section we introduce the $\theta$-subsumption problem. In section 3 we present Subsumer’s algorithms and overview how those compare with the equivalents in Django and Resumer2. The empirical results comparing Subsumer with Django and Resumer2 are presented in section 4. Section 5 concludes and suggests directions for future research.
2 The $\theta$-subsumption problem

$\theta$-subsumption (Robinson 1965) is an approximation to logical implication. While implication is undecidable in general $\theta$-subsumption is a NP-complete problem (Kapur and Narendran 1986). A clause $C\theta$-subsumes a clause $D$ ($C \vdash_\theta D$) if and only if there exists a substitution $\theta$ such that $C\theta \subseteq D$.

Example 1 ($\theta$-subsumption)

$C : h(X_0) \leftarrow l_1(X_0, X_1), l_1(X_0, X_2), l_2(X_1, X_2), l_2(X_1, X_3)$

$D : h(c_0) \leftarrow l_1(c_0, c_1), l_1(c_0, c_2), l_2(c_1, c_2)$

$C\theta$ subsumes $D$ with $\theta = \{X_0/c_0, X_1/c_1, X_2/c_2, X_3/c_2\}$.

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

The standard algorithm for $\theta$-subsumption is based on Prolog’s SLD-resolution (Kowalski and Kuehner 1971). 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 SLD-resolution (in)efficiency.

2.1 $\theta$-subsumption time complexity

Let $N$ and $M$ be the lengths of clauses $C$ and $D$. The standard $\theta$-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 $D$. We can map the $\theta$-subsumption problem to the problem of finding 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.

3 Subsumer: A Prolog $\theta$-subsumption engine

Subsumer is a publicly available (http://ilp.doc.ic.ac.uk/Subsumer), simple ($\approx 1000$ lines of Prolog) fully general $\theta$-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 (see Section 3.3) is the most expensive part of the algorithm. 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
Subsumer: A Prolog $\theta$-subsumption engine

3.2 Datastructures

The subsumer clause, $C = h \leftarrow b_1, \ldots, b_n$ 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. $\text{hl}(X_0), X_1$ and $\text{hl}(X_1, X_2)$ have the same calling pattern but $\text{hl}(X_0, X_1)$ and $\text{hl}(X_0, X_0)$ are distinct.

The subsumee clause, $D = e \leftarrow g_1, \ldots, g_n$ 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 $\text{Val}(p_s/a)$, that is the predicate symbol domain. For instance, we would compactly represent clause $D$ in Example 1, as \{1/2 : [(c_0, c_1), (c_0, c_2)], 2/2 : [(c_0, c_2)]\}.

The space needed to store clause’s $D$ related information is thus: $O(\sum_1^N \text{Val}(p_s/a))$ 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 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 1: $X_0 : \{X_1, X_2, X_3\}, X_1 : \{X_0, X_2, X_3\}, X_2 : \{X_0, X_1\}, X_3 : \{X_0, X_3\}$.

Let $V$ denote the set of distinct variables in $C$ and $\bar{V}$ denote the average number of variable interactions. Then, this requires $O(|V|^{|V|})$ space which in the worst case is $O(|V|^{|V|})$.

We also have a datastructure that, for each variable, holds the indexes of the literals where the variable occurs in the clause (clause’s head being index 1). For the same clause $C$ from Example 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 Variable domain update

For each literal $l$ in a clause $C$, we have a sophisticated indexing and back-indexing structure, to allow for efficient assignment of a value to a variable and respective propagation of its new value to its direct interacting variables.

---

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).
In this literal index structure we keep, for each variable $v$ in literal $l$’s arguments, and for each value in the initial domain of $v$, the list of positions where that value occurs (due to $v$) in $l$’s arguments.

For instance, suppose we have a literal $l = l0(V_A, V_B, V_C)$ which has domain $\{\{a_1, b_2, c_1\}, \{a_1, b_3, c_1\}, \{a_1, b_1, c_2\}\}$ and that the initial domains for the individual variables are: $V_A \in \{a_1\}$, $V_B \in \{b_1, b_2, b_3\}$, $V_C \in \{c_1, c_2\}$. Our literal index is then a list of hashtables (one per variable) where the keys are the variable values and the hashtable values are the $l$’s domain index where the key occurs. For the example above the hashtables would be $V_A = \langle a1 - [1, 2, 3]\rangle$, $V_B = \langle b1 - [3], b2 - [1], b3 - [2]\rangle$, $V_C = \langle c1 - [1, 2], c2 - [3]\rangle$.

