Learning First-Order Acyclic Horn Programs from Entailment*

Chandra Reddy **Prasad Tadepalli**

Dearborn 303
Department of Computer Science
Oregon State University,
Corvallis, OR 97331-3202.
{reddyc,tadepalli}@cs.orst.edu

Abstract. In this paper, we consider learning first-order Horn programs from entailment. In particular, we show that any subclass of first-order acyclic Horn programs with constant arity is exactly learnable from equivalence and entailment membership queries provided it allows a polynomial-time subsumption procedure and satisfies some closure conditions. One consequence of this is that first-order acyclic determinate Horn programs with constant arity are exactly learnable from equivalence and entailment membership queries.

1 Introduction

Learning first-order Horn programs—sets of first-order Horn clauses—is an important problem in inductive logic programming with applications ranging from speedup learning to grammatical inference.

We are interested in speedup learning, which concerns learning domain-specific control knowledge to alleviate the computational hardness of planning. One kind of control knowledge, which is particularly useful in many domains, is represented as goal-decomposition rules. Each decomposition rule specifies how a goal can be decomposed into a sequence of subgoals, given that a set of conditions is true in the initial problem state. Each of the subgoals might in turn have a set of decomposition rules, unless it is a primitive action, in which case it can be directly executed.

Unlike in logical inference, for which Horn clauses are ideally suited, in planning, one needs to keep track of time. In spite of this difference, goal-decomposition rules can be represented as first-order Horn clauses by adding two situation variables to each literal to indicate the time interval in which the literal is true. Hence, the problem of learning goal-decomposition rules for a *single* goal can be mapped to learning first-order Horn definitions—a set of Horn clauses, all having the same head or consequent literal. Learning goal-decomposition rules for *multiple* goals corresponds to learning first-order Horn programs. Henceforth, we omit the prefix "first-order", except when there is a possibility of ambiguity.

* This paper also appears in the proceedings of 15th International Conference on Machine Learning, 1998. Reprinted here with permission.

In learning from entailment, a positive (negative) example is a Horn clause that is implied (not implied) by the target. Results by Cohen (1995a, 1995b), Dzeroski et al. (1992) and others indicate that classes of Horn programs having a single or a constant number of clauses are learnable from examples. Khardon shows that "actions strategies" consisting of a variable number of constant-size first-order production rules can be learned from examples (Khardon, 1996). However, Cohen (1995a) proves that even predicting very restricted classes of Horn programs (viz. function-free 0-depth determinate constant arity) with variable number of clauses of variable size from examples alone is cryptographically hard.

Frazier and Pitt (1993) first used the entailment setting for learning arbitrary propositional Horn programs. In addition to examples, they also used entailment membership queries ("entailment queries" from now on) which ask if a Horn clause is entailed by the target. Moving to first order representations, Frazier and Pitt (1993) showed that CLASSIC sentences are exactly learnable in polynomial time from examples and entailment queries. A Horn clause is simple if the terms and the variables in the body of the clause are restricted to the terms that appear in the head. Page (1993) considered non-recursive Horn programs restricted to simple clauses and predicates of constant arity, and showed that they are learnable from examples and entailment queries. Arimura (1997) generalized Page's result to acyclic (possibly, recursive) simple Horn programs with constant-arity predicates. Reddy and Tadepalli (1997b) showed that function-free non-recursive Horn definitions are learnable from examples and entailment queries. The result we present here applies to non-generative Horn programs, where the variables and the terms in the head are restricted to those in the body. We show that acyclic non-generative Horn programs with constant arity that have polynomial-time subsumption procedure are learnable from examples and entailment queries when certain closure conditions are satisfied. In particular, the result applies to acyclic Horn programs with constant arity determinate clauses.

Goal-decomposition rules are hierarchical in nature, as are Horn programs. One aspect of learning in hierarchical domains is the hierarchical order of literals (goals or concepts). In many systems, learning hierarchically organized knowledge assumes that the structure of hierarchy or the order of the literals is known to the learner. Examples of such work include Marvin (Sammut & Banerji, 1986) and XLearn (Reddy & Tadepalli, 1997a), on the experimental side; learning from exercises by Natarajan (1989) and learning acyclic Horn sentences by Arimura (1997), on the theoretical side. In fact, Khardon shows that learning hierarchical strategies can be computationally hard when the structure of the hierarchy is not known (Khardon, 1996). Our algorithm also assumes that the hierarchical order of the literals is known.

The rest of the paper is organized as follows. Section 2 provides definitions for some of the terminology we use. Section 3 describes the learning model and the learning algorithm, and proves the learnability result. Section 4 concludes the paper with some discussion on implications and limitations of the work.

2 Preliminaries

In this section, we define and describe some of the terminology we use in the rest of the paper. For brevity, we omit some of the standard terminology (as given in books such as (Lloyd, 1987)). In the following, we use p and its variants, and a and its variants each to stand for a conjunction of literals; and b, q, l and their variants each to stand for a single literal.

Definition 1. A **definite Horn clause** (Horn clause or clause, for short) is a finite set of literals that contains exactly one positive literal—$\{l, \neg l_1, \neg l_2, \ldots, \neg l_n\}$. It is treated as a disjunction of the literals in the set with universal quantification over all the variables. Alternately, it is represented as $l_1, l_2, \ldots, l_n \rightarrow l$, where l is called the *head* or *consequent*, and $l_1, l_2, \ldots, l_n$ is called the *body* or *antecedent* and is interpreted as $l_1 \wedge l_2 \wedge \ldots \wedge l_n$. A **unit Horn clause** is a Horn clause with no negative literals and hence no body. A **Horn program** or **Horn sentence** is a set of definite Horn clauses interpreted conjunctively.

Definition 2. Let C_1 and C_2 be sets of literals. We say that C_1 **subsumes** C_2 (denoted $C_1 \succeq C_2$) iff there exists a substitution θ such that $C_1\theta \subseteq C_2$. We also say C_1 is a generalization of C_2.

Definition 3 (Plotkin, 1970). Let C, C', C_1 and C_2 be sets of literals. We say that C is the **least general generalization** (lgg) of C_1 and C_2 iff $C \succeq C_1$ and $C \succeq C_2$, and $C' \succeq C$, for any C' such that $C' \succeq C_1$ and $C' \succeq C_2$.

Definition 4 (Plotkin, 1970). A **selection** of clauses C_1 and C_2 is a pair of literals (l_1, l_2) such that $l_1 \in C_1$ and $l_2 \in C_2$, and l_1 and l_2 have the same predicate symbol, arity, and sign.

