An Efficient Subsumption Algorithm for Inductive Logic Programming

Jörg–Uwe Kietz and Marcus Lübbe
German National Research Center for Computer Science,
FIT, Schloß Birlinghoven, D–53754 St.Augustin, Germany
kietz@gmd.de, luebbe@gmd.de

Abstract

In this paper we investigate the efficiency of θ–subsumption ($\vdash_\theta$), the basic provability relation in ILP. As $D \vdash_\theta C$ is NP–complete even if we restrict ourselves to linked Horn clauses and fix C to contain only a small constant number of literals, we investigate in several restrictions of D. We first adapt the notion of determinate clauses used in ILP and show that θ–subsumption is decidable in polynomial time if D is determinate with respect to C. Secondly, we adapt the notion of k–local Horn clauses and show that θ–subsumption is efficiently computable for some reasonably small k. We then show how these results can be combined, to give an efficient reasoning procedure for determinate k–local Horn clauses, an ILP–problem recently suggested to be polynomial predictable by Cohen (1993) by a simple counting argument. We finally outline how the θ–reduction algorithm, an essential part of every lgg ILP–learning algorithm, can be improved by these ideas.

1 INTRODUCTION

The field of Inductive Logic Programming (ILP), concerned with the induction of first–order Horn clauses[1] from examples and background knowledge, has recently received increased attention (Quinlan, 1990; Muggleton, 1992; De Raedt, 1992; Lavrač and Džeroski, 1993; Morik et al., 1993). The problem addressed by Inductive Logic Programming is usually described formally as follows:

Definition 1 (ILP Learning Problem)
The ILP problem is a tuple $(\vdash, LB, LE, LH)$*, such that* $\vdash$ *is a correct provability relation for a first–order language* L*, i.e., for all* $A, B \in L$ *: if* $A \vdash B$*, then* $A \models B$*,* $LB \subseteq L$ *is a language for background knowledge,* $LE \subseteq L$ *is a language for examples and* $LH \subseteq L$ *is a language for hypotheses. An algorithm which accepts any* $B \subseteq LB$ *and* $E \subseteq LE$ *as input and computes an* $H \in LH$ *such that*

($B, H, E \not\vdash \square$*), i.e.,* H *is consistent with* B *and* E*,*

($B, H \vdash E^+$*), i.e.,* H *and* B *explain* E^+*, and*

($\forall e \in E^- : B, H \not\vdash e$*), i.e.,* H *and* B *do not explain* E^-

if it exists, or "no" if it does not exist, is called an ILP–learning algorithm *for* $(\vdash, LB, LE, LH)$.

As can be easily seen, the provability relation $\vdash$ plays a central role in the definition of the ILP–problem. In existing ILP systems it is used for several purposes. First of all, $\vdash$ is used to test if a hypothesis covers the examples. Several systems use it to reduce redundancy within the hypothesis space. This includes reducing lgg's (Plotkin, 1970; Muggleton and Feng, 1990), the use of a refinement graph containing only reduced clauses, (van der Laag and Nienhuys-Cheng, 1993; Shapiro, 1983; De Raedt and Bruynooghe, 1993) and redundancy testing between competing hypotheses (Buntine, 1988; De Raedt and Bruynooghe, 1993).

The efficiency of $\vdash$ is therefore crucial for the efficiency of ILP learning algorithms.

2 MAKING PROVABILITY DECIDABLE

A provability relation often used in ILP is θ–subsumption, first defined by Robinson (1965) and first used in the ILP–framework by Plotkin (1970).

Definition 2 *A clause* D *θ–subsumes a clause* C *(*$D \vdash_\theta C$*) iff there exists a substitution* θ *such that* $D\theta \subseteq C$.

θ–subsumption is a correct consequence relation, i.e., if $D \vdash_\theta C$ then $D \models C$. As θ–subsumption between clauses

[1] We assume the reader is familiar with logic programming terminology as described in the textbook by Lloyd (1987).

is decidable, it must be incomplete with respect to implication. The incompleteness of $\vdash_\theta$ can lead to incorrect results of the ILP learning problem, i.e. a negative example would not be realized as being implied by a hypothesis. Therefore, the cases where $\vdash_\theta$ is incomplete must be excluded from the learning problem. Gottlob (1987) has shown that this incompleteness can be precisely characterized. If D is self–resolving or C is tautological, then $D \models C$ is not equivalent to $D \vdash_\theta C$. Hence, tautologies must be excluded from the learning situation. Concerning self–resolving clauses D, self–resolution of D is used to derive a clause D' from D. Then $D \models C$, iff $D' \vdash_\theta C$ (Muggleton, 1993) .

As θ–subsumption is only defined between two clauses, the use of background knowledge is excluded at first sight. To repair this deficiency Plotkin (1971) has defined the notion of subsumption relative to a background theory. Buntine (1988) defined a special case of relative subsumption called generalized subsumption. As this paper is mostly concerned with provability between Horn clauses with the same head, we will use Buntine's definition.

Definition 3 *Let $B \subseteq LB$ be background knowledge and let σ be a Skolem substitution for a Horn clause C. A clause D is more general than a clause C w.r.t. generalized subsumption, $D \vdash_B C$, iff there is a substitution θ, such that $D_{Head}\theta = C_{Head}$ and $B, C_{Body}\sigma \models C_{Body}\theta\sigma$.*

Buntine has shown that θ–subsumption is a special case of generalized subsumption occurring when the background knowledge is empty, i.e., we have an extension of θ–subsumption incorporating background knowledge. However, the background knowledge used in ILP programs must always be restricted, otherwise the ILP problem inherits the undecidability of the deduction process within the background knowledge.