All these datastructures are central to efficiently update the domain of interacting variables, the most expensive operation in Subsumer’s algorithm. Suppose we now do $V_C = c_2$. What must be done to propagate this assignment? We would first check with which free variables $V_C$ interacts. Then, for each of those variables find which literals $V_C$ shares in common with them. All this can be done in time proportional to the number of variables $V_C$ interacts with.

Suppose we had already discover that $V_A$ and $V_B$ interact with $V_C$ in literal $l$. We then check in which indexes of $l$’s domain does $V_C = c_2$ occur (in this example index $= \{3\}$ only) and can then, in time proportional to $\text{min}(N, K)$, update (i.e. restrict) the domain of each neighbouring variable $V_i$ (here $V_i \in \{V_A, V_B\}$), where $N$ is the number of occurrences of the particular $V_C$ value (here $c_2$) in $l$’s domain and $K$ is the current domain size of the neighbouring variable $V_i$ ($V_A$’s domain size is 1 and $V_B$’s is 3).

This update is done by iterating over the possible indexes for $V_C$, collecting the distinct values for each of the interacting variables and finally intersecting these values with the current variables domain.

In this example we would find $b_1$ as the only possible value for $V_B$’s new domain and would then intersect it with $\{b_1, b_2, b_3\}$ to yield $V_B = \{b_1\}$. Note that in this simple example it could look like that the last intersection was avoidable but in general it is not as we have to iterate over all the literals shared by $V_C$ and $V_B$ and have to keep the running intersection. If at any point the intersection becomes empty we then know that the assignment for $V_C$ was inconsistent.

### 3.4 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 $\prod_{i}^{N} k_i$ 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 $\sum_{i}^{N} O(b^{N_i})$, which is $O(b^{\text{max}\{N_i\}})$. That is, the worst case number of predicate calls clause is no longer exponential in the length of the clause but exponential in the size of the longest group.
The reasoning is then applied recursively to the newly found subcomponents. This idea, named once-transformation, was initially presented in (Costa et al. 2003). 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.

In our algorithm the subsumer clause is mapped to an undirected graph where nodes are variables. There exists an arc between two variables iff they share at least one literal. The independent components (groups of literals) of a clause are the disconnected components of this underlying graph. This leads to Definition 1.

**Definition 1 (Independent clause components)**

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 2 (Clause decomposition)**

\[ h(X) \leftarrow a(X, Y), b(X, Z), c(Y, A), d(Y, B), e(Z, C), f(Z, D) \]

In this example all variables are connected and thus the whole clause is a single component. However, when variable \( X \) (the head variable) 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. Resumer2 does this level of decomposition only (called the cut-transformation in (Costa et al. 2003)). It decomposes a clause when the head variable becomes bound (i.e. at the beginning of the subsumption test). 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.
There are only two other subsumption engines comparable with Subsumer in terms of the complexity of clauses they can handle: Django (Maloberti and Sebag 2004) and Resumer2 (Kuzelka and Zelezný 2008). Both are complex engines, around 10,000 lines of source code each, implemented in relatively low-level languages (compared to Prolog): C and Java respectively.

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 requires particularly large quantities of memory because it performs determinate matching between the literals in the subsumer clause and the literals in the subsumee prior of starting its normal non-determinate matching. The determinate matching is an idea originally presented in (Kietz and Lbbe 1994), 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. This has the potential of significantly speeding up the subsumption but is only effective if indeed those unique signatures exist. These signatures are expensive to compute and, specially, store. Furthermore, to be discriminative they look for second level neighbours which has a even bigger cost. Resumer2 only tries determinate matching with first level neighbours which has a much lower memory cost. Subsumer does not implement any form of determinate matching.

Because of these signatures both Django and Resumer2 take a significant amount of time to initialize their datastructures, 20 to 30 times more than Subsumer. If we only need to do a single subsumption test, one subsumer clause against one subsumee clause, Subsumer is highly likely to finish before Django or Resumer2 are ready for the test. However, in a realistic application we will have many hypotheses to be tested against many examples and it can pay off to incur those heavy initialization costs.

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). In a Prolog implementation dealing with function symbols comes relatively naturally (although we could have optimized further our algorithm had we...
Subsumer: A Prolog \(\theta\)-subsumption engine

opted not to support them) but in C or Java it is a significant extra burden to support.