If C_1 and C_2 are sets of literals, then $lgg(C_1, C_2)$ is $\{lgg(l_1, l_2) : (l_1, l_2)$ is a selection of C_1 and $C_2\}$. If l is a predicate, $lgg(l(s_1, s_2, \ldots, s_n), l(t_1, t_2, \ldots, t_n))$ is $l(lgg(s_1, t_1), \ldots, lgg(s_n, t_n))$. The lgg of two terms $f(s_1, \ldots, s_n)$ and $g(t_1, \ldots, t_m)$, if $f = g$ and $n = m$, is $f(lgg(s_1, t_1), \ldots, lgg(s_n, t_n))$; else, it is a variable x, where x stands for the lgg of that pair of terms throughout the computation of the lgg of the set of literals.

As an example, let C_1 be $l(a, b), l(b, c), m(b) \rightarrow l(a, c)$, and C_2 be $l(1, 2), l(2, 3), m(2) \rightarrow l(1, 3)$. $(l(a, c), l(1, 3))$ and $(\neg m(b), \neg m(2))$ are two of the selections of C_1 and C_2. $lgg(C_1, C_2)$ is $l(x, y), l(y, z), l(t, u), l(v, w), m(y) \rightarrow l(x, z)$, where x, y, z, t, u, v and w are variables standing for the pairs $(a, 1), (b, 2), (c, 3), (a, 2), (b, 3), (b, 1)$ and $(c, 2)$.

Definition 5. A **derivation** of a Horn clause $p \rightarrow q$ from a Horn program H is a finite directed acyclic graph G such that there is a node q, there is no arc (q, r) in G, and for each node l in G, either $l \in p$ or if $(l_1, l), \ldots, (l_d, l)$ are the only arcs of G terminating at l, then $l_1, \ldots, l_d \rightarrow l = C\theta$ for some clause $C \in H$ and a substitution θ.

For example, let H be $\{parent(x, y), parent(y, z) \rightarrow grandParent(x, z)$; *mother*$(x, y) \rightarrow parent(x, y)\}$. Figure 1 shows a derivation of $mother(a, b)$, *mother*$(b, c) \rightarrow grandParent(a, c)$.

Proposition 6. *In a derivation G of a clause $p \rightarrow q$ from a Horn program H, for any node l, either l is in p or $H \models p \rightarrow l$.*

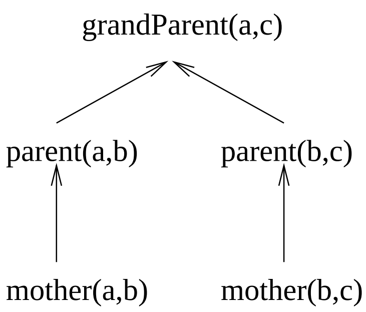

Fig. 1. A derivation of $mother(a, b)$, $mother(b, c) \rightarrow grandParent(a, c)$ from H.

Let P be a set of predicate symbols, and T be a set of terms. Let L be a set of atoms defined using P and T. Let $\mathcal{H}$ be a set of Horn programs using atoms in L only. If k is an integer, then P_k is a subset of P containing only those predicate symbols of arity k or less. Further, L_k is a set of atoms defined using P_k and T, and $\mathcal{H}_k$ is a set of Horn programs using atoms in L_k only. In the following three definitions, we describe a class of Horn programs AH_k for which minimal models are of polynomial size.

Definition 7 (Arimura, 1997). Let $\Sigma \in \mathcal{H}$. Then a binary relation **supported by** (denoted, $\succ$) over atoms in L w.r.t. Σ is such that (1) for all $p \rightarrow l \in \Sigma$, and for all $l_1 \in p$, $l \succ l_1$; (2) for all $l_1, l_2 \in L$ and every substitution θ, if $l_1 \succ l_2$, then $l_1\theta \succ l_2\theta$; and (3) if $l_1 \succ l_2$ and $l_2 \succ l_3$ then $l_1 \succ l_3$.

Definition 8. A Horn program Σ is **acyclic** over L if the relation $\succ$ over L w.r.t. Σ is terminating; i.e., for any $l \in L$, there is no infinite decreasing sequence $l \succ l_1 \succ \ldots$.

In the last example, H is acyclic because $grandParent(x, y) \succ parent(x, y) \succ mother(x, y)$ and there is no cycle formed by the $\succ$ relation.

Following Khardon (1998), we call a definite clause a *non-generative clause* if the set of terms in its consequent are a subset of the set of terms and subterms in its antecedent.

Definition 9. If k is a constant, we define a Horn program $\Sigma \in \mathcal{H}_k$ to be in the class AH_k, if Σ is acyclic over L_k, and each clause is either non-generative or has an empty antecedent.

Definition 10. Let $a \rightarrow b$ be a clause in a Horn program Σ, and $p \rightarrow q$ be a clause. Then, $a \rightarrow b$ is a **target clause** in Σ of $p \rightarrow q$ iff $a \rightarrow b \succeq p \rightarrow q$, i.e., for a substitution θ, $a\theta \subseteq p$, $b\theta = q$. We call $p \rightarrow q$ a **hypothesis clause** of $a \rightarrow b$.

Definition 11. For an antecedent p, q' is a **prime consequent** of p wrt Σ if $\Sigma \models p \rightarrow q'$, $q' \notin p$, and there is no $l \in L$ such that $q' \succ l$, $\Sigma \models p \rightarrow l$ and $l \notin p$.

In the last example, $parent(a, b)$ is a prime consequent of $mother(a, b)$, $mother(b, c)$, but $grandParent(a, c)$ is not—since $parent(a, b) \succ grandParent(a, c)$.

3 Learning Horn Programs

In this section, we show that a subclass of AH_k is exactly learnable, using the exact learning model (Angluin, 1988), in entailment setting. Henceforth, $\Sigma \in AH_k$ denotes a target Horn program.

3.1 The Learning Model

In learning from entailment, an example is a Horn clause. An example $p \to q$ is a positive example of Σ if $\Sigma \models p \to q$; negative, otherwise. An *entailment query* takes as input an example $(p \to q)$, and outputs *yes* if it is a positive example of Σ ($\Sigma \models p \to q$), and *no* otherwise. An *equivalence query* takes as input a Horn program H and outputs *yes* if H and Σ contain (entail) exactly the same Horn clauses; otherwise, returns a *counterexample* that is in (entailed by) exactly one of H and Σ. A *derivation-order query*, $\succ$, takes as input two atoms l_1 and l_2 in L and outputs *yes* if $l_1 \succ l_2$, and *no* otherwise. An algorithm *exactly learns* a Horn program Σ in AH_k in polynomial time from equivalence, entailment, and derivation-order ($\succ$) queries if and only if it runs in time polynomial in the size of Σ and in the size of the largest counterexample, and outputs a Horn program in AH_k such that equivalence query answers *yes*.