A common restriction in ILP programs (e.g., (Muggleton and Feng, 1990; Quinlan, 1990; Kietz and Wrobel, 1992)) is the restriction to ground background knowledge and ground unit clauses as examples.

Definition 4 *The background knowledge B is ground if it consists of ground unit clauses only. A clause is ground if it does not contain any variables.*

The restriction to ground unit clauses as examples and ground background knowledge enables us to use θ–subsumption as an inference procedure completely incorporating background knowledge in the following way.

The ILP problem ($\vdash_B$, ground B, ground unit E, LH) is equivalent to the ILP problem ($\vdash_\theta$, $\{\}$, E_{new}, LH), where E_{new} is defined as $E^+_{new} := \{e \leftarrow B_1, \ldots, B_n \mid e \in E^+, \{\{B_1\}, \ldots, \{B_n\}\} = B\}$ and $E^-_{new} := \{e \leftarrow B_1, \ldots, B_n \mid e \in E^-, \{\{B_1\}, \ldots, \{B_n\}\} = B\}$.

For most LH used in ILP we only need to consider example clauses which are much smaller, i.e., example clauses which are instantiations of hypotheses. $E_{new} := \{e \leftarrow B_1, \ldots, B_n \mid e \in E, \{\{B_1\}, \ldots, \{B_n\}\} \subseteq B$, and there exists a $H \in LH$ and a substitution σ such that $H\sigma = e \leftarrow B_1, \ldots, B_n\}$.

3 MAKING PROVABILITY EFFICIENT

So far, we have seen how the decidable provability relation $\vdash_\theta$ can be used in ILP. Unfortunately there is most likely no efficient algorithm for $\vdash_\theta$, as $\vdash_\theta$ between linked Horn clauses is NP–hard as the following theorem shows.

Theorem 1 [2] *Let (V, S) be any 3sat instance, where*
$V = \{x_1, \ldots, x_n\}$,
$S = \{\{l_{11}, l_{12}, l_{13}\}, \ldots, \{l_{m1}, l_{m2}, l_{m3}\}\}$,
$\{l_{11}, \ldots, l_{m3}\} \subseteq \{x_1, \ldots, x_n, \overline{x_1}, \ldots, \overline{x_n}\}$.

Let φ be a 1-to-1 mapping between the Boolean literals $\{x_1, \ldots, x_n, \overline{x_1}, \ldots, \overline{x_n}\}$ and $2n$ first–order–logic variables $X_1, \ldots, X_n, NX_1, \ldots, NX_n$. The 3sat instance (V, S) is solvable iff $D \vdash_\theta C$, for the following two linked Horn clauses D and C:

$$
\begin{aligned}
D = h(L) \quad \leftarrow\ & p(L, \varphi(x_1)), \ldots, p(L, \varphi(x_n)),\\
& c(\varphi(l_{11}), \varphi(l_{12}), \varphi(l_{13})), \ldots,\\
& c(\varphi(l_{m1}), \varphi(l_{m2}), \varphi(l_{m3})),\\
& ne(\varphi(x_1), \varphi(\overline{x_1})), \ldots, ne(\varphi(x_n), \varphi(\overline{x_n})).\\
C = h(sat) \leftarrow\ & p(sat, 0), p(sat, 1),\\
& c(1,1,1), c(1,0,1), c(1,1,0), c(1,0,0),\\
& c(0,1,1), c(0,1,0), c(0,0,1),\\
& ne(1,0), ne(0,1).
\end{aligned}
$$

Proof:
$|D|$ is linear in $|V| + |S|$, whereas $|C|$ is of constant size. Note, that C contains the codings of all 7 possibilities to satisfy a 3–literal Boolean clause, which ensures that all the coded 3sat clauses are satisfied, if $D \vdash_\theta C$ and the coding of any satisfiable clause in D is an element of C. Further note, that the ne literals ensure, that $\varphi(x_i)$ and $\varphi(\overline{x_i})$ are not both 0 and not both 1, i.e, are valid truth assignments and that any valid truth assignment makes $ne(\varphi(x_1), \varphi(\overline{x_1}))$ of D an element of C. This proves the claim. □

Example 1 *The 3sat instance $V = \{x1, x2, x3, x4\}$, $S = \{\{x1, \overline{x2}, x4\}, \{\overline{x2}, \overline{x3}, \overline{x4}\}, \{\overline{x1}\}, \{x1, x2\}\}$ is*

[2] As stated in (Garey and Johnson, 1979), the NP–completeness of D $\vdash_\theta$ C between two clauses C and D was originally proven in an unpublished paper by Baxter (1977). As we did not succeed in obtaining this paper, and other published proofs of the NP–completeness of $D \vdash_\theta C$ (Kapur and Narendran, 1986; Cohen, 1993), did not prove the claim in Garey and Johnson that the problem remains NP–hard even for a fixed clause C, we have given a new proof of this old theorem.

coded in the following Horn clause.

$$\begin{aligned} D = h(L) \quad \leftarrow\ & p(L,X_1), p(L,X_2), p(L,X_3), p(L,X_4), \\ & c(X_1, NX_2, X_4), c(NX_2, NX_3, NX_4), \\ & c(NX_1, 0, 0), c(X_1, X_2, 0), \\ & ne(X_1, NX_1), ne(X_2, NX_2), \\ & ne(X_3, NX_3), ne(X_4, NX_4). \end{aligned}$$

In this case, (V, S) is satisfiable and D θ–subsumes our fixed Horn clause C with the substitution
$\theta = \{L/sat, X_1/0, X_2/1, X_3/0, X_4/1, NX_1/1, NX_2/0, NX_3/1, NX_4/0\}$,
which is indeed a valid truth assignment for V satisfying all clauses in S.

Therefore, not even fixing the size of C (i.e., the background knowledge) to a small number of literals reduces the complexity of $\vdash_\theta$.

3.1 θ–SUBSUMPTION BETWEEN DETERMINATE HORN CLAUSES IS IN P

The inefficiency of $\vdash_\theta$ is caused by the nondeterminism of choosing θ. If we restrict D (i.e. LH) to a clause containing only determinate literals, then $\vdash_\theta$ is polynomially computable.

Definition 5 *Let $D = D_0 \leftarrow D_{Body}$ and $C = C_0 \leftarrow C_{Body}$ be two Horn clauses. D deterministically θ–subsumes C, written $D \vdash_{\theta DET} C$, by $\theta = \theta_0\theta_1 \ldots \theta_n$ iff $D_0\theta_0 = C_0$ and there exists an ordering $D'_{Body} = D_1, \ldots, D_n$ of D_{Body} such that for all $i, 1 \leq i \leq n$, there exists exactly one θ_i such that $\{D_1, \ldots, D_i\}\theta_0\theta_1 \ldots \theta_i \subseteq C_{Body}$.*

Theorem 2 *Testing $D \vdash_{\theta DET} C$ can be computed with $O(|D|^2 * |C|)$ basic unification attempts.*

Proof:
Algorithm 1 is a straight–forward Prolog implementation of definition 5. It copies C and D in order to hide the unifications performed. It skolemizes C so that unifications of C and D can only affect D as $\vdash_{\theta DET}$ requires. To see the correctness of `find-det-sub`, assume in the first line `select` replaced by `select==`, where `select==` is equal to `select`, except that the first clause uses == instead of =. Assume further that we guess a correct order D'_{Body} of D_{Body} and that we call `find-det-sub` with `Det` $= D'_{Body}$. Then, the first line of `find-det-sub` verifies that the first element of D'_{Body} is indeed an element of D_{Body}, the second line verifies, that there exists a θ_i such that `L`$\theta_i \in C_{Body}$ and applies this θ_i to D'_{Body} and D_{Body}, and the third line verifies that this θ_i is unique. All this is completely deterministic, so the cut does nothing. Finally, these checks are performed on all further elements of D'_{Body} by recursion, which ends if D'_{Body} is empty. This means that under these assumptions `find-det-sub` verifies our guess of D'_{Body}. The correctness of `find-det-sub` without these assumptions then follows from the fact that $\theta = \theta_0\theta_1 \ldots \theta_n$ must be a valid substitution, and each θ_i produces ground terms if $D \vdash_{\theta DET} C$. This means that the order of the θ_i and therefore the D_i is irrelevant as long as the substitutions performed so far ensure the uniqueness of θ_i. This is exactly what happens in `find-det-sub`. It selects the first literal of D_{Body} which uniquely matches into C_{Body}.

The computations performed in `subsumes-determ` need less than $O(|C| + |D|)$ unification attempts. The first clause of `find-det-sub` needs less than $O(|C| * |D|)$ unification attempts per invocation and as `DBody` is reduced by one on each invocation, it is invoked less than $|D|$ times. Therefore, `subsumes-determ` needs $O(|D|^2 * |C|)$ unification attempts. □

Algorithm 1 [3]

```
subsumes-determ(D,C):-
    copy_term(D,DHead-DBody),
    copy_term(C,CHead-CBody),
    skolemize(CHead-CBody),
    DHead = CHead,
    find-det-sub(DBody,CBody,_,[]).
find-det-sub(DBody,CBody,[L|Det],NDet):-
    select(L,DBody,DBodyR),
    copy_term(L,LCopy),
    (select(L,CBody,CBodyR)
          -> true | !, fail),
    (select(LCopy,CBodyR,_)
          -> false | true),
    !,
    find-det-sub(DBodyR,CBody,Det,NDet).
find-det-sub(NDet,_,[],NDet).
select(E,[F|R],R):- E = F.
select(E,[F|R],[F|NR]):- select(E,R,NR).
```

This definition of determinate–subsumption is strongly related to the definition of a determinate clause of Muggleton and Feng (1990). There are only two small differences. First, we have defined determinacy with respect to one other clause, instead of examples and background knowl-

[3] This algorithm assumes, that a Horn clause $L_0 \leftarrow L_1, \ldots, L_n$ is represented as L_0`-[`L_1`,...,`L_n`]`, `copy_term` is a built–in predicate, which makes a copy of the first argument, where each variable is replaced by a variable not occurring anywhere else, and `skolemize` is a built–in predicate, which unifies each variable with a constant not occurring anywhere else, and `(A -> B | C)` is a built–in predicate, which implements `if A then B else C` such that goals in `A`, `B`, and `C` (especially cuts) are executed in the context of the clause in which `(A -> B | C)` appears.

edge. This is only a small difference. If we use the transformation from section 2 and define determinacy to a set S of clauses as determinate with respect to every clause in S the difference disappears. Secondly, our definition does not depend on a preexisting order of the literals in D. This makes a real difference if we look at determinacy with respect to several clauses. Whereas Muggleton and Feng's definition requires one order, which makes the clause determinate with respect to all clauses our definition allows the order to change, depending on the clause against which determinate–subsumption is checked. This leads to the following corollary.

Corollary 3 *Let D and $C = C_0 \leftarrow C_1, \ldots, C_m$ be two Horn clauses. If D is determinate according to the definition of (Muggleton and Feng, 1990) with respect to $E = \{C_0\}, B = \{\{C_1\}, \ldots \{C_m\}\}$ and $D \vdash_\theta C$, then $D \vdash_{\theta DET} C$. If $D \vdash_{\theta DET} C$, then there exists an ordering D' of D, such that D' is determinate according to the definition of Muggleton and Feng with respect to $E = \{C_0\}, B = \{\{C_1\}, \ldots \{C_m\}\}$.*

Note, that the algorithm 1 not only computes $D \vdash_{\theta DET} C$, but that the second clause of `find-det-sub` also computes the remaining non–determinate literals $D_{NONDET} = D/D_{DET}$ if $D_{DET} \vdash_{\theta DET} C$. The correctness of this follows immediately from theorem 2. In this view algorithm 1 is not only a subsumption algorithm for determinate clauses but also an improvement of the subsumption algorithm for all clauses.

Corollary 4 *Let D and C be two Horn clauses , D_{DET} the maximal subclause of D which deterministically θ–subsumes C and $D_{NONDET} = D \setminus D_{DET}$. Then $D \vdash_\theta C$ iff $D_{DET} \vdash_{\theta DET} C$ by θ_{DET} and $D_{NONDET}\theta_{DET} \vdash_\theta C$. The complexity of splitting D into D_{DET} and D_{NONDET} and testing $D_{DET} \vdash_{\theta DET} C$ is $O(|D_{DET}| * |D| * |C|)$.*

3.2 θ–SUBSUMPTION BETWEEN k–LOCAL HORN CLAUSES IS IN P

We have seen how we can reduce the complexity of $\vdash_\theta$ for the determinate part of a clause. However, the remaining non–determinate part may be large, resulting in an expensive test of $\vdash_\theta$ for the non–determinate part. In this section, we will describe a further improvement to be applied to the non–determinate part. The improvement to $D \vdash_\theta C$ is to compute the independent parts (locals) of D_{NONDET}, which results in independent subsumption tests of disjoint subsets of D_{NONDET}.

Definition 6 *Let $D = D_0 \leftarrow D_{DET}, D_{NONDET}$ be a Horn clause. Let $vars$ be a function computing the set of variables of a clause. $LOC_i \subseteq D_{NONDET}$ is a non–determinate local of D, iff $(vars(LOC_i) \setminus vars(\{D_0, D_{DET}\})) \cap vars(D_{NONDET} \setminus LOC_i) = \emptyset$ and there does not exist a $LOC_j \subset LOC_i$, which is also a non–determinate local of D. A nondeterminate local LOC_i is a k–local for a constant k iff $k \geq min(|(vars(LOC_i) \setminus vars(\{D_0, D_{DET}\})|, |LOC_i|)$. A clause is k–local iff every non–determinate local is a k–local.*[4]

Theorem 5 *Let C be an arbitrary Horn clause, and $D = D_0 \leftarrow D_{DET}, LOC_1, \ldots, LOC_n$ be a Horn clause where $D_0 \leftarrow D_{DET}$ is determinate with respect to C and each $LOC_i, 1 \leq i \leq n$, is a k–local. $D \vdash_\theta C$ can be decided with $O(|D| * |D_{DET}| * |C| + |LOC_1, \ldots, LOC_n|^2 + n * (k^k * |C|))$ basic unification steps.*

Proof:
Algorithm 2 does this. The correctness of this algorithm follows immediately from theorem 2, corollary 4 and the fact that any substitution θ_i necessary for $LOC_i\theta_{DET}\theta_i \subseteq C$ does not affect any $LOC_j\theta_{DET}, i \neq j$, i.e., from the fact that the LOC_i do not share non–determinate variables. The complexity is easy to see. □

Algorithm 2 [5]

```
subsumes-determinate-local(D,C):-
    copy_term(D,DHead-DBody),
    copy_term(C,CHead-CBody),
    skolemize(CHead-CBody),
    DHead = CHead,
    find-det-sub(DBody,CBody,_,DNonDet),
    find-locals(DNonDet,DLocals),
    subsumes-locals(DLocals,CBody).
subsumes-locals([],_).
subsumes-locals([LOC|Locals],CBody):-
    theta-subsumes(LOC,CBody),
    !,
    subsumes-locals(Locals,CBody).
```

[4] Note that if we use `find-det-sub` to find D_{DET} and D_{NONDET}, we need not consider $vars(\{D_0, D_{DET}\})$ as these are grounded due to the skolemization of C and the application of θ_{DET} to D_{DET} and D_{NONDET}.

[5] We do not include a definition of `find-locals`, as this is basically a set operation. The idea is as follows. For each local `find-locals` remembers vars and literals in two sets. The algorithm starts with no locals. Then for each literal in `DNonDet` it looks for an intersection between the variables of the literals and the variables of a local. All locals intersecting with the variables of the literal are merged together with the literal into one new local. This can performed with $O(|D|^2)$ calls to intersection and union. `theta-subsumes` is a call to the standard θ–subsumption algorithm. There are several possibilities. The simplest implementation would be `theta-subsumes([],_).` `theta-subsumes([L|Lits],C):- member(L,C),` `theta-subsumes(Lits,C).` The complexity of this algorithm is $O(|D|^{|D|} * |C|)$.

Recently, Cohen (1993) used the notion of k–locality to characterize a polynomially predictable hypothesis space in ILP. This section can be seen as describing a polynomial algorithm for testing these hypotheses against the examples, i.e., as one step towards a learning algorithm for this hypothesis space.

This algorithm can also be seen as an improvement of θ–subsumption for general Horn clauses. However, it should be noted that in the worst case $|D|$ consist of one big non–determinate local, which gives the algorithm the worst–case complexity $O(|D|^{|D|} * |C|)$. This is not surprising as all known correct algorithms for NP–complete problems have an exponential worst–case complexity. In section 5 we therefore investigate into several speed tests comparing different θ–subsumption algorithms used in ILP. Gottlob and Leitsch (1985) previously presented a divide–and–conquer θ–subsumption algorithm based on the independence of subsumption between locals. They have shown theoretically, that for several worst–case situations their algorithm behave better — but still exponential — than simple resolution or backtracking based methods. However, these authors used locality only as a heuristic method, which tries to split a clause into 1–local clauses, they did not show, that subsumption between k–local clauses is in P, nor did they notice the improvements possible due to determinacy. Both notions, k-locality and determinancy, are not unrealistic artificial restrictions, but are strongly motivated by current ILP results.

4 IMPROVING THE REDUCTION OF CLAUSES

Now let us show how the ideas of this section can be used to improve the related problem of reduction under θ–subsumption in a more direct manner than simply using the improved θ–subsumption algorithm.

Most ILP–learning algorithm have in a way to deal with the reduction of redundancy. One type of redundancy elimination most often used in lgg–based learning systems is based on the equivalence relation induced by θ–subsumption: $D \sim C$ iff $C \vdash_\theta D$ and $D \vdash_\theta C$. Plotkin defines reduced clauses as clauses containing no redundant literals with respect to θ–subsumption (Plotkin, 1970).

Definition 7 *A clause D is reduced iff $C \subseteq D$ and $C \sim D$ imply $C = D$.*

Gottlob and Fermüller (1993) have recently shown, that deciding whether a clause D is reduced is co–NP–complete, and that finding the reduction of a clause D requires a at least a linear (in $|D|$) number of calls to an NP oracle ($\vdash_\theta$).

Our improvement of the reduction algorithm is based on the same ideas as used for improving $\vdash_\theta$. Given the clause D we first identify the maximal subclause D_{DET} of D which deterministically θ–subsumes D. We first show, that D_{DET} does not contain redundant literals. This means, that a Horn clause which deterministically θ–subsumes itself does not contains redundant literals. Moreover, this means, that we must only reduce the nondeterminate part of D.

Theorem 6 *Let $D = D_0 \leftarrow D_{DET}, D_{NONDET}$ be a Horn clause, where $D_0 \leftarrow D_{DET}$ is the maximal subclause of D, such that $(D_0 \leftarrow D_{DET}) \vdash_{\theta DET} D$. Then the reduction of D is the clause $D_0 \leftarrow D_{DET}, D^R_{NONDET}$, where D^R_{NONDET} is the minimal subset of D_{NONDET}, such that the following holds:*
$D \vdash_\theta (D_0 \leftarrow D_{DET}, D^R_{NONDET})$.

Proof:
There is exactly one substitution θ_{DET}, whose domain are the variables of D_{DET}, such that $D_{DET}\theta_{DET} \subseteq D$. It follows that $\theta_{DET} = \{X/X|X \in vars(D_{DET})\}$, i.e.: $D_{DET}\theta_{DET} = D_{DET}$.
Let D^R be the reduction of D. It holds that $D^R \sim D$
$\Rightarrow \quad D^R \subseteq D \wedge \exists\theta : D\theta \subseteq D^R$
$\Rightarrow \quad D_{DET}\theta \subseteq D^R \subseteq D$
$\Rightarrow \quad \theta_{DET} \subseteq \theta$
$\Rightarrow \quad D_{DET}\theta = D_{DET} \subseteq D^R$.
Let $D^R_{NONDET} = D^R \setminus D_{DET}$. It holds that $D^R_{NONDET} \subseteq D_{NONDET}$ and $D \vdash_\theta D^R = (D_0 \leftarrow D_{DET}, D^R_{NONDET})$. In addition D^R_{NONDET} must be a minimal subset of D_{NONDET} with these properties, otherwise D^R would not be the reduction of D. □

The notion of locality is also very useful for reduction. First of all we may be able to drop a whole local, i.e., if LOC_i is a local of D_{NONDET}, LOC_i can be removed from the clause, if $LOC_i \vdash_\theta (D \setminus LOC_i)$. Even if only parts of the locals are redundant, we can reduce them independently, i.e. by computing for every LOC_i the minimal subset $LOC^R_i \subseteq LOC_i$ such that $LOC_i \vdash_\theta (LOC^R_i \cup D_{DET} \cup \bigcup^j_{j\neq i} LOC_j)$. This leads to the following theorem.

Theorem 7 *Let $D_0 \leftarrow D_{DET}, LOC_1, \ldots, LOC_n$ be a Horn clause where $D_0 \leftarrow D_{DET}$ deterministically θ–subsumes D and each LOC_i is a k–local of D_{NONDET}. The reduction D^R of D can be computed with $O(|D_{DET}| * |D|^2 + |LOC_1, \ldots, LOC_n|^2 + n * (k^{k+1} * |D|))$ basic unification attempts.*

Proof:
The correctness of this algorithm follows from theorem 6 and the fact that all substitutions neccessary to remove redundant literals from a local LOC_i do not affect any

$LOC_j, j \neq i$. [6]. The complexity follows from theorem 5. □

Note, that this algorithm also uses atmost $|D_{NONDET}|$ calls to the NP–complete θ–subsumption algorithm, as the algorithm of (Gottlob and Fermüller, 1993). However, these calls involve much smaller clauses as in the algorithm of (Gottlob and Fermüller, 1993). As, we do not have an NP–oracle with unit computation time, but a worst case exponential algorithm, this gives a great pay–off.

Like our algorithm for testing θ–subsumption between general Horn clauses this reduction algorithm has an exponential worst case complexity $O(|D|^{|D|})$, if $|D|$ consists of one non–determinate local.

5 TESTING THE EFFICIENCY

To analyse the efficiency of our algorithms we compare them with other algorithms for testing θ–subsumption between Horn clauses and removing redundant literals from a clause.

We have tested the efficiency of the algorithms on several domains.

1. We demonstrate the need for an efficient θ–subsumption algorithm, by showing, that even in the toy–domain of learning the definiton of an arch, naive implementations of θ–subsumption as used in the descriptions of several ILP–learners are not pratically useable.

2. We use artificial, constructed determinate k–local Horn clauses of increasing size to demonstrate the usefulness of the algorithms presented in section 3 for learning determinate k–local clauses.

3. We summarize the results of reducing lgg's and testing lgg's against the examples in several well known ILP–domains, i.e. krk (Quinlan, 1990), mesh (Dolsak and Muggleton, 1992), and speed (Sommer et al., 1993).

For this comparison we choose the following algorithms, implemented in Quintus Prolog and running on a Sun Sparcstation IPX.

- `select-best-subsumes`: A θ–subsumption algorithm, which tries to reduce backtracking by using maximally instantiated literals first. This algorithm is actually used in our implementation of `subsumes-determinate-local` to test the subsumption between locals. This algorithm additionally remembers all literals of C which are used to show $D \vdash_\theta C$. This is used by the reduction algorithm `select-best-reduce`, which is based on this subsumption algorithm.

- `miles-correct-reduce`: an algorithm for reducing clauses in the ILP–toolbox MILES using resolution methods (Stahl and Tausend, 1994).

- `prolog-subsumes`[7], this is based on prologs SLD–resolution. Something similar is proposed by (De Raedt and Bruynooghe, 1993) and used in MILES.

- `simple-subsumes`: the simple implementation of θ–subsumption described in footnote 5 in this paper and in (Idestam-Almquist, 1993).

The clause in table 1 is the unreduced Plotkin–lgg of two arch examples of the blocksworld domain. Due to the `not_touches` literal, parts of this clause deterministically θ–subsumes itself according to definition 5. We compute all possible subclauses of the clause `arch` and test if they are subsumed by `arch`. The results are represented in figure 1. Some of the smaller subclauses, up to a length of 10 literals, are not subsumed by the clause `arch`, this is responsible for the in between higher computation times. Qur `subsumes-determinate-local` and `select-best-subsumes` need nearly constant time, whereas `simple-subsumes` increases very fast and `prolog-subsumes` even exceeds the time limit.

In order to test different determinate k–local Horn clauses with an increasing number of literals we construct clauses whose structure can be determined by three parameters d, r and k. With respect to itself all clauses have a determinate subset, whose size is $2d + 2$, and r k–locals. Each local θ–subsumes all other locals and in each local exists exactly one redundant literal. So the reduced clause is of size $2d + 2 + (k - 1)$. We rise d, r and k step by step, compute the reduced clause and test if the not reduced clause θ–subsumes the former one. The results are shown in figure 2. Because of the order of the literals in the construction of these clauses the standard θ–subsumption algorithms do not need to backtrack. Therefore, the subsumption tests of the not permutated clauses result in a nearly constant time behaviour of the algorithms `simple-subsumes` and `prolog-subsumes`. `select-best-subsumes`

[6] Note that using `find-det-sub` all $vars(\{D_0, D_{DET}\})$ have been replaced by ground terms, so that the locals share no variables.

[7] `prolog-subsumes` is the simple θ–subsumption algorithm, using Prolog as a theorem prover:

```
prolog-subsumes(DHead-DBody,CHead-CBody):-
skolemize(CHead-CBody),  (member(L,CBody),
assert(L),fail|true),  DHead=CHead,!,
call(DBody).
```

and `subsumes-determinate-local` increases linear, but up to the tested length of 120 literals just `simple-subsumes` is faster than the latter one.

The situation becomes different if we permutate the literals in the clauses randomly. As expected, `simple-subsumes` and `prolog-subsumes` show an exponential time behaviour due to their nondeterminism in choosing θ. Their computations have to be aborted for even small clauses. `subsumes-determinate-local` and `select-best-subsumes` do not spend much more time for these tests than they do for the ordered clauses. The reason for this is that the heuristic of `select-best-subsumes` sorts the given clause, so that backtracking can be avoided to a large extent. The results of testing the permutated clauses are displayed in figure 3. Reducing these clauses with the mentioned algorithms leads to similar results. `select-best-reduce` and `reduce-determinate-local` are faster than `miles-correct-reduce` if clauses are not permutated. `select-best-reduce` exceeds the time limit while reducing permutated clauses with more than 60 literals, the other two behave like before.

As a final test in this domain, we tried the θ–subsumption algorithms on clauses which do not subsume each other. This is achieved by adding an additional literal to the unreduced clause, which therefore cannot be mapped to the reduced one. Our new algorithm shows clearly better results than the others, as can be seen in figure 4. The other algorithms have to try each possible substitution, that means they carry out full exponential backtracking, which explains their bad time behaviour. They all exceed the time limit while testing even small clauses. Like for permutated clauses `select-best-subsumes` sorts the given clause to avoid backtracking, but as all these algorithms are complete, this algorithm has to do a complete search, if the heuristic order does not lead to success. Therefore it computes as many unifications as the former ones. In contrast, `subsumes-determinate-local` identifies the determinate part, which needs no backtracking, and splits the remaining part into independent sets, the locals. This leads to a much lower computation time, because the algorithm correctly stops, if one of these sets does not subsume the clause C.

Finally, we have investigated how these algorithms perfom in practice. We used a Plotkin–lgg based ILP algorithm to learn concepts in different domains. Among these domains is the well known krk domain (Quinlan, 1990), which models an endgame situation of chess, the finite element mesh design domain (Dolsak and Muggleton, 1992), and speed, a domain for managing the security policy of distributed open systems (Sommer et al., 1993). We use the reduction–algorithms to remove redundant literals from the lgg's and the subsumption–algorithms to test which examples are covered by the found hypotheses. No reduction algorithm performs faster than the others in all tests, but in the average `miles-correct-reduce` spends the most time for removing the redundant literals and `reduce-determinate-local` is the most robust one, i.e., in contrast to the others, it rarely exceeds the time limit of 30 minutes. Examples which are covered by the hypotheses are recognized by all subsumption algorithms very fast, just for not covered examples `subsumes-determinate-local` is much superior.

In summary, one can say that in all tests the new algorithms perform even as good or better than the naive procedures or the heuristic one. In addition they even allow efficient computations in some cases for which standard procedures need exponential time.

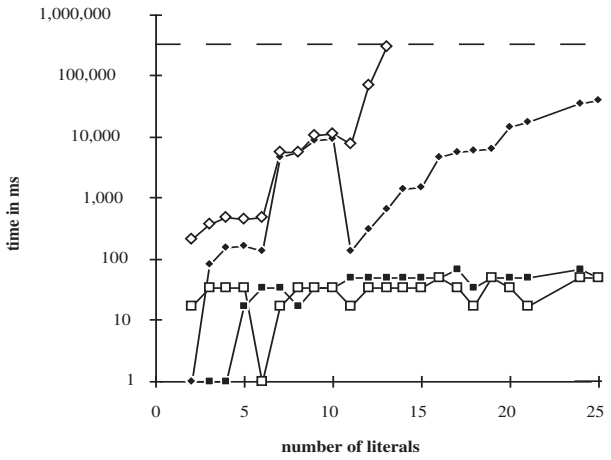

Figure 1: θ–subsumption tests between `arch`–subsets.

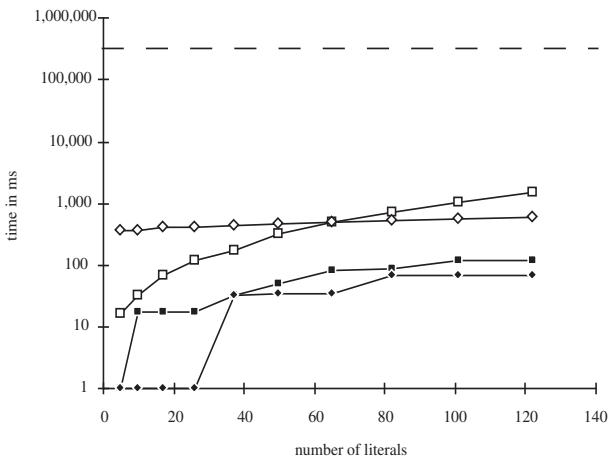

Figure 2: Testing $D \vdash_\theta C$ for ordered clauses.

Table 1: The unreduced lgg of two arches.

$$
\begin{aligned}
arch(K) \leftarrow\ & \\
& part(K,L), part(K,T), part(K,M), \\
& part(K,N), part(K,O), part(K,P), \\
& part(K,Q), part(K,R), part(K,S), \\
& rectangle(L), rectangle(T), rectangle(M), \\
& rectangle(N), rectangle(O), rectangle(P), \\
& on_top_of(S,L), on_top_of(S,T), \\
& on_top_of(S,N), on_top_of(S,O), \\
& touches(S,L), touches(S,T), \\
& touches(S,N), touches(S,O), \\
& not_touches(L,O).
\end{aligned}
$$

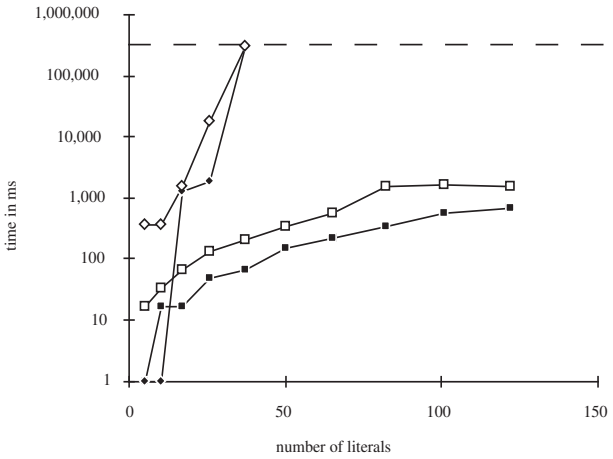

Figure 3: Testing $D \vdash_\theta C$ for permutated clauses.

6 CONCLUSION

We have shown that the efficiency of θ–subsumption is a necessary prerequisite of all known ILP–algorithms. We have shown that the general problem $D \vdash_\theta C$ is NP–complete even if we restrict ourselves to linked Horn clauses and fix C to contain only a reasonably small number of literals. The main part of the paper has shown how some hypothesis space restrictions of ILP, namely determinate and k–local clauses, can be used to greatly improve the efficiency of θ–subsumption. These implemeted θ–subsumption algorithms, which are supported by theoretical results and pratical tests, provide a necessary prerequisite for efficiently learning determinate k–local Horn clauses, an ILP–problem recently shown to be polynomially predictable (Cohen, 1993) (based on a counting argument, but without giving a learning algorithm for that problem). Together with the improved θ–reduction algorithm for determinate and k–local clauses, our results will probably enable to implement an efficient lgg–based learning algorithm for that hypothesis space. The authors will further investigate in this promissing direction.

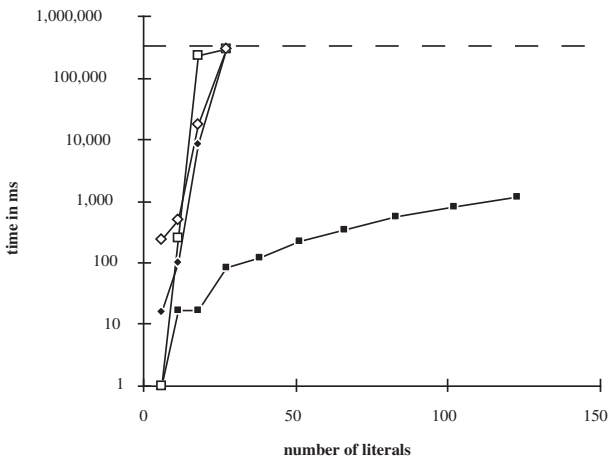

Figure 4: Testing $D \vdash_\theta C$ in the case of $D \not\vdash_\theta C$.

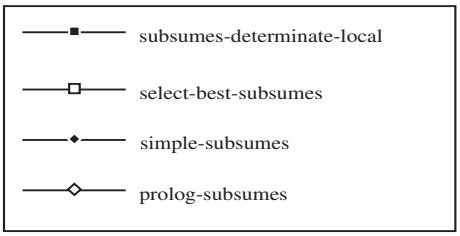

Figure 5: The legend for figures 2, 3, 4 and 1. Note, that in all figures the ordinate has a logarithmic scale. The time limit of 5 minutes is indicated by the broken line on top.

Acknowledgements

Parts of this work are partially supported by the European Community ESPRIT program under contract number P6020 “Inductive Logic Programming”. The authors thank Katharina Morik, Werner Emde, Stefan Wrobel and the anonymous reviewers for commenting earlier versions of the paper.

References

Baxter, L. D. (1977). The NP–completeness of subsumption. unpublished manuscript.

Buntine, W. (1988). Generalized subsumption and its applications to induction and redundancy. *Artificial Intelligence*, 36:149 – 176.

Cohen, W. W. (1993). Learnability of restricted logic programs. In Muggleton, S., editor, *Proc. Third International Workshop on Inductive Logic Programming, ILP'93*, pages 41–71. Jožef Stefan Institute, Ljubljana, Slovenia.

De Raedt, L. (1992). *Interactive Theory Revision: An Inductive Logic Programming Approach*. Academic Press, London.

De Raedt, L. and Bruynooghe, M. (1993). A theory of clausal discovery. In Muggleton, S., editor, *The Third International Workshop on Inductive Logic Programming*.

Dolsak, B. and Muggleton, S. (1992). The application of inductive logic programming to finite element mesh design. In Muggleton, S., editor, *Inductive Logic Programming*, pages 453–472. Academic Press.

Garey, M. R. and Johnson, D. S. (1979). *Computers and Intractability - A Guide to the Theory of NP-Completeness*. Freeman, San Francisco, Cal.

Gottlob, G. (1987). Subsumption and implication. *Information Processing Letters*, 24:109–111.

Gottlob, G. and Fermüller, C. G. (1993). Removing redundancy from a clause. *Artificial Intelligence*, 61:263–289.

Gottlob, G. and Leitsch, A. (1985). On the efficiency of subsumption algorithms. *Journal of the Association for Computing Machinery*, 32(2):280–295.

Idestam-Almquist, P. (1993). *Generalization of clauses*. PhD thesis, Stockholm University.

Kapur, D. and Narendran, P. (1986). NP–completeness of the set unification and matching problems. In *8th International Conference on Automated Deduction*, LNCS 230. Springer.

Kietz, J.-U. and Wrobel, S. (1992). Controlling the complexity of learning through syntactic and task-oriented models. In Muggleton, S., editor, *Inductive Logic Programming*, pages 107 – 126. Academic Press.

Lavrač, N. and Džeroski, S. (1993). *Inductive Logic Programming: Techniques and Applications*. Ellis Horwood, Chichester.

Lloyd, J. W. (1987). *Foundations of Logic Programming*. Springer, Berlin, New York, 2nd edition.

Morik, K., Wrobel, S., Kietz, J.-U., and Emde, W. (1993). *Knowledge Acquisition and Machine Learning: Theory, Methods and Applications*. Academic Press, London.

Muggleton, S. H., editor (1992). *Inductive Logic Programming*. Academic Press, London.

Muggleton, S. H. (1993). Inverting implication. *Artificial Intelligence Journal*. To appear.

Muggleton, S. H. and Feng, C. (1990). Efficient induction of logic programs. In *Proc. First Conference on Algorithmic Learning Theory*, pages 368–381. Ohmsha, Tokyo.

Plotkin, G. D. (1970). A note on inductive generalization. In Meltzer, B. and Michie, D., editors, *Machine Intelligence*, volume 5, chapter 8, pages 153 – 163. American Elsevier.

Plotkin, G. D. (1971). *Automatic Methods of Inductive Inference*. PhD thesis, Edinburgh University.

Quinlan, J. R. (1990). Learning logical definitions from relations. *Machine Learning*, 5(3):239 – 266.

Robinson, J. (1965). A machine-oriented logic based on the resolution principle. *Journal of the ACM*, 12(1):23 – 41.

Shapiro, E. Y. (1983). *Algorithmic Program Debugging*. ACM Distinguished Doctoral Dissertations. The MIT Press, Cambridge, Mass.

Sommer, E., Morik, K., Andre, J.-M., and Uszynski, M. (1993). What online machine learning can do for knowledge acquisition — a case study. Technical report, GMD Report No. 757.

Stahl, I. and Tausend, B. (1994). *MILES– a Modular Inductive Logic Programming Experimentation System*. Institut für Informatik, Universität Stuttgart.

van der Laag, P. and Nienhuys-Cheng, S.-H. (1993). Subsumption and refinement in model inference. In Brazdil, P. B., editor, *European Conference on Machine Learning*, volume 667 of *Lecture Notes in Artificial Intelligence*. Springer.