4 Empirical evaluation

In this section we will extensively compare Django, Resumer2 and Subsumer. The goal of these experiments is to compare running times and memory requirements for the three engines on a very challenging benchmark for \(\theta\)-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 (Giordana and Saitta 2000). This artificial problem was originally developed to be a challenge for relational learners like Inductive Logic Programming systems.

In an ILP system the challenge is to induce a theory (the target concept) that together with the existing background knowledge covers (i.e. entails) the given positive examples and does not cover the 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 \((\text{range}(M) \times \text{range}(L) = 26 \times 27 = 702)\) 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 (Botta et al. 2003) but there to highlight the difficulty of learning concepts from the PT and No regions for a relational learning system.

This dataset is challenging to a relational learner mainly due to the high non-determinacy and large concept sizes for ILP standards (typically those are between 3 to 6 literals). Evaluating the coverage of a large non-determinate clause is prohibitively expensive with the traditional built-in, left-to-right, depth-first search
implementation of SLD-resolution in Prolog compilers, which is what most ILP systems use.

4.2 Subsumees/Examples

Each example is a single (saturated) clause with all facts known to be true about it. They were generated by running an ILP system and retrieving their fully ground bottom clause. Note that ILP’s bottom clause is variablized, e.g. $h(X) ← p(X, Y)$. We had to update the ILP system’s bottom clause generation algorithm so that the actual terms beneath the variables were displayed, e.g. $h(a) ← p(a, b)$.

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.

\[
p(d0) ← br0(d0, d0_9, d0_5), br0(d0, d0_9, d0_3), br0(d0, d0_9, d0_2), \ldots , \]
\[
br3(d0, d0_0, d0_1), br3(d0, d0_0, d0_1), br4(d0, d0_9, d0_6), \ldots , \]
\[
br7(d0, d0_0, d0_3), br7(d0, d0_0, d0_15), br7(d0, d0_0, d0_13).
\]

4.3 Subsumers/Hypotheses

The clauses used as subsumers (i.e. hypotheses) were generated using the concept of assymmetric relative minimal generalizations (ARMG) (Muggleton et al. 2009), implemented recently in the bottom-up ILP system ProGolem. 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 for this experiment, 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.

Naturally, the more examples used to construct an ARMG the smaller (and thus more general) it will be. Furthermore, by construction, an ARMG will at least entail
Subsumer: A Prolog \( \theta \)-subsumption engine

all the examples used in its construction. But note that the end result of an ARMG is still a relatively large and specific clause.

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

We could have used ARMGs of negative examples as well and the results would be identical but we did not want to mix positive and negative examples in the ARMG. The reason is that we know that this dataset is noise free and since an ARMG covers the examples used in its construction, the resulting ARMG would likely be much shorter (more literals would have need to be removed) and thus pose a less interesting challenge.

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.

\[
p(A) \leftarrow \text{br}0(A, B, C), \text{br}0(A, B, D), \text{br}0(A, E, F), \text{br}0(A, E, G), \text{br}0(A, E, H), \ldots, \\
\text{br}1(A, E, O), \text{br}1(A, E, N), \text{br}1(A, E, L), \text{br}1(A, E, J), \text{br}2(A, J, K), \ldots, \\
\text{br}4(A, H, Q), \text{br}4(A, D, F), \text{br}4(A, D, C), \text{br}5(A, I, N), \text{br}6(A, J, Q).
\]

The number of variables in any hypothesis is always equal to the number of distinct terms (\(L\)) in the examples. This is because the secret target concept, created by the authors of the PT problem, has that property and, due to the way ARMGs are constructed the property is shared with all the hypotheses. Note that, at each stage during ARMG construction, only the literals that make the clause not entail a positive example are removed.

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.

4.4 Subsumption engines

Subsumer, implemented in YAP Prolog (da Silva and Costa 2006), was compared with Django (Maloberti and Sebag 2004) implemented in C and Resumer2 (Kuzelka and Zelezný 2008) implemented in Java. These are relatively recent subsumption engines, older subsumption engines based on determinate matching (Kietz and Lbbe 1994) and maximal clique search (Scheffer et al. 1996) were not tested because we could no longer find them publicly available. However in (Maloberti and Sebag 2004) they were tested against Django and it clearly outperformed those older engines by several orders of magnitude (speedups between 150x to 1200x) in this same dataset.

In that paper Django was tested with randomly generated hypotheses with lengths varying from 10 to 50 literals but always 10 variables. Our ARMG based hypotheses range from 29 to 626 literals (see Table 2) and the number of variables range from 12 to 31 (\(L\)'s range), which should pose a much bigger challenge to the subsumption engine.
However, notice that it is not easy to come up with a simple reliable measure of how difficult a given dataset will be. It depends on many factors: examples length, hypotheses length, average ratio between the latter two, distinct terms in the examples, distinct variables in the hypotheses, distinct predicate symbols, . . .

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, 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

In Table 2 the $|Ex|$ column has the number of literals of each saturated example for a given $M$ and $L$ instance of the PT dataset. Please remember that, due to the way the PT dataset was constructed, all examples for a given instance have the same length. The $|Hyp|$ column has the range of the number of literals of the hypotheses used for subsumption testing.

The CPU time column represents the total time, in seconds, taken by the subsumption engine to do the 50 (hypotheses) * 400 (examples) = 20,000 subsumption tests per particular instance of $M$ and $L$. Likewise, the RAM column represents the memory, in megabytes, required by the subsumption engine to perform those tests.

A first point that is important to mention is that the four subsumption engines returned the same list of subsumed examples for each instance. This was obviously expected as otherwise there would be at least one faulty implementation. Nevertheless, this is a very strong indication that all algorithms correctly implement $\theta$-subsumption.

Inspecting Table 2 results, the first conclusion is that Django consumes too much memory. It consumes so much memory to the extent 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 challenging one. This is partly because the Django engine has memory leaks. After each subsumption test there is a small increment in the memory footprint that is never reclaimed. 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, Resumer2 and Subsumer. Although Resumer1 is still faster than Subsumer, the difference is merely, on average, 5% which can be partially attributed to the underlying programming languages. Standard deviation in Subsumer running times are about half of Resumer1’s which
is positive. More importantly, Subsumer’s memory requirements are only a small fraction (1/8 to 1/10) of either Resumer.

Table 1\(^2\) summarizes the results of the four subsumption engines in the three regions of the PT dataset. Resumer2 is clearly best on all regions. It is followed by Resumer1 though Subsumer manages to outperform Resumer1 in the hardest 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 on average 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 helpful. Randomized restarts are more helpful as the difficulty of the subsumption test increases. This is as expected as, for simple instances randomized restarts either do not have the time to occur or, if they do, are likely to be overhead.

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 (Kuzelka and Zelezný 2008) that randomized restarts are helpful to reduce expected subsumption time.

However, it is relevant to point out that the version of Resumer2 (and thus 1) used in these experiments incorporates improvements not available in the original implementation (Kuzelka and Zelezný 2008). Namely this version consumes 55% less memory and is 20% faster. These improvements were made during Subsumer’s development as a result of fruitful discussions with Resumer2’s authors. In absolute terms, the memory requirements of Resumer are still very high, though. Notice that the reported results are for a 64 bit operation system. In similar experiments in a 32 bit OS the memory required by all subsumption engines is about half.

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 without it (i.e. using only the cut transformation). Although for the Yes and PT regions the dynamic clause evaluation turned out to be mainly overhead (10%-25% slower), for all the problems in the No region it proved essential. Without it Subsumer would still be working after a few hours. This raises the important question on how, without it, Resumer1 and Resumer2 still manage to complete them with competitive time. This may be due to determinate matching that Subsumer does not implement but Resumer1/2 does. Notice that those problems, especially 14, 15, 16 and 19 (highlighted in bold in Table 2), are the ones where Subsumer outperforms Resumer1 and that is due to the dynamic clause decomposition.

Naturally the ideas embedded in the engines play the dominating role in the final runtime and memory results but compilers also have a relevant role. In a separate

---

\(^2\) Note that the PT and Overall columns are misleading in favor of Django as, naturally, we can just take into account the datasets where Django successfully finished. If Django could successfully run in those datasets the CPU times and memory would be even higher.
Table 1. Summary of performance comparison between Django, Resumer1, Resumer2 and Subsumer on each region of the Phase Transition dataset. Average CPU times are in seconds, average RAM is in megabytes.