3.2 The Learning Algorithm

In this section, we describe the learning algorithm, PLearn, shown in Figure 2. PLearn always maintains a hypothesis H which is entailed by the target, so that every instance of H is also an instance of Σ and all counterexamples are positive.

Suppose that a counterexample $p \to q$ is given to the learner—see Figure 2. Every such counterexample has a derivation from the target theory, Σ. Since this derivation is not possible from the current hypothesis H, there is some clause used in the derivation that has not been learned with sufficient generality. The algorithm tries to identify the antecedent literals of such a clause, $c*$, in the target by expanding the derivation graph from its leaves in p toward the goal using the clauses in H. In other words, PLearn computes the minimal model (p'_f) of H implied by p ("closure" or "saturation") by forward chaining (line 4). To identify the consequent of $c*$, also called the "prime consequent" of p'_f, PLearn calls PrimeCons in line 5. PrimeCons finds the prime consequent of p'_f by tracing the "supported-by" chain starting from q for a literal q_f not in p'_f, but is directly supported by some of the literals in p'_f (lines 13–18). In line 6, PLearn makes use of Reduce to trim away "irrelevant" literals from the antecedent p'_f to form a new clause $p_f \to q_f$ that is also a counterexample to the hypothesis and is subsumed by a single target clause—see Lemmas 20,12,13.

PLearn combines $p_f \to q_f$ with an "appropriate" clause $p_i \to q_i$ in H using lgg (lines 7–9). It uses the entailment query to find an appropriate hypothesis clause by checking if the result of lgg is implied by the target (line 7). If no such clause exists in H, $p_f \to q_f$ is appended to H as a new clause (line 10).

One problem with this approach is that the size of the lgg is a product of the sizes of its two arguments. This causes the size of a hypothesis clause to grow exponentially in the number of examples combined with it in the worst case. To avoid this, the antecedent literals of the clause after lgg are again trimmed using Reduce so that the size of the resulting clause is bounded, while it is still subsumed by the target clause (lines 19–25). The result of Reduce then replaces the original hypothesis clause $p_i \rightarrow q_i$ it is derived from (line 9). After this step, only the antecedents of the target clause and some of their consequents remain in the resulting hypothesis clause—see Lemma 16. This process repeats until the hypothesis H is equivalent to Σ. The algorithm works for unit clauses (which have empty antecedents) without change.

PLearn
Given equivalence, entailment queries, and the derivation order $\succ$
outputs a Horn program H such that **equivalent?**(H, Σ) is *Yes.*

(1) $H = \{\}$ `/* empty hypothesis-clauses set */`
(2) **while** not **equivalent?**(H, Σ) **do** {
(3)   Let $p \rightarrow q$ be the counterexample returned
(4) $p'_f = \{l : H \models (p \rightarrow l)\}$ `/* forward chaining */`
(5) $q_f =$ PrimeCons$(p'_f \rightarrow q)$
(6) $p_f \rightarrow q_f =$ Reduce$(p'_f \rightarrow q_f)$
(7)   **if** $\exists p_i \rightarrow q_i \in H$ such that $\Sigma \models p_g \rightarrow q_g$,
(8)   where $p_g \rightarrow q_g = lgg(p_i \rightarrow q_i, p_f \rightarrow q_f)$
(9)   **then** replace first such $p_i \rightarrow q_i$ by Reduce$(p_g \rightarrow q_g)$
(10)   **else** append $p_f \rightarrow q_f$ to H
(11) } `/* while */`
(12) **return** H

PrimeCons$(p \rightarrow q)$ `/* finds prime consequents */`
(13) Let L be the set of all possible literals having only those terms that are in p
(14) $q' = q$;
(15) $L' = \{l : l \in L - p$ and $\Sigma \models p \rightarrow l\}$
(16) **while** $\exists l \in L'$ such that $q' \succ l$
(17) $q' = l$;
(18) **return** q'

Reduce$(p \rightarrow q)$ `/* trims irrelevant literals */`
(19) $p' = p$
(20) **repeat**
(21)   **for each** literal l in p' in sequence **do**
(22)   **if** $\Sigma \not\models (p' - \{l\}) \rightarrow l$ and $\Sigma \models (p' - \{l\}) \rightarrow q$
(23)   **then** $p' = p' - \{l\}$
(24) **until** there is no change to p'
(25) **return** $p' \rightarrow q$

Fig. 2. PLearn Algorithm

3.3 An Example

As an example to see how PLearn works, consider $\Sigma = \{l_1(f(x)), l_2(x), l_3(x) \rightarrow l_4(x); l_1(f(x)), l_2(x) \rightarrow l_5(x); l_4(x), l_5(x) \rightarrow l_7(x)\}$ where f is a function symbol. Suppose $H = \{l_1(f(c)), l_2(c) \rightarrow l_5(c)\}$. We adopt the convention that the letters such as a, b, c, etc. at the beginning of the alphabet are constants and the letters at the end of the alphabet such as x, y, z, etc. are variables. Let the counterexample be $l_1(f(d)), l_2(d), l_3(d) \rightarrow l_7(d)$. In step 4, it does not change. In PrimeCons, since $l_7(d) \succ l_4(d)$ and $l_7(d) \succ l_5(d)$, $l_7(d)$ is not a prime consequent, but any one of $l_4(d)$ and $l_5(d)$ is. Suppose PrimeCons returns $l_5(d)$. Reduce eliminates $l_3(d)$ from the antecedent, because $\Sigma \models l_1(f(d)), l_2(d) \rightarrow l_5(d)$, and $\Sigma \not\models l_1(f(d)), l_2(d) \rightarrow l_3(d)$. Thus, $p_f \rightarrow q_f = l_1(f(d)), l_2(d) \rightarrow l_5(d)$. Combining this with the clause in H, we obtain $p_g \rightarrow q_g = l_1(f(x)), l_2(x) \rightarrow l_5(x)$ is entailed by Σ, new H is $\{l_1(f(x)), l_2(x) \rightarrow l_5(x)\}$.

Suppose the next counterexample is $l_1(f(c)), l_2(c), l_3(c) \rightarrow l_7(c)$. Then, $q_f = l_4(c)$, and $p'_f = \{l_1(f(c)), l_2(c), l_3(c), l_5(c)\}$. $p_f \rightarrow q_f = l_1(f(c)), l_2(c), l_3(c), l_5(c) \rightarrow l_4(c)$, since Reduce cannot remove $l_5(c)$, because it is implied by the other literals wrt Σ (line 22). The modified counterexample $p_f \rightarrow q_f$ cannot be combined with the clause in H, because the resultant $p_g \rightarrow q_g$ after lgg, $l_1(f(x)), l_2(x) \rightarrow$, is not entailed by Σ. Hence, it is appended to H to make $H = \{l_1(f(x)), l_2(x) \rightarrow l_5(x); l_1(f(c)), l_2(c), l_3(c), l_5(c) \rightarrow l_4(c)\}$.

Suppose the next counterexample is again $l_1(f(c)), l_2(c), l_3(c) \rightarrow l_7(c)$. After line 4, $p'_f = l_1(f(c)), l_2(c), l_3(c), l_5(c), l_4(c)$. q_f now is $l_7(c)$, because it is a prime consequent of p'_f. After Reduce, $p_f = l_5(c), l_4(c)$. $p_f \rightarrow q_f$ cannot be combined with the clauses in H, because the resultant lgg's are not entailed by Σ. Again, $p_f \rightarrow q_f$ is added to H. This process continues until H and Σ are equivalent.

To bring out the nuances in Reduce, let us revisit the last part of the previous example. Consider the input $l_5(c), l_2(c), l_3(c), l_1(f(c)), l_4(c) \rightarrow l_7(c)$ to Reduce. Although $\Sigma \models l_1(f(c)), l_2(c), l_3(c) \rightarrow l_7(c)$, since $\Sigma \models l_1(f(c)), l_2(c), l_3(c) \rightarrow l_4(c)$ and $\Sigma \models l_1(f(c)), l_2(c) \rightarrow l_5(c)$, the literal $l_5(c)$ cannot be removed. This is because $l_5(c)$ is implied by the other literals $(l_1(f(c)), l_2(c))$ wrt Σ. The order in which the literals are removed in Reduce follows the derivation order: if $l_i \succ l_j$, if at all l_i is removed, it is removed after l_j is removed. This can be intuitively imagined in the following way. Consider a derivation tree for a counterexample, with the consequent literal on top and the antecedent literals at the bottom. The above process trims off the literals bottom-up in the tree up to the appropriate level, so that the resulting clause is subsumed by some clause in the target. In the above case, if Reduce removes $l_5(c)$ and leaves over $l_1(f(c)), l_2(c)$, the resulting clause $(l_1(f(c)), l_2(c), l_3(c), l_4(c) \rightarrow l_7(c))$ is not subsumed by any clause in Σ.

However, this means that Reduce leaves over literals which are implied by the remaining literals, i.e., l cannot be removed from p' if $\Sigma \models (p' - \{l\}) \rightarrow l$ (line 22). Removing such literals could result in hypothesis clauses which are not subsumed by any target clause, as the following example illustrates. Let Σ be $\{l_1(a) \rightarrow l_2(a);\ l_1(x), l_2(x) \rightarrow l_3(x)\}$. Suppose the first counterexample is $l_1(a), l_2(a) \rightarrow l_3(a)$. Hence $p'_f = \{l_1(a), l_2(a)\}$ and $q_f = l_3(a)$ in line 6. If Reduce were to remove $l_2(a)$ from p'_f because $\Sigma \models l_1(a) \rightarrow l_3(a)$, it ends up with a clause that is not subsumed by any target clause. We would like to prevent such redundant hypothesis clauses so that their number is not too high compared to the number of target clauses. (This argument is formalized in Lemmas 17, 18 and 19.)

3.4 Learnability of AH_k

In this section, we prove that PLearn algorithm in Figure 2 exactly learns a subclass of AH_k for which subsumption is of polynomial-time complexity. The plan of the proof is as follows: Through a series of lemmas, we first establish that every hypothesis clause learned has a target clause (Lemma 17). We then show that every target clause has at most one hypothesis clause (Lemma 19). Together, these two lemmas establish that the number of hypothesis clauses is bounded by the number of target clauses. We use this fact and the bounds of the sizes on the hypothesis clauses (established in Lemma 16) to show that PLearn learns successfully in polynomial time (Theorems 21 and 24). We then define a specific hypothesis class that obeys the conditions of these theorems and prove that this class is learnable (Theorem 26).

Lemmas 12 and 13 show that PrimeCons with the input $p \rightarrow q$ finds a (prime) consequent q' of p such that $p \rightarrow q'$ is subsumed by a clause in Σ.

Lemma 12. *Let $p \rightarrow q$ be the input and q' be the output of* PrimeCons*. Assume that $q \notin p$ and $\Sigma \models p \rightarrow q$. Then, (1)* PrimeCons *terminates; (2) q' is a prime consequent of p wrt Σ.*

Proof. (1) Since Σ is acyclic, there is a terminating sequence $q \succ l_1 \succ l_2 \ldots$. Since the loop of lines 16–17 can only iterate as many times as the length of the sequence, PrimeCons terminates.
(2) q' is such that $\Sigma \models p \rightarrow q'$, and $q' \notin p$ (by lines 15–17). Since q' is as in line 17 in the iteration immediately prior to the terminating iteration of lines 16–17, there is no l such that $q' \succ l$, $\Sigma \models p \rightarrow l$ and $l \notin p$. Thus, q' is a prime consequent of p wrt Σ. □

Lemma 13. *If q' is a prime consequent of p wrt Σ, then there is a clause $C \in \Sigma$ such that $C \succeq p \rightarrow q'$.*

Proof. Assume that q' is a prime consequent of p wrt Σ. Consider a derivation G of $p \rightarrow q'$ in Σ. Let $(l_1, q'), \ldots, (l_d, q')$ be the only arcs of G that terminate at q'. This implies that $q' \succ l_i$ for all $l_i \in \{l_1, \ldots, l_d\}$. It must be that every l_i is in p; otherwise, there is an l (viz. l_i) such that $q' \succ l$, $\Sigma \models p \rightarrow l$ and $l \notin p$—contradicting the assumption that q' is a prime consequent. Thus, $\{l_1, \ldots, l_d\} \subseteq p$. But, $l_1, \ldots, l_d \rightarrow q' = C\theta$ for some clause $C \in H$ and a substitution θ, following the definition of derivation. Thus, $C\theta \subseteq p \rightarrow q'$, implying that $C \succeq p \rightarrow q'$. □

The following definition and Lemmas 15 and 16 help show that Reduce, given a clause $p \rightarrow q$ as input, removes irrelevant literals from antecedent p, while maintaining q as a consequent.

Definition 14. If a is a conjunction, **closure** of a with respect to Σ, denoted by κ_a, is defined as $\{l | \Sigma \models (a \rightarrow l)\}$.

Lemma 15. *If q is a prime consequent of p and $p' \rightarrow q =$ Reduce$(p \rightarrow q)$, then q is a prime consequent of p' also.*

Proof. Because q is a prime consequent of p and $p' \subseteq p$, any literal other than the ones in $p - p'$, cannot be prime consequents of p'. By lines 22–23, only those literals l that are not supported by p' are removed. In which case, no literal l in $p - p'$, can be such that $\Sigma \models p' \rightarrow l$. Hence, q is a prime consequent of p' as well. $\square$

Lemma 16. *If the input $p \rightarrow q$ to* Reduce *is s.t. q is a prime consequent of p wrt Σ, then the output $p' \rightarrow q$ is such that $p' \subseteq \kappa_{a\theta}$ where $a \rightarrow b$ is a clause in Σ and $a\theta \subseteq p'$ and $b\theta = q$.*

Proof. Since q is a prime consequent of p, by Lemma 15, q is a prime consequent of p' also. Then, by Lemma 13, there is a clause $a \rightarrow b \in \Sigma$, and a θ such that $a\theta \subseteq p'$ and $b\theta = q$. We now show that $p' \subseteq \kappa_{a\theta}$. Assume that there exists a literal in $p' - \kappa_{a\theta}$. Let $l \in p' - \kappa_{a\theta}$ be a least such literal so that there is no literal l' in $p' - \kappa_{a\theta}$ such that $l \succ l'$. Such a literal must exist, because Σ is acyclic. There are two reasons for l to remain in $p' - \kappa_{a\theta}$: either (a) $\Sigma \not\models (p' - \{l\}) \rightarrow q$ or (b) $\Sigma \models (p' - \{l\}) \rightarrow l$. We disprove both the cases: (a) Since $a\theta \subseteq p'$, and l is not in $\kappa_{a\theta}$ and thus not in $a\theta$, $a\theta \subseteq (p' - \{l\})$. Therefore, $\Sigma \models (p' - \{l\}) \rightarrow q$. (b) The only other reason why l remains in p' is that $\Sigma \models (p' - \{l\}) \rightarrow l$. That means that $p' - \{l\}$ contains literals that imply l. There must be at least one such literal in p' that is not in $\kappa_{a\theta}$, or else $l \in \kappa_{a\theta}$, contradicting $l \in p' - \kappa_{a\theta}$. But then $p' - \kappa_{a\theta}$ contains literals l' such that $l \succ l'$, which contradicts the statement that there is no such l'. Thus, we disprove both the possibilities. Hence, $p' \subseteq \kappa_{a\theta}$. $\square$

Lemmas 17, 18 and 19, below, show that PLearn only maintains right clauses in H.

Lemma 17. *Every clause $p_i \rightarrow q_i \in H$ has a target clause.*

Proof. We first show that each $p_i \rightarrow q_i \in H$ is such that q_i is a prime consequent of p_i. Then, by Lemma 13, $p_i \rightarrow q_i$ has a clause $C \in \Sigma$ such that $C \succeq p_i \rightarrow q_i$.

We show that q_i is a prime consequent of p_i by induction on the number of times a clause at position i in H is updated. It is first introduced by line 10. By Lemmas 12 and 15, q_f is a prime consequent of p_f. This proves the base case. The other way a clause becomes a hypothesis clause is by line 9. The clause at position i in H ($p_i \rightarrow q_i$) is updated by line 9. As inductive hypothesis, assume that each $p_i \rightarrow q_i$ in H is such that q_i is a prime consequent of p_i, at the beginning of an iteration of the loop of lines 2–11 when position i in H is updated. Consider $p_g \rightarrow q_g = lgg(p_i \rightarrow q_i, p_f \rightarrow q_f)$. Suppose q_g is not a prime consequent of p_g, but q'_g such that $q_g \succ q'_g$ is. Let θ_f and θ_i be substitutions such that $p_g\theta_f \subseteq p_f$, $q_g\theta_f = q_f$, $p_g\theta_i \subseteq p_i$, and $q_g\theta_i = q_i$. Let $q'_f = q'_g\theta_f$ and $q'_i = q'_g\theta_i$. Since $q_g \succ q'_g$, by the definition of $\succ$ order, $q_f \succ q'_f$ and $q_i \succ q'_i$. Since q_f is a prime consequent of p_f, q'_f must be in p_f. Similarly, q'_i must be in q_i. Therefore, $lgg(q'_i, q'_f) = q'_g$ must be in p_g, contradicting the assumption that q'_g is a prime consequent of p_g. Hence, q_g is a prime consequent of p_g. By Lemma 15 if $p_i \rightarrow q_i = \mathsf{Reduce}(p_g \rightarrow q_g)$, then q_i is a prime consequent of p_i. So by Lemma 13, $p_i \rightarrow q_i$ has a target clause. $\square$

Lemma 18. *If* PLearn *combines a modified counterexample $p_f \rightarrow q_f$ with a clause $p_i \rightarrow q_i \in H$, then there is a target clause C s.t. $C \succeq p_f \rightarrow q_f$ and*

$C \succeq p_i \rightarrow q_i$. Further, there is no C' s.t. $C' \succeq p_j \rightarrow q_j$ and $C' \succeq p_f \rightarrow q_f$, for any $j < i$.

Proof. PLearn combines $p_f \rightarrow q_f$ with $p_i \rightarrow q_i$ only if $\Sigma \models lgg(p_i \rightarrow q_i, p_f \rightarrow q_f)$. By Lemma 17, q_g is a prime consequent of p_g where $p_g \rightarrow q_g = lgg(p_i \rightarrow q_i, p_f \rightarrow q_f)$. By Lemma 13, there is a $C \in \Sigma$ such that $C \succeq p_g \rightarrow q_g$. Hence, $C \succeq p_i \rightarrow q_i$ and $C \succeq p_f \rightarrow q_f$. Since $p_f \rightarrow q_f$ is combined with $p_i \rightarrow q_i$, for any $j < i$, $\Sigma \not\models lgg(p_j \rightarrow q_j, p_f \rightarrow q_f)$. Therefore, there is no C' s.t. $C' \succeq lgg(p_j \rightarrow q_j, p_f \rightarrow q_f)$. Thus, there is no C' s.t. $C' \succeq p_j \rightarrow q_j$ and $C' \succeq p_f \rightarrow q_f$. □

Lemma 19. *Every clause $C \in \Sigma$ has at most one hypothesis clause.*

Proof. First, we show that any new hypothesis clause added to H has a target clause distinct from the target clauses of the other hypothesis clauses in H. Next, we show that if two hypothesis clauses do not have common target clauses at the beginning of an iteration of the loop of lines 2–11, then they still have distinct target clauses at the end of the iteration.

When $p_f \rightarrow q_f$ is added to H, by Lemma 18, for any clause H_i in H, there is no $C \in \Sigma$ such that $C \succeq H_i$ and $C \succeq p_f \rightarrow q_f$. Therefore, $p_f \rightarrow q_f$, a new clause added to H, has a target clause distinct from the target clauses of the other hypothesis clauses then in H. Next, at most one of H_i and H_j can change in an iteration of the loop. If neither changes, we are done with the proof. Suppose that H_i changes, without loss of generality. Let C be any target clause of H_j. Assume that H_i and H_j do not have a common target clause at the beginning of an iteration. Hence, C is not a target clause of H_i. That is, $C \not\succeq H_i$. Let e be the counterexample for the current iteration. We first show that $lgg(H_i, e)$ does not have C as a target clause. Since $C \not\succeq H_i$, $C \not\succeq lgg(H_i, e)$. Therefore, C is not a target clause of $lgg(H_i, e)$. Let $lgg(H_i, e)$ be $p_g \rightarrow q_g$, and C be $a \rightarrow b$. Hence, for every θ, either $a\theta \not\subseteq p_g$ or $b\theta \neq q_g$. If $a\theta \not\subseteq p_g$, $a\theta$ is not a subset of any subset of p_g. Since Reduce outputs a clause with a subset of p_g as the antecedent and q_g as the consequent, $C \not\succeq \text{Reduce}(lgg(H_i, e))$. Therefore, H_j and the new clause in position i, $\text{Reduce}(lgg(H_i, e))$, do not have a common target clause even at the end of the iteration. □

The following lemma shows that even after the modifications due to Prime-Cons and Reduce counterexample remains a counterexample.

Lemma 20. *$p_f \rightarrow q_f$ as in line 6 of* PLearn *is a positive counterexample.*

Proof. First, we show that every counterexample $p \rightarrow q$, as in line 3, is a positive counterexample. Then, we argue that $p'_f \rightarrow q_f$ (lines 4 and 5) is also a positive counterexample. Finally, we show that $p_f \rightarrow q_f$ (line 6) is a positive counterexample.

Since, by Lemma 17, for every $H_i \in H$, there is a clause $C \in \Sigma$ such that $C \succeq H_i$, $\Sigma \models H$. Therefore, $p \rightarrow q$, as in line 3, is a positive counterexample. Since $p \subseteq p'_f$, $\Sigma \models p'_f \rightarrow q$. Since p'_f contains all and only those literals l such that $H \models p \rightarrow l$, for any literal $l' \notin p'_f$, $H \not\models p'_f \rightarrow l'$. Since q_f (by lines 5 and 15) is not in p'_f, $H \not\models p'_f \rightarrow q_f$. By line 15, $\Sigma \models p'_f \rightarrow q_f$. Therefore, $p'_f \rightarrow q_f$ is also a positive counterexample. Finally, since $p_f \subseteq p'_f$, $H \not\models p_f \to q_f$. By lines 6 and 22, $\Sigma \models p_f \to q_f$. Thus, $p_f \to q_f$ is a positive counterexample. □

Finally, Theorem 21 shows that PLearn exactly learns AH_k when forward chaining using H is of polynomial-time complexity. Theorem 24 identifies conditions on Σ such that PLearn returns an H for which time complexity of forward chaining is polynomial.

Theorem 21. PLearn *exactly learns AH_k with equivalence, $\succ$, and entailment queries, provided that determining $H \models p \to l$ is polynomial in the sizes of H and p.*

Proof. By Lemma 20, $p_f \to q_f$ is a positive counterexample. For each counterexample, either a new antecedent is added (line 10) or an existing antecedent is replaced (line 9). In the latter case, the replaced clause $p_i \to q_i$ must be subsumed by the replacing clause $p' \to q_g$, since both lgg and Reduce generalize the original clause by turning constants to variables and dropping literals. On the other hand, the replaced clause must not subsume (and hence be different from) the replacing clause $p' \to q_g = \text{Reduce}(p_g \to q_g)$. If not, that is if $p_i \to q_i \succeq p' \to q_g$, since $p' \to q_g \succeq p_g \to q_g \succeq p_f \to q_f$, $p_i \to q_i \succeq p_f \to q_f$. Since $p_i \to q_i \in H$, $H \models p_f \to q_f$—thus contradicting that $p_f \to q_f$ was a counterexample of H. Hence, the replacement at a position in H changes the clause at that position. The minimum change there can be is either a variablization of a constant or a removal of a literal.

Let n be the number of clauses, and s be the number of distinct predicate symbols in Σ. Further, let the maximum number of terms in any clause be t, and in any counterexample be t_e.

The maximum possible number of literals there can be using t terms is at most st^k. Hence, the maximum number of literals in κ_a, and therefore, by Lemmas 16 and 17, in each clause is at most st^k. This includes all literals and their variablized versions. Hence, we can consider variablization as removing a literal. Thus, we need at most st^k counterexamples for each clause. (This includes one base counterexample to introduce a clause into H.) By Lemmas 17 and 19, there are at most n clauses in H. Hence, we need at most nst^k counterexamples or equivalence queries. A call to PrimeCons from line 5 takes at most st_e^k entailment queries, because the literals we need to try as possible consequents are all in L, and $|L| \leq st_e^k$. PrimeCons is called once for each of the counterexamples.

For each of the nst^k counterexamples, the condition in line 7 is tested at most n times, which needs at most n entailment queries. Reduce is called with the argument $p'_f \to q_f$ once for each of the counterexamples, and with the arguments $p_g \to q_g$ for at most nst^k counterexamples. In $\text{Reduce}(p \to q)$, in $|p|$ iterations of the loop of lines 21–23, at least one literal is removed. So, this loop can be tried at most $|p|$ times. Each iteration of the loop of lines 21–23 takes two entailment queries. Therefore, $\text{Reduce}(p \to q)$ needs at most $|p|(|p|+1)$ entailment queries. Hence, $\text{Reduce}(p'_f \to q_f)$ needs at most $n_f = st_e^k(st_e^k + 1)$ entailment queries. Since $p_i \to q_i$ and $p_f \to q_f$ are outputs of Reduce, the maximum possible number of literals in $p_g \to q_g = lgg(p_i \to q_i, p_f \to q_f)$ is at most s^2t^{2k}. Hence, $\text{Reduce}(p_g \to q_g)$ needs at most $n_g = s^2t^{2k}(s^2t^{2k} + 1)$ entailment queries. Thus, the total number of entailment queries is at most $nst^k(st_e^k + n + n_f + n_g)$.

If determining $H \models (p \rightarrow l)$ takes $\mathcal{P}(n, l, t_e)$ time where $\mathcal{P}$ is a polynomial, then line 4 takes at most $st_e^k \cdot \mathcal{P}(n, l, t_e)$ time. In the rest, the number of entailment queries dominates the time. Hence, the time taken by PLearn is polynomial in n, s, l, v, t, and t_e. □

Definition 22. Let $p \rightarrow q$ be a Horn clause. $p' \rightarrow q$ is called its **antecedent expansion** if $p \subseteq p'$ and p' contains only those variables in p. A class $\mathcal{C}$ of Horn sentences is closed under antecedent expansion, if every Horn sentence obtained by selecting a subset of its Horn clauses and replacing them with their antecedent expansions is also in C.

Definition 23. A **subsumption algorithm** takes a clause $a \rightarrow b$, a conjunction of literals p, and a ground substitution θ for the variables in b, and returns true if and only if $a\theta \succeq p$.

Theorem 24. PLearn *exactly learns a subclass $\mathcal{C}$ of AH_k with equivalence, $\succ$, and entailment queries, provided that (a) $\mathcal{C}$ is closed under substitution and antecedent expansion and (b) the clauses $a \rightarrow b$ of the target concepts in $\mathcal{C}$ have a polynomial-time subsumption algorithm.*

Proof. By Lemma 16, each clause $p_i \rightarrow q_i \in H$ in PLearn has a target clause $a \rightarrow b$ and a substitution θ such that $a\theta \subseteq p_i \subseteq \kappa_{a\theta}$. Since the target class is closed under substitution and antecedent expansion, the hypothesis clauses have a polynomial-time subsumption algorithm. Hence, the forward-chaining step of computing the consequents of p in line 4 of PLearn can be done in polynomial time by repeatedly checking for a hypothesis clause $a \rightarrow b$ whose antecedent subsumes p after a substitution θ of the variables in b, and adding $b\theta$ to p. Hence, by the previous theorem, PLearn exactly learns $\mathcal{C}$. □

The following definition and theorem identify some syntactic restrictions on AH_k such that the resulting subclass satisfies the conditions of the previous theorem.

Definition 25. Let p be a set of literals. A Horn clause $l_1, \ldots, l_n \rightarrow q$ is ***i*-determinate** w.r.t. p iff there exists an ordering $l_{o_1}, \ldots, l_{o_n}$ of $l_1, \ldots, l_n$ such that for every $i < j \leq n$ and every substitution θ such that $(l_{o_1}, \ldots, l_{o_{j-1}} \rightarrow q)\theta$ is ground and $\{l_{o_1}, \ldots, l_{o_{j-1}}\}\theta \subseteq p$, there is at most one substitution α for the variables in $l_{o_j}\theta$ such that $l_{o_j}\theta\alpha$ is ground and is in p.[2] We call such an ordering of the literals in the clause an ***i*-determinate ordering** w.r.t. p. A Horn program is i-determinate w.r.t. p iff each of the clauses in the program is i-determinate w.r.t. p.

Theorem 26. *The class of i-determinate Horn programs in AH_k, denoted as $iDetAH_k$, is exactly learnable with equivalence, $\succ$, and entailment queries.*

[2] This definition strictly generalizes the standard definition of determinacy (Muggleton & Feng, 1990), in that a Horn clause (program) is determinate w.r.t. a set of literals p when it is 0-determinate w.r.t. p. i-determinacy should not be confused with ij-determinacy, or constant-depth fixed-arity determinacy, which is more restricted than determinacy.

Proof. First we show that $iDetAH_k$ is closed under substitution and antecedent expansion. Consider a target clause $(l_1, \ldots, l_n \to q)$ for a target program in $iDetAH_k$, whose antecedent literals are sorted in the determinate order. Let $(l_1, \ldots, l_n, l_{n+1}, \ldots, l_m \to q)\beta$ be the target clause after antecedent expansion and substitution. We want to show the new clause to be i-determinate.

For every set of literals p, substitution θ, and j such that $i < j \leq m$ and $(l_1, \ldots, l_{j-1})\beta\theta \subseteq p$ is ground, there is a substitution γ which is equivalent to applying β and θ one after another so that $(l_1, \ldots, l_{j-1})\beta\theta = (l_1, \ldots, l_{j-1})\gamma$ and $l_j\beta\theta = l_j\gamma$ for any l_j. Since the target clause satisfies i-determinacy, there must be at most a single ground substitution α for $l_j\gamma$, $j \leq n$, so that $l_j\gamma\alpha \in p$, which means that this is true for $l_j\beta\theta$ as well. Since the literals from l_{n+1} through l_m do not have any variables not already in l_1 through l_n, there is at most a single ground substitution for them as well. Hence, $(l_1, \ldots, l_m \to q)\beta$ is also i-determinate.

Now we show that the clauses of the programs in $iDetAH_k$ have a polynomial-time subsumption algorithm. Given a set of literals p and a clause $l_1, \ldots, l_n \to q$ (whose literals have an unknown determinate ordering), consider all possible subsets of $\{l_1, \ldots, l_n\}$ of size i and less. Note that there are at most $O(n^i)$ such subsets. For each such subset, instantiate all the ki variables in that subset in all possible ways. If the total number of terms in p and Σ is t, this gives us t^{ki} different substitutions. For each such substitution, there is at most one substitution for the remaining literals in the clause. The order in which the remaining literals have to be substituted can be determined by sequential search—apply the current substitution to each literal and pick the one that only allows one possible substitution for its remaining variables. This can be done in $O(n^2|p|)$ time. If the antecedent $l_1, \ldots, l_n$ subsumes p, then one of the considered subsets should yield a successful match. Hence, the total time for the algorithm is bounded by $O(n^i t^{ki} n^2|p|)$, which is polynomial in all variables except k and i which are assumed to be constants.

Since the class $iDetAH_k$ satisfies the two conditions required by Theorem 24 for PLearn to be successful, the result follows. □

4 Discussion and Conclusions

In this paper, we have shown the learnability of certain subclasses of acyclic k-ary Horn programs. More specifically i-determinate Horn programs in AH_k, are exactly learnable with equivalence and entailment queries. Unlike the work of Page (1993) and Arimura (1997), the programs we considered allow local variables in the antecedents. However, the clauses must be non-generative in that the set of terms and variables that occur in the head of the clause must be a subset of those that occur in the body of the clause. This is needed to constrain the forward-chaining inference step to finish in polynomial-time, which could otherwise become unbounded. It appears that simultaneously removing both the non-generative and simplicity restrictions could be difficult when functions are present, due to the unbounded nature of inference in that case.

Learning from entailment and *learning from interpretations* are two of the standard settings for first-order learning (De Raedt, 1997). In learning from interpretations, the learner is given a positive (or negative) interpretation for which the Horn sentence is true (or false). Interpretations can be partial in that the truth values of some ground atoms may be left unspecified. When membership queries are available, learning from entailment and learning from interpretations are equivalent for Horn programs. Hence we can use PLearn to learn from (negative) interpretations as follows. Given a negative interpretation, "minimize" it by removing the negative literals from it and asking membership queries. Since every negative interpretation must violate some Horn clause, this yields an interpretation with a set of positive literals $l_1, \ldots, l_n$ and at most one negative literal q_i. We can convert this into a positive counterexample for PLearn: $l_1 \wedge \ldots \wedge l_n \rightarrow q_i$. Similarly, if PLearn asks an entailment membership query on some clause, say, $l_1 \wedge \ldots \wedge l_n \rightarrow q_i$, we can turn that into a membership query on the interpretation $l_1, \ldots, l_n, \neg q_i$ after substituting a unique skolem constant for each variable in the clause. The answer to the entailment query is *true* iff the answer to the membership query is *false*.

One limitation of our algorithm is that it assumes that the *supported by* relation, $\succ$, is given. While this is a reasonable assumption in some planning domains, where it is known which goals occur as subgoals of which, it is desirable to learn this relation. Unfortunately, this seems difficult due to a number of problems. One of the main difficulties is that it is sometimes not possible to determine which, of the set of consequents of an antecedent, is the prime consequent. For example, consider the target $\Sigma : \{l_1(x) \wedge l_2(x) \rightarrow l_3(x);\, l_1(x) \wedge l_3(x) \rightarrow l_4(x)\}$. Given the counterexample $l_1(c) \wedge l_2(c) \rightarrow l_4(c)$, the literal $l_4(c)$ is not a correct consequent, but $l_3(c)$ is. Although Lemma 13 says that prime consequent is a right consequent to choose, without knowing the order it is not clear how to identify it. Learning all possible clauses while maintaining all consequents also does not seem to work, resulting in spurious matches between some of these redundant clauses and counterexamples in some cases.

As shown in (Reddy & Tadepalli, 1997b), Horn programs can be used to express goal-decomposition rules (d-rules) for planning using the situation-calculus formalism. We believe that the algorithm discussed here and its extensions can be applied to learn d-rules, which is an important problem in speedup learning. d-rules are a special case of hierarchical task networks or HTNs (Erol, Hendler, & Nau, 1994)—in that HTNs allow partial ordering over subgoals and non-codesignation constraints over variables whereas d-rules do not. Nevertheless, it can be shown that HTNs can be expressed as Horn programs.

Acknowledgments We gratefully acknowledge the support of ONR under grant # N00014-95-1-0557 and NSF under grant # IRI-9520243. We thank Roni Khardon and the anonymous reviewers of ICML-98 and ILP-98 for their insightful comments.

References

Angluin, D. (1988). Queries and concept learning. *Machine Learning*, *2*, 319–342.

Arimura, H. (1997). Learning acyclic first-order horn sentences from entailment. In *Proceedings of the Eigth International Workshop on Algorithmic Learning Theory*. Ohmsha/Springer-Verlag.

Cohen, W. (1995a). Pac-learning non-recursive prolog clauses. *Artificial Intelligence*, *79*(1), 1–38.

Cohen, W. (1995b). Pac-learning recursive logic programs: efficient algorithms. *Jl. of AI Research*, *2*, 500–539.

De Raedt, L. (1997). Logical settings for concept learning. *Artificial Intelligence*, *95*(1), 187–201.

Džeroski, S., Muggleton, S., & Russell, S. (1992). Pac-learnability of determinate logic programs. In *Proceedings of the Fifth Annual ACM Workshop on Computational Learning Theory*, pp. 128–135.

Erol, K., Hendler, J., & Nau, D. (1994). HTN planning: complexity and expressivity. In *Proceedings of the Twelfth National Conference on Artificial Intelligence (AAAI-94)*. AAAI Press.

Frazier, M., & Pitt, L. (1993). Learning from entailment: An application to propositional Horn sentences. In *Proceedings of the Tenth International Conference on Machine Learning*, pp. 120–127.

Khardon, R. (1996). Learning to take actions. In *Proceedings of the Thirteenth National Conference on Artificial Intelligence (AAAI-96)*, pp. 787–792.

Khardon, R. (1998). Learning first order universal horn expressions. In *Proceedings of the Eleventh Annual Conference on Computational Learning Theory (COLT-98)*.

Lloyd, J. (1987). *Foundations of Logic Programming* (2nd ed.). Springer-Verlag, Berlin.

Muggleton, S., & Feng, C. (1990). Efficient induction of logic programs. In *Proceedings of the First Conference on Algorithmic Learning Theory*, pp. 368–381. Ohmsha/Springer-Verlag.

Natarajan, B. (1989). On learning from exercises. In *Proceedings of the Second Workshop on Computational Learning Theory*, pp. 72–87. Morgan Kaufmann.

Page, C. D. (1993). *Anti-Unification in Constraint Logics: Foundations and Applications to Learnability in First-Order Logic, to Speed-up Learning, and to Deduction*. Ph.D. thesis, University of Illinois, Urbana, IL.

Plotkin, G. (1970). A note on inductive generalization. In Meltzer, B., & Michie, D. (Eds.), *Machine Intelligence*, Vol. 5, pp. 153–163. Elsevier North-Holland, New York.

Reddy, C., & Tadepalli, P. (1997a). Learning goal-decomposition rules using exercises. In *Proceedings of the 14th International Conference on Machine Learning*. Morgan Kaufmann.

Reddy, C., & Tadepalli, P. (1997b). Learning Horn definitions using equivalence and membership queries. In *Proceedings of the 7th International Workshop on Inductive Logic Programming*. Springer Verlag.

Sammut, C. A., & Banerji, R. (1986). Learning concepts by asking questions. In *Machine learning: An artificial intelligence approach*, Vol. 2. Morgan Kaufmann.

This article was processed using the LaTeX macro package with LLNCS style