Learning Horn Definitions: Theory and an Application to Planning

Chandra REDDY and Prasad TADEPALLI

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

Received 31 May 1998

Abstract

A Horn definition is a set of Horn clauses with the same head literal. In this paper, we consider learning non-recursive, first-order Horn definitions from entailment. We show that this class is exactly learnable from equivalence and membership queries. It follows then that this class is PAC learnable using examples and membership queries. Finally, we apply our results to learning control knowledge for efficient planning in the form of goal-decomposition rules.

Keywords Horn Definition, Horn Program, Horn Sentence, Horn Clause, Learning from Entailment, Planning, Control Knowledge, Queries, PAC-learning.

§1 Introduction

Horn clauses is one of the popular ways of representing first order knowledge. In this paper, we consider learning Horn definitions—multiple Horn clauses with the same predicate in the heads of all clauses—in the *learning from entailment* setting.[15, 8] In this setting, the target concept is a Horn definition. A positive (negative) example is a Horn clause (not) entailed by the target. Learning Horn definitions is a fundamental problem both in Inductive Logic Programming (ILP) and in Computational Learning Theory. Since it is NP-hard to test membership in this concept class, it immediately follows that even non-recursive Horn definitions are hard to learn from examples alone.[33] Using only equivalence queries, single non-recursive Prolog clauses are learnable with restrictions such as determinacy and bounded arity.[5, 10] Restricted versions of single recursive clauses are also shown to be learnable.[6] However, learning multiple clauses or even slightly more general versions of either recursive or non-recursive clauses is shown to be hard without further help.[7] Page has shown that non-recursive Horn definitions with predicates having fixed arity and with the restriction that the clauses be "simple," i.e., only the variables and terms that occur in the head literal of a clause appear in the body of the clause, are learnable using equivalence and subset queries.[25] In this paper, we examine the learnability of a more general class.

In particular, we show that first-order non-recursive Horn definitions are exactly learnable from membership and equivalence queries with no other restrictions. In particular, the target concepts may have arbitrary number of clauses with the number and the arity of the literals in each clause also being unbounded. The literals might also contain functions. Learning from equivalence and membership queries is one of the standard models (also called "minimally adequate teacher" by Angluin[2]) used in Computational Learning Theory literature. This is a natural model to consider when the learner has a choice of asking whether a given instance is positive, i.e., entailed by the target. Some languages such as deterministic finite state automata and propositional Horn sentences which appear not to be learnable from examples alone are learnable in this model. At the same time, it is a nontrivial model in that there are many languages, even apparently "simple" ones, such as arbitrary Boolean formulas, which are not learnable in this model (under some cryptographic assumptions). It is also known that for some languages such as DNF, membership queries do not help. Thus, learning a first-order language such as Horn definitions in this learning model is an important problem, left open by by Angluin, et al.[3] Most previous theoretical work in ILP relies on the corresponding propositional algorithms, and hence does not really show the importance of using a first-order language. Our work is almost unique in that the hypothesis space we consider cannot be reduced to one that a propositional learner can learn efficiently. This is discussed in more detail in Section §5.

Our algorithm combines the ideas of several previous learning algorithms that use membership queries.[2, 15, 17, 16] It maintains a set of hypothesis clauses, each of which is subsumed by a corresponding target Horn clause. Given a new positive example, it either combines it with one of its hypothesis clauses producing a least general generalization (lgg) of the example and the hypothesis clause, or stores it as a new hypothesis clause. It uses membership queries to decide which hypothesis clause, if any, should an example be combined with. An example is combined with that clause which yields an lgg that is entailed by the target. The algorithm exploits the fact that there is at most one positive literal in a Horn clause, which makes it possible to show that any clause which is entailed by the target must be subsumed by one of the clauses in the target—a property called "strong compactness." This guarantees that the membership queries, in effect, check whether a hypothesis clause is subsumed by a target clause. After combining the example with a hypothesis clause, the resulting lgg is pruned of redundant literals using membership queries. Without this step, the number of literals in the hypothesis clause can grow geometrically with each new example, exceeding any polynomial bound.

Learnability in our "exact-learning model" that uses equivalence and membership queries, implies learnability in the PAC-learning model that uses random examples and membership queries.[2] On the practical side, our research is motivated by an application that involves learning goal-decomposition rules (d-rules) in planning. We show that this problem can be reduced to one of learning Horn definitions, which enabled us to apply our learning algorithm for Horn definitions to efficiently learn d-rules. We implemented a system called ExEL that employs the learning algorithm for Horn definitions to learn d-rules. We demonstrate ExEL's success in learning d-rules in the STRIPS-world and a simplified version of Air Traffic Control domain.

The rest of the paper is organized as follows: Section 2 presents some formal preliminaries about Horn definitions. Section 3 describes the learning problem, proves some properties of Horn definitions, describes the learning algorithm and proves its correctness. Section 4 employs this result to show that goal-decomposition rules are learnable. This section also gives experimental results for learning d-rules by ExEL. Section 5 concludes the paper by relating it to previous work in this area and discussing its implications.

§2 Preliminaries

In this section, we define and describe the terms we use in the rest of the paper, omitting some of the standard terminology and notation of logic (as given in standard books[22]).