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

experiment we compiled Resumer1 with GNU Java compiler (gcj 4.3.3) also with full optimizations enabled. This version of Resumer1 took 2.5 times longer and required 25% more memory than Resumer1 compiled with Sun’s JVM. We then compiled Subsumer with SWI-Prolog (5.6.59) which took 5.5 times longer than with YAP6. These empirical results shed some light on the impact of a compiler’s generated code on a program’s performance.

5 Conclusions and future directions

We have systematically compared the performance of Subsumer, Resumer1, Resumer2 and Django subsumption engines on a very challenging θ-subsumption benchmark. Subsumer clearly outperformed Django both in time and memory, showing that it is possible to do efficient θ-subsumption in Prolog. Resumer1 (i.e. Resumer2 with no randomized rapid restarts) is, overall, only 5% faster but requires about 8 times more memory than Subsumer which may be prohibitive in certain scenarios, e.g. when integrated within an ILP system, where the ILP system itself already requires considerable memory. Furthermore, nor Django nor Resumer can handle function symbols in clauses.

Resumer2 has the same memory requirements as Resumer1 but is about 2.5 times faster. Randomized restarts pay off as shown in (Kuzelka and Zelezný 2008). This should be enough incentive to implement a randomized restart strategy in a future version of Subsumer as it is likely identical performance gains will be achieved.

An important point worth investigating is the impact of our θ-subsumption engine embedded in a Prolog based ILP system such as ProGolem (Muggleton et al. 2009). Namely, could ILP systems then tackle problems they cannot now? Efficient θ-subsumption is a necessary condition (but probably insufficient by itself) to solve the Phase Transition dataset challenge posed a decade ago in (Giordana and Saitta 2000).

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 Sub-
Table 2. Performance comparison between Django, Resumer1, Resumer2 and Subsumer on the Phase Transition dataset. CPU times are in seconds, RAM is in Megabytes.

<table>
<thead>
<tr>
<th>Region</th>
<th>Dataset</th>
<th>Django</th>
<th>Resumer1</th>
<th>Resumer2</th>
<th>Subsumer</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td>CPU</td>
<td>RAM</td>
<td>CPU</td>
<td>RAM</td>
</tr>
<tr>
<td>Id</td>
<td>M</td>
<td>L</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Y</td>
<td>0</td>
<td>5</td>
<td>15</td>
<td></td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>6</td>
<td>20</td>
<td>601</td>
<td>34-54</td>
<td>0</td>
</tr>
<tr>
<td>n</td>
<td>7</td>
<td>19</td>
<td>30-56</td>
<td>2067</td>
<td>2597</td>
</tr>
<tr>
<td>s</td>
<td>8</td>
<td>16</td>
<td>801</td>
<td>45-79</td>
<td>3040</td>
</tr>
<tr>
<td>r</td>
<td>9</td>
<td>15</td>
<td>901</td>
<td>49-84</td>
<td>3594</td>
</tr>
<tr>
<td>e</td>
<td>10</td>
<td>16</td>
<td>1001</td>
<td>43-75</td>
<td>3677</td>
</tr>
<tr>
<td>g</td>
<td>11</td>
<td>13</td>
<td>1101</td>
<td>78-152</td>
<td>5800</td>
</tr>
<tr>
<td>i</td>
<td>12</td>
<td>13</td>
<td>1201</td>
<td>74-173</td>
<td>6261</td>
</tr>
<tr>
<td>o</td>
<td>13</td>
<td>13</td>
<td>1301</td>
<td>77-166</td>
<td>7110</td>
</tr>
<tr>
<td>n</td>
<td>14</td>
<td>12</td>
<td>1401</td>
<td>118-364</td>
<td>9287</td>
</tr>
<tr>
<td>P</td>
<td>15</td>
<td>33</td>
<td>1310</td>
<td>34-53</td>
<td>N/A</td>
</tr>
<tr>
<td>N</td>
<td>16</td>
<td>31</td>
<td>1513</td>
<td>35-48</td>
<td>N/A</td>
</tr>
<tr>
<td>o</td>
<td>17</td>
<td>35</td>
<td>1513</td>
<td>35-49</td>
<td>N/A</td>
</tr>
<tr>
<td>r</td>
<td>18</td>
<td>31</td>
<td>1515</td>
<td>35-50</td>
<td>N/A</td>
</tr>
<tr>
<td>g</td>
<td>19</td>
<td>36</td>
<td>1616</td>
<td>37-51</td>
<td>N/A</td>
</tr>
<tr>
<td>i</td>
<td>20</td>
<td>18</td>
<td>1815</td>
<td>34-52</td>
<td>N/A</td>
</tr>
<tr>
<td>o</td>
<td>21</td>
<td>18</td>
<td>1818</td>
<td>37-47</td>
<td>N/A</td>
</tr>
<tr>
<td>n</td>
<td>22</td>
<td>19</td>
<td>2187</td>
<td>34-50</td>
<td>N/A</td>
</tr>
<tr>
<td>P</td>
<td>23</td>
<td>16</td>
<td>2417</td>
<td>41-61</td>
<td>N/A</td>
</tr>
<tr>
<td>N</td>
<td>24</td>
<td>18</td>
<td>2520</td>
<td>38-52</td>
<td>N/A</td>
</tr>
<tr>
<td>o</td>
<td>25</td>
<td>18</td>
<td>2719</td>
<td>42-69</td>
<td>N/A</td>
</tr>
<tr>
<td>n</td>
<td>26</td>
<td>19</td>
<td>2923</td>
<td>43-73</td>
<td>N/A</td>
</tr>
<tr>
<td>P</td>
<td>27</td>
<td>19</td>
<td>2925</td>
<td>36-61</td>
<td>N/A</td>
</tr>
<tr>
<td>N</td>
<td>28</td>
<td>20</td>
<td>2921</td>
<td>35-54</td>
<td>N/A</td>
</tr>
<tr>
<td>T</td>
<td>29</td>
<td>26</td>
<td>601</td>
<td>36-49</td>
<td>N/A</td>
</tr>
<tr>
<td>T</td>
<td>30</td>
<td>28</td>
<td>601</td>
<td>36-49</td>
<td>N/A</td>
</tr>
<tr>
<td>T</td>
<td>31</td>
<td>27</td>
<td>701</td>
<td>35-50</td>
<td>N/A</td>
</tr>
<tr>
<td>T</td>
<td>32</td>
<td>31</td>
<td>701</td>
<td>35-50</td>
<td>N/A</td>
</tr>
<tr>
<td>T</td>
<td>33</td>
<td>31</td>
<td>801</td>
<td>34-50</td>
<td>N/A</td>
</tr>
<tr>
<td>T</td>
<td>34</td>
<td>31</td>
<td>1109</td>
<td>38-60</td>
<td>N/A</td>
</tr>
<tr>
<td>T</td>
<td>35</td>
<td>32</td>
<td>1111</td>
<td>36-53</td>
<td>N/A</td>
</tr>
<tr>
<td>T</td>
<td>36</td>
<td>32</td>
<td>1310</td>
<td>39-57</td>
<td>N/A</td>
</tr>
<tr>
<td>T</td>
<td>37</td>
<td>36</td>
<td>1311</td>
<td>34-52</td>
<td>N/A</td>
</tr>
<tr>
<td>T</td>
<td>38</td>
<td>36</td>
<td>1401</td>
<td>40-64</td>
<td>N/A</td>
</tr>
<tr>
<td>T</td>
<td>39</td>
<td>27</td>
<td>1414</td>
<td>29-53</td>
<td>N/A</td>
</tr>
<tr>
<td>T</td>
<td>40</td>
<td>27</td>
<td>1701</td>
<td>59-121</td>
<td>N/A</td>
</tr>
<tr>
<td>T</td>
<td>41</td>
<td>27</td>
<td>1701</td>
<td>48-98</td>
<td>N/A</td>
</tr>
<tr>
<td>T</td>
<td>42</td>
<td>26</td>
<td>1914</td>
<td>49-78</td>
<td>N/A</td>
</tr>
<tr>
<td>T</td>
<td>43</td>
<td>26</td>
<td>2601</td>
<td>47-89</td>
<td>N/A</td>
</tr>
<tr>
<td>T</td>
<td>44</td>
<td>26</td>
<td>2601</td>
<td>194-626</td>
<td>N/A</td>
</tr>
</tbody>
</table>

Avg CPU time, Avg RAM: 15023 2361 301 855 119 883 316 105
Std. Dev. CPU time and RAM: 30576 1225 441 314 62 323 170 40
Max CPU time, Max RAM: 114946 4071 2495 1471 333 1521 999 178
sumer’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 \( \theta \)-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 would like to thank Ondrej Kuzelka and Filip Zelezný for kindly providing us Resumer2 and Django and specially for fruitful discussions that helped improving 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