Definition 2.1 A **term** is defined recursively as follows: (1) a variable is a term; (2) a constant is a term; and (3) if f is an n-ary function symbol and $t_1, t_2, \ldots, t_n$ are terms, then $f(t_1, t_2, \ldots, t_n)$ is also a term.

Definition 2.2 If p is an n-ary predicate symbol, and $t_1, t_2, \ldots, t_n$ are terms, then $p(t_1, t_2, \ldots, t_n)$ is called an **atom**. A **literal** is an atom (positive literal), or a negation of an atom (negative literal).

Definition 2.3 A **definite Horn clause** (Horn clause or clause, for short) is a finite set of literals that contains exactly one positive literal. It is treated as a disjunction of the literals in the set with universal quantification over all the variables. The positive literal is called the *head* of the clause, and the set of negative literals is called the *body*. A Horn clause is **non-recursive** if the predicate symbol of the head literal of the Horn clause does not occur in its body.

We usually denote a Horn clause as *body* $\rightarrow$ *head*.

Definition 2.4 A **Horn definition** is a set of Horn clauses where the heads of all clauses have the same predicate symbol.[*1] It is **non-recursive** if the head predicate symbol does not occur in

[*1] A Horn definition is also called a *predicate definition*.

any negative literal in any clause in the definition.

Definition 2.5 A clause D **subsumes** a clause E if there exists a substitution θ such that $D\theta \subseteq E$. We denote this as $D \succeq E$, and read it as D subsumes E or as D is more general than E.

Definition 2.6 If D and E are clauses such that $D \succeq E$, then a literal l in a clause E is relevant (irrelevant) w.r.t the clause D, if $D \not\succeq E - \{l\}$ ($D \succeq E - \{l\}$, respectively).

Definition 2.7 If D and E are two clauses such that $D \succeq E$, then a **condensation** of E w.r.t. D is a clause E' such that $E' \subseteq E$, $D \succeq E'$, and for any $l \in E'$, $D \not\succeq E' - \{l\}$.

For example, if $D = \{\neg p_1(x), p_2(y)\}$ and $E = \{\neg p_1(a), p_2(b), p_2(c), p_3(c)\}$, then both $\{\neg p_1(a), p_2(b)\}$ and $\{\neg p_1(a), p_2(c)\}$ are condensations of E w.r.t. D.

Definition 2.8 **Least general generalization** of a set of clauses S over $\succeq$ is a clause D such that (1) for every E in S, $D \succeq E$, and (2) if there exists a clause F such that for every E in S, $F \succeq E$, then $F \succeq D$.

The definitions 2.5 and 2.8 are due to Plotkin.[27] The existence of least general generalization is shown by Plotkin[27], and by Nienhuys-Cheng and de Wolf.[24]

We follow the description by Muggleton and Feng[23] of Plotkin's algorithm to find the least general generalizations (lgg) of a set of clauses. The lgg of two clauses C_1 and C_2 is $\cup_{l_1 \in C_1, l_2 \in C_2} lgg(l_1, l_2)$. The lgg of two literals $p(a_1, a_2, \ldots, a_n)$ and $p(b_1, b_2, \ldots, b_n)$ is $\{p(lgg(a_1, b_1), lgg(a_2, b_2), \ldots, lgg(a_n, b_n))\}$; if the predicate symbols are not equal, their sign differs or their arity differs, then their lgg is $\{\}$, the empty set. 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 pair of clauses. We use symbols that start with a '?' to denote variables.

For example, let $C_1 = \{\neg p_1(f(a), b), \neg p_2(a, c), p_3(b)\}$ and $C_2 = \{\neg p_1(f(c), d), \neg p_1(b, a), \neg p_2(c, c),\ p_3(a)\}$. Then $lgg(C_1, C_2) = \{\neg p_1(f(?x), ?y), \neg p_1(?z, ?u), \neg p_2(?x, c), p_3(?u)\}$, where the variables $?x$, $?y$, $?z$ and $?u$ stand for the pairs $lgg(a, c)$, $lgg(b, d)$, $lgg(f(a), b)$ and $lgg(b, a)$, respectively.

Note that $|lgg(C_1, C_2)|$ can be equal to $|C_1| \times |C_2|$.

Lemma 2.1 Let C_1, C_2 and C_3 be Horn clauses. Then $C_1 \succeq C_2$ and $C_1 \succeq C_3$ if and only if $C_1 \succeq lgg(C_2, C_3)$.

Proof. The only-if part follows from the property (2) of the definition of least-general generalization. The if part follows from the transitive property of $\succeq$. □

We state the following fact explicitly, although it is straightforward, for it is useful later.

Proposition 2.1 If $C_1 \succeq C_2$ then $C_1 \succeq C_3$ for any C_3 such that $C_2 \subseteq C_3$.

§3 Learning Horn Definitions

In this section, we first specify our learning problem. Next we describe the learning algorithm and then give the learnability result.

3.1 Learning Problem

Our learning problem is motivated by learning control knowledge for planning in structural domains. The following definitions reflect that motivation.

Definition 3.1 A *scene* is a conjunction of positive ground literals describing a set of objects.[*2] We call the predicates that occur in scenes *base predicates*. We differentiate the base predicates from a special predicate called the *goal predicate*. An *instance* is a 2-tuple comprising a scene *scene* and a ground goal literal g, meaning that g is true whenever *scene* is true. We alternately write an instance as a clause $\mathit{scene} \rightarrow g$.

We consider the hypothesis space of Horn definitions for the goal predicate. Following the *learning from entailment* model, an instance $\langle \mathit{scene}, g\rangle$ is *in a* hypothesis H iff the minimal model of H with respect to the literals in *scene* satisfies g. In other words, $\langle \mathit{scene}, g\rangle$ is an instance of H iff $H \models (\mathit{scene} \rightarrow g)$. Such an instance is a *positive example* of H. All other instances are *negative examples*.

Henceforth, Σ denotes the target concept in the hypothesis space.

Example 3.1 The following illustrates the above definitions in a toy version of an air-traffic control domain.

$\Sigma = \{$

```
plane-at(?p,?loc), level(L1,?loc), free-runway(?r), short-runway(r), land-short?(?p) → land-plane(?p);
plane-at(?p,?loc), level(L1,?loc), free-runway(?r), long-runway(?r) → land-plane(?p)
```

$\}$

The first clause in Σ gives the conditions under which a plane can land on short runways. The second clause is for long-runway landing. The following is a positive example of Σ (for the second clause):

```
plane-at(P737, 10), level(L1, 10), free-runway(R1), long-runway(R1), short-runway(R2),
wind-speed(high), wind-dir(south), free-runway(R2) → land-plane(P737).
```

□

Before stating the learning problem, we define the queries we will need.[2)]

[*2] We employ closed-world assumption and assume that all other literals are negative.

Definition 3.2 A *membership query* takes as input an instance x, and outputs *yes* if x is in Σ, and *no* otherwise. An *equivalence query* takes as input a hypothesis H, and outputs *yes* if H and Σ are logically equivalent; otherwise, returns a *counterexample* from $H \oplus \Sigma$ —i.e., an instance that is in one but not in the other.

The above combination of queries is called a "minimally adequate teacher" by Angluin. The learning problem in the exact learning model[2)] is as follows:

Definition 3.3 An algorithm *exactly learns* a concept class $\mathcal{C}$ if for every concept $\Sigma \in C$, if it asks equivalence and membership queries, terminates in time polynomial in the size of Σ and the size of the largest counterexample, and outputs a hypothesis which is logically equivalent to Σ.

In the rest of this section, we will be showing that the class of non-recursive Horn definitions is exactly learnable from equivalence and membership queries. Note that learning exactly does not mean learning a syntactically equivalent definition, but only a semantically equivalent one. In other words, the learner must ask an equivalence query for which there is no counterexample.

3.2 Strong Compactness of Non-recursive Horn Definitions

In this section we describe a property of non-recursive Horn definitions, which is called *strong compactness* by Lassez, et al.[20)] and Page[25)], and relate this property to membership queries.

Strong compactness says that for non-recursive Horn definitions if we know that a clause is logically implied by a set of clauses Σ, then we can conclude that that clause is subsumed by a clause in Σ. The following lemma, in addition, says that the converse is true. This is useful to show later that each clause in the current hypothesis of our algorithm is always a specialization of some target clause.

Lemma 3.1 Let Σ be a non-recursive Horn definition, and h be a Horn clause which is not a tautology. Then, $\Sigma \models h$ if and only if there exists a clause C in Σ such that $C \succeq h$. We call C the **target clause** of h, and h the **hypothesis clause** of C.

Proof. The if part follows from the conjunctive interpretation of all the clauses in Σ. The only-if part is a direct consequence of the Subsumption theorem.[19)] We give a brief sketch of the proof here. Since $\Sigma \models h$ and h is not a tautology, there must be a non-trivial proof of h from the clauses of Σ. However, since the head predicate symbol of the clauses in Σ does not appear in the body of any clause, there can be no chaining of the clauses in the proof of h. This implies that h must be subsumed by a single clause in Σ. □

If a clause h has variables, determining $\Sigma \models h$ is equivalent to determining whether all instances in h are also in Σ—which is the same as a subset query.[2)] However, by substituting each variable in h by a unique constant—skolemization—we can form a fully ground clause that is an instance of h. Now, determining whether $\Sigma \models h$ is equivalent to asking whether $\Sigma \models \mathsf{Skolemize}(h)$. Asking whether $\Sigma \models \mathsf{Skolemize}(h)$ is the same as a membership query, since $\mathsf{Skolemize}(h)$ is ground. In effect, this membership query simulates a subset query.

3.3 Learnability of Non-recursive Horn Definitions

Horn-learn is an algorithm to learn non-recursive Horn definitions using equivalence and membership queries (Fig. 1). Horn-learn makes use of Generalize algorithm. Generalize takes as input a Horn clause and generalizes it by eliminating literals from that Horn clause. It removes a literal from the Horn clause and checks whether the resultant Horn clause is overgeneral. It can do this by substituting each variable in the hypothesis clause with a unique constant and asking a membership query. If it is overgeneral the literal is retained; otherwise, it is eliminated to form a new, more general Horn clause.

Horn-learn starts with hypothesis H that is initially empty. As long as H is not equivalent to the target concept C, the equivalence query returns an example e that is not included in H, and the algorithm modifies H to cover e. To include e in H, Horn-learn checks each Horn clause h_i of H whether generalizing h_i to cover e would not make the hypothesis overgeneral—i.e., whether $lgg(h_i, e)$ is in the target concept. If so, concluding that it has found the right Horn clause h_i to include e in, Horn-learn further generalizes $h = lgg(h_i, e)$, by removing irrelevant literals, i.e., those literals whose removal preserves the entailment relation between Σ and h. The entailment relation is checked by using the membership oracle on the result of skolemizing h (see Generalize in Fig. 1). Horn-Learn finally replaces h_i in H by the new generalized h. If there is no h_i such that $lgg(h_i, e)$ is entailed by the target, it generalizes e and makes it a new Horn clause of H.

Example 3.2
Let Σ be $\{\rightarrow q(f(f(?x))), ?x); p_1(?x, ?y), p_1(?y, ?z) \rightarrow q(?x, ?z);\ p_1(?x, ?y), p_2(?y, ?z) \rightarrow q(?x, ?z)\}$.
Let the first example be $e1$: $p_1(a, b), p_1(a, d), p_1(b, z), p_2(c, b), p_2(c, d), p_2(d, e) \rightarrow q(a, e)$.
Since H is empty, next step is Generalize($e1$).

In Generalize:

  $\Sigma \models p_1(a, d), p_1(b, z), p_2(c, b), p_2(c, d), p_2(d, e) \rightarrow q(a, e)$?
  yes, so drop $p_1(a, b)$.
  $\Sigma \models p_1(b, z), p_2(c, b), p_2(c, d), p_2(d, e) \rightarrow q(a, e)$?
  no, keep $p_1(a, d)$.
  ...
  Finally, $h' = p_1(a, d), p_2(d, e) \rightarrow q(a, e)$

$h_1 = p_1(a, d), p_2(d, e) \rightarrow q(a, e)$
Let the next example be $e2$: $p_1(a, b), p_1(a, d), p_1(b, z), p_2(c, b), p_2(c, d), p_2(d, e) \rightarrow q(a, z)$.

```
Horn-learn
1.  Let $\Sigma$ be the target concept.
2.  $H := \{\}$ /* empty hypothesis, initially */
3.  $m := 0$ /* number of clauses in the hypothesis */
4.  while equivalence($H$, $\Sigma$) is not true and $e$ is a counterexample do
/* fix the clause in $H$ for the example $e$ */
5.      if ($m > 0$) then, Let $H$ be $\{h_1, h_2, \ldots, h_m\}$
6.      found := false; $i := 1$
7.      while ($i \leq m$) and found is false do
8.          $h := lgg(e, h_i)$
9.          if $\Sigma \models h$ then found := true; /* Member?(Skolemize($h$)) implements $\Sigma \models h$ */
10.         else $i := i + 1$
11.     endwhile /* $i \leq m$ */
12.     if found = false then $h := e$; $m := m + 1$;
13.     $h_i :=$ Generalize($h$) /* further generalize $h$ */
14. endwhile
15. return $H$

Generalize($h$)
1.  $h' := h$
2.  for each literal $l$ in $h$ do
3.      if $\Sigma \models h' - \{l\}$ then $h' := h' - \{l\}$ /* Implemented by Member?(Skolemize($h' - \{l\}$)) */
4.  Return $h'$.
```

Fig. 1 Horn-learn: An algorithm to learn Horn definitions

$lgg(h_1, e2) = p_1(a, ?db), p_1(a, d), p_1(?ab, ?dz), p_2(?dc, ?eb), p_2(?dc, ?ed), p_2(d, e) \rightarrow q(a, ?ez)$
$\Sigma \models lgg(h_1, e2)$? *no.*
So, Generalize$(e2) = h_2 = p_1(a, b), p_1(b, z) \rightarrow q(a, z)$.

Let the next example be $e3$: $p_1(r, s), p_2(s, t), p_1(r, u), p_2(u, v) \rightarrow q(r, t)$.
$lgg(h_1, e3) = p_1(?ar, ?ds), p_2(?ds, ?et), p_1(?ar, ?du), p_2(?du, ?ev) \rightarrow q(?ar, ?et)$
$\Sigma \models lgg(h_1, e3)$? *yes.*
$h_1 =$ Generalize$(lgg(h_1, e3)) = p_1(?ar, ?ds), p_2(?ds, ?et) \rightarrow q(?ar, ?et)$.

Let the next example be $e4$: $p_1(a, b) \rightarrow q(f(f(a)), a)$
$lgg(h_1, e4) = p_1(?ar, ?ds) \rightarrow q(?arf, ?eta)$
$\Sigma \models lgg(h_1, e4)$? *no.*
$lgg(h_2, e4) = p_1(a, b) \rightarrow q(?af, ?za)$

$\Sigma \models lgg(h_2, e4)$? *no.*
Generalize$(e4) = h_3 = \rightarrow q(f(f(a)), a)$.
... □

The generalization process of Generalize serves a critical purpose. Recall that the size of lgg grows as a product of the sizes of the hypotheses being generalized. Unless the hypothesis size is limited, it can grow exponentially in the number of examples used to create the hypothesis. Lemma 3.2 and Lemma 3.3 together show that Generalize guarantees that the sizes of the hypothesis clauses are at most the sizes of their corresponding target clauses. Lemma 3.2 shows that Generalize does not over-generalize in the process.

Lemma 3.2 If the argument h of Generalize is such that $\Sigma \models h$ then, at the end of Generalize, h' has a target Horn clause C_j—i.e., $C_j \succeq h'$. Moreover, h' in line 4 of Generalize is a condensation of h w.r.t. C_j.

Proof. In the beginning of Generalize, h', which is the same as the argument h, is not overgeneral. h' is modified only when the modification still leaves the result inside Σ. That is, $\Sigma \models h'$. By Lemma 3.1, there exists a target Horn clause for h', say C_j, and $C_j \succeq h'$.

To show that h' in line 4 of Generalize is a condensation of h w.r.t. C_j, we need only to show that for any literal $l \in h'$, $C_j \not\succeq (h' - \{l\})$. Suppose that for some $l \in h'$, $C_j \succeq (h' - \{l\})$. Let h'' be the value of h' when l is considered for removal in the loop of lines 2—3. Since $h' \subseteq h''$, by Proposition 2.1, $C_j \succeq (h'' - \{l\})$. From Lemma 3.1, $\Sigma \models (h'' - \{l\})$. In that case, l would have been removed by line 3 of Generalize. But, $l \in h'$, a contradiction. Therefore, for any literal $l \in h'$, $C_j \not\succeq (h' - \{l\})$. □

Lemma 3.3 If h' is a condensation of h w.r.t. C_j, then $C_j\theta = h'$ for some substitution θ. Moreover, $|h'| \leq |C_j|$.

Proof. Suppose h' is a condensation of h w.r.t. C_j. Then there exists a θ such that $C_j\theta \subseteq h'$. Suppose $C_j\theta \subset h'$. Then, for some $l \in h' - C_j\theta$, $C_j\theta \subseteq h' - \{l\}$. Hence, $C_j \succeq (h' - \{l\})$. This is a contradiction, since h' is a condensation w.r.t. C_j. Therefore, $C_j\theta = h'$. This implies that $|h'| = |C_j\theta| \leq |C_j|$. □

The following definition relates an example to a hypothesis clause and to a target clause.

Definition 3.4 If $C_1, C_2, \ldots, C_n$ are the Horn clauses in the target concept Σ, and $h_1, h_2, \ldots, h_m$ are the Horn clauses in the hypothesis H, then a *correct hypothesis Horn clause* in H for an example e is a Horn clause h_i such that for some $1 \leq j \leq n$, $C_j \succeq e$ and $C_j \succeq h_i$.

Lemma 3.4 In Horn-learn, suppose that e is a counterexample returned by the equivalence query such that e is covered by Σ, but not by H. Then Horn-learn includes e in a correct hypothesis Horn clause in H for e if and only if one exists.

Proof. First the only-if part. Horn-learn includes e in h_i of H if $\Sigma \models lgg(e, h_i)$. If $\Sigma \models lgg(e, h_i)$, then, by Lemma 3.1, $C_j \succeq lgg(e, h_i)$ for some C_j of C. Then, by Lemma 2.1, $C_j \succeq e$ and $C_j \succeq h_i$. Therefore, if Horn-learn includes e in h_i of H, then h_i is a correct hypothesis Horn clause for e.

Now, the if part of the claim. Let h_i be a correct hypothesis Horn clause for e in H such that no h_k such that $k < i$ is one. Then there exists a C_j of C such that $C_j \succeq e$ and $C_j \succeq h_i$. This implies, by Lemma 2.1, that $C_j \succeq lgg(e, h_i)$. By Lemma 3.1, $\Sigma \models lgg(e, h_i)$. Also, for $k < i$, $C_j \not\succeq h_k$, which implies $\Sigma \not\models lgg(e, h_k)$. Therefore, h_i is the first clause in the hypothesis H for which $\Sigma \models lgg(e, h_i)$. Then, by lines 7–13 in Fig. 1, e is included in h_i by assigning the result of Generalize$(lgg(e, h_i))$ to h_i. □

Lemma 3.5 Suppose that e is a counterexample such that e is covered by Σ, but not by H. Then Horn-learn adds a new Horn clause to H that includes e if and only if H does not already have a correct hypothesis Horn clause for e.

Proof. By line 12 and Lemma 3.4. □

Lemma 3.6 The following are invariant conditions of Horn-learn:
1. Every Horn clause h_i in the hypothesis H has a target clause;
2. Every Horn clause C_j in the target concept Σ has at most one hypothesis clause. C.

Proof.
Proof of (1). For every Horn clause h_i in H, $\Sigma \models h_i$. This is true because (a) the input h to Generalize is checked to be such that $\Sigma \models h$, (by lines 7–13 of Horn-learn), and (b) by Lemma 3.2 the output of Generalize, which replaces h_i, preserves this condition. Therefore, by Lemma 3.1, h_i has a target Horn clause.
Proof of (2). 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 have distinct target clauses at the beginning of an iteration of the loop of lines 4–14, then they still have distinct target clauses at the end of the iteration.

Let h_i be the first hypothesis Horn clause in H for C_j. That is, there is no h_k such that $k < i$ and h_k is a hypothesis Horn clause in H for C_j. Another hypothesis clause $h_{i'}$ with the target clause C_j would have been added to H such that $i' > i$, only if there was a counterexample e belonging to C_j for which h_i is not the correct hypothesis Horn clause (by Lemma 3.5). That means $C_j \succeq e$ and $C_j \not\succeq lgg(h_i, e)$. This implies, by Lemma 2.1, $C_j \not\succeq h_i$. That is a contradiction, because h_i is a hypothesis Horn clause for C_j. Therefore, such a $h_{i'}$ cannot exist in H. That is, $h_{i'}$ could have been added only if it had a distinct target clause.

Let C_j and $C_{j'}$ be two distinct target clauses for the clauses h_i and $h_{i'}$ in H, respectively, at the beginning of an iteration of the loop of lines 4–14. That means, $C_j \succeq h_i$ and $C_j \not\succeq h_{i'}$. Also, $C_{j'} \succeq h_{i'}$ and $C_{j'} \not\succeq h_i$.

At most one of h_i and $h_{i'}$ 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. h_i can change in the lines 8 and 13. We need to show that both these changes maintain that $C_{j'} \not\succeq h_i$. Since $C_{j'} \not\succeq h_i$, $C_{j'} \not\succeq lgg(h_i, e)$ (by Lemma 2.1). Therefore, line 8 maintains the property. In line 13, Generalize returns a subset of its argument. By the contrapositive of Proposition 2.1, $C_{j'} \not\succeq$ Generalize$(lgg(h_i, e))$, thus maintaining the property. Therefore, h_i and $h_{i'}$ have different target clauses at the end of the iteration. □

Now to the main theorem on the exact learnability.

Theorem 3.1 Non-recursive Horn definitions are exactly learnable using equivalence and membership queries.

Proof. We prove this theorem by showing that Horn-learn exactly learns non-recursive Horn definitions.

Part 1 of Lemma 3.6 implies that for every h_i of H, there is a C_j such that $C_j \succeq h_i$. That means, H covers no example that is not covered by the target concept C. In other words, H is never over-general in Horn-learn. Therefore, every counterexample is an example that is covered by Σ, but not by H.

Equivalence query guarantees that whenever Horn-learn gets a new example, it is not already covered by the hypothesis H. At the end of each iteration, before asking an equivalence query, by Lemma 3.4 and Lemma 3.5, Horn-learn guarantees that all the previous examples are covered by H. Each example, either modifies an existing hypothesis Horn clause (its correct hypothesis Horn clause) or adds a new Horn clause. The minimum change in H that is required to cover a new example is a change of a variable in its correct hypothesis Horn clause if one exists. That is, each new example, except the ones that add new Horn clauses, contributes at least one variable. Let n be the number of Horn clauses in a concept, l be the maximum number of literals in a clause in the concept, v be the maximum number of variables in a clause in the concept, and k be the number of literals in the largest counterexample. Because Generalize guarantees that each Horn clause in the hypothesis has at most as many literals as there are in its target Horn clause (by Lemma 3.2 and Lemma 3.3), the number of variables in each Horn clause is at most v. Part 1 of Lemma 3.6 guarantees that H has at most n Horn clauses. Therefore, the total number of variables is at most nv. Horn-learn requires n examples to add each of the n Horn clauses in H. It requires at most nv examples to variablize all the Horn clauses in H. Therefore, Horn-learn requires $n(v + 1)$ examples, and, hence, $n(v + 1)$ equivalence queries.

Let m be the number of hypothesis clauses in the hypothesis H at any time. Then, for each of the base examples that form new Horn clause in H, Horn-learn asks at most m membership queries for deciding that there is no correct hypothesis Horn clause in H, and at most k membership queries to simplify and generalize using Generalize (because there are at most k literals in an example). Each new Horn clause has at most l literals (by Lemma 3.2). For each of the other examples, at most m membership queries are needed to determine a correct hypothesis Horn clause, and kl (which is the size of lgg) number of membership queries to generalize using Generalize. Therefore, the total number of queries is at most $mn + kn + nv(m + kl)$, which is at most $n^2 + kn + nv(n + kl)$. This is also an upper bound on the running time of the algorithm. □

By the above theorem and the transformation result from the exact learning model to the PAC model[2], we have the following.

Corollary 3.1 Non-recursive Horn definitions are polynomial-time PAC-learnable using membership queries.

§4 Learnability of Goal-Decomposition Rules

In AI planning, domain-specific control knowledge is necessary to make the planning task computationally feasible. Goal-decomposition rules (d-rules) is a natural method for representing control knowledge.[32] They are similar to hierarchical transition networks.[11]

A d-rule is a 3-tuple $\langle g, c, sg \rangle$ that decomposes a goal g into a sequence of subgoals sg, provided condition c holds in the initial state. Goal g is a positive first-order literal, condition c is a conjunction of positive first-order literals. Subgoals sg are first-order positive literals.

The following is an example of a simple d-rule from Blocks-world (BW) domain:

```
       goal:   on(?x, ?y)
   subgoals:   clear(?x), clear(?y), put-on(?x, ?z, ?y)
 conditions:   block(?x), block(?y), table(?z)
```

A goal may have multiple d-rules—i.e., multiple condition-and-subgoals pairs. For example, there could be some other rules for the goal `on(?x, ?y)` with different condition-and-subgoals pairs. We call these disjunctive d-rules. If a goal does not appear anywhere in the conditions or subgoals, then we refer to them as non-recursive d-rules. The above example is a non-recursive d-rule.

So far, we have looked at d-rules for a single goal. In a general planning domain, there can be several goals each with their own set of d-rules. Each goal has subgoals, which themselves can be regarded as goals, having their own d-rules. That is, in general, there are goal-subgoal hierarchies, and, correspondingly, d-rule hierarchies.

First, we look at how d-rules for a goal can be seen as Horn definitions. Next, we adapt the algorithm for learning Horn definitions to learn a d-rule hierarchy.

4.1 D-rules as Horn definitions

In this section, we show that non-recursive disjunctive d-rules are learnable by converting them into a set of non-recursive Horn clauses.

Recall that a d-rule is comprised of three parts: goal, initial conditions, and a sequence of subgoals. The examples for this purpose are positive examples, each of which has a goal, and a sequence of successive states starting from an initial state and leading to a goal state. To represent the notion of state, which is missing in the Horn clause, we add special symbols that denote "situations" to the literals. In particular, the first two parameters of each literal are new, and denote the names of the situations in between which that literal must be true. The first parameter specifies the starting situation in which the literal must be true. The second parameter specifies the situation up to which the literal must be true. We call these two parameters of a literal the situation parameters. When we mean that a literal is true in a particular situation alone, that situation is mentioned in both the situation parameters of the annotation of the literal. When the situation parameters in a literal are different, it means that the literal is true throughout the duration between the situations represented by the situation parameters. In addition, the two situation parameters in a literal implicitly indicate that the first situation occurs before the second situation or that both the situations are the same. However, this in itself may not fully represent all the relative orderings between situations we want to specify. Therefore, to explicitly represent the relative ordering of two situations `Si` and `Sj`, we use a special predicate symbol `not-after` and add the literal `not-after(Si, Sj)`, meaning that the situation `Si` does not occur after the situation `Sj`.

A d-rule can be declaratively read as follows: starting from a state that satisfies the initial conditions of the d-rule, if each of the subgoals is achieved one by one in sequence, then the goal of the d-rule would be true in the state that achieved the last subgoal in the sequence. This declarative reading makes the connection between d-rules and Horn clauses explicit. The goal of a d-rule corresponds to the head literal of the corresponding Horn clause and is true in the final situation. The initial conditions, which are conjunctions of positive literals, and the subgoals, which are single positive literals, when properly annotated with situation variables, correspond to the body of the Horn clause. In addition, we might need to add some `not-after` literals to constrain the relative orderings between different situation variables that correspond to different subgoals.

The d-rule in BW domain, mentioned at the beginning of this section, translates into the following Horn clause.

```
block(?S0, ?S0, ?x), block(?S0, ?S0, ?y), table(?S0, ?S0, ?z), clear(?S1, ?S2, ?x),
clear(?S2, ?S3, ?y), put-on(?S3, ?S4, ?x, ?z, ?y), not-after(?S0, ?S1) → on(?S4, ?S4, ?x, ?y)
```

The first three literals in the body of the Horn clause correspond to the initial conditions of the d-rule. Since these literals must be true in the initial state, they are given the situation parameters corresponding to the initial state (?S0). Next three literals in the body of the Horn clause correspond to the subgoals of the d-rule. They are given situations such that they are true one after the other. The last literal in the body, explicitly states that the situation ?S0 does not come after the situation ?S1. This, with the implicit orderings between the situations present in the other literals, suggests that ?S0 is the initial situation or state. Finally, in ?S4, the goal literal is true. It is mentioned as the head of the clause.

Note that it is possible to express partial orders using this notation, by simply not specifying not-after relation between situations. For instance, if we would like to specify that clear(?x) and clear(?y) can be achieved in any order, we can replace the literal clear(?S1, ?S2, ?x) by clear(?S1, ?S3, ?x), and add not-after(?S0, ?S2). By this, we say that ?S1 and ?S2 are not relatively ordered, but both of them are preceded by ?S0, and succeeded by ?S3.

Since the objective is learning d-rules via learning Horn clauses, training examples for d-rules should be converted to training examples for Horn clauses. A training example for learning d-rules has a sequence of states, S0, S1,…, Sn and a goal. An example can be viewed as a fully instantiated (ground) d-rule specifying the initial condition and a sequence of subgoals, with both including several irrelevant literals. It can then be converted into a Horn-clause as described above.

In particular, each state of the example is a set of positive literals describing the relationships between objects in a state. A state may have literals corresponding to subgoals achieved in that state. Along with a sequence of states the example has an instance of a goal that is true in the last state. Each state is given a situation number. In the corresponding Horn-clause form, each literal is annotated with situation parameters Si and Sj as its first two parameters, where Si and Sj represent a maximal duration in which the literal is true. For example, suppose the literal clear(?x) is true in S0, S1, S2, and S3, and again in S5 and S6, but nowhere else. Then, its corresponding Horn-clause form would have only the literals clear(S0, S3, ?x) and clear(S5, S6, ?x). Then, for each state Si, there is a set of literals comprising the literal not-after(Si, Si) and the literals not-after(Si, Sj) for each Sj such that $i < j \leq n$. These two sets of literals form the body of the corresponding Horn-clause example. The goal literal annotated with the situation number of the state in which the goal is true, becomes the head of the Horn-clause example.

Thus a target d-rule and its examples can be converted to Horn clauses. Target d-rules for a goal can then be represented as Horn definitions.

However, there is a glitch here. There is a dependency among the not-after literals in that they are transitive: not-after(?Si, ?Sj), not-after(?Sj, ?Sk) $\rightarrow$ not-after(?Si, ?Sk). This makes the Lemmas 3.1 and 3.2 inapplicable. Following our work on Horn programs[31)], we order the not-after literals in the input h of Generalize such that the literals that match not-after(?Si, ?Sj) and not-after(?Sj, ?Sk) come earlier than the literals that are implied by them, such as not-after(?Si, ?Sk). Hence, Generalize considers the literals for removal in that order. This way, the output of Generalize would be a condensation, as was the case without the not-after literals. The idea here is that, in Generalize, if we removed `not-after(?Si, ?Sk)`, before we removed `not-after(?Si, ?Sj)` and `not-after(?Sj, ?Sk)`, the membership query would be answered *yes*, and we would remove `not-after(?Si, ?Sk)`. This may cause Generalize to leave over two literals `not-after(?Si, ?Sj)` and `not-after(?Sj, ?Sk)`, instead of just `not-after(?Si, ?Sk)`. This way the output of Generalize may neither be a condensation nor be subsumed by a clause in the target. Ordering the removal of literals in the above manner, overcomes this problem.

With the above change, and from Theorem 3.1, it follows that d-rules can be learned from examples and membership queries.

4.2 Implementation

There is a problem in utilizing the algorithm for learning Horn definitions to the task of learning d-rules. A d-rule hierarchy, for a planning domain, when converted into Horn clause notation, is a Horn program rather than a Horn definition. That is, literals appearing as heads also appear in the bodies of clauses.

Since a hierarchy of d-rules is equivalent to Horn programs, to learn a hierarchy of d-rules, we need a way to learn Horn programs. As far as we know, excepting the work of Arimura4) which learns Horn programs, most ILP methods that guarantee correctness are directed towards learning Horn definitions. In Arimura's work, among other restrictions, the clauses in a Horn program are required to be "simple"—that is, only the terms in the head of a clause can appear in the body of the clause. This is too restrictive for our purposes. Instead, we use the Horn-definition learning algorithm to learn Horn programs.

In Horn programs, since the head literal of a clause can appear in the body of another clause, the resulting interactions between clauses make Lemma 9 inapplicable. Hence, the Horn-definition learner cannot be used directly to learn Horn programs. However, if clauses for each head literal are learned separately, assuming that the other clauses are known, we can use the Horn-definition learner. In our implementation, we learned d-rules for each goal separately—i.e., assuming that d-rules for lower-level (sub)goals are already known.

[1] Experimental Results

The algorithm discussed so far has been implemented in Common Lisp as a system called ExEL. It has been tested using two domains—a variation of STRIPS world and a simplified air-traffic control domain.

For the purpose of experiments, the teacher is implemented by providing a set of target d-rules for each domain. Training and test problems are randomly chosen. Each training problem and its solution make several training examples for learning, for a problem may require several applications of a d-rule. Similarly a test problem tests several learned d-rules. The membership queries are answered by syntactic match with the teacher's d-rules.

STRIPS World (SW). This is a minor variation of the STRIPS-world domain[13]; the configuration of the rooms in this domain is a grid, whereas in the standard STRIPS world it could be of any shape. The domain consists of rooms which are connected to each other by doors. The doors can be open or closed. Each room has zero or more boxes. There is a robot that can open doors, move around and push a box from room to room. A typical goal for the robot is to move a box that is in some room to some other room. This goal required six d-rules. It has subgoals such as moving from one room to another room, holding a box, releasing a box, opening a door, and closing a door. The subgoal for moving from one room to another room had six d-rules, and the rest of the subgoals had one d-rule each. Eight of these 16 d-rules are recursive. The Horn-learn algorithm is not applicable for recursive Horn definitions—because Lemma 3.1 is not valid for recursive definitions. However, since the queries are answered by syntactic match with the teacher's d-rules in the implementations, the need for Lemma 3.1 is obviated.[*3] In other words, our membership query is answered *yes* if and only if there is a single target clause that subsumes the given hypothesis clause, and no attempt is made to prove it by chaining different instances of target clauses.

The configuration has 12 rooms (4×3 grid) with 17 doors connecting them. There are also 5 to 10 boxes distributed among the 12 rooms. The robot is placed randomly in one of the rooms. Fig. 2 plots the number of training problems and their solutions versus percentage of test problems ExEL successfully solves in SW. Each data point is a mean of 5 runs. The error bars show one standard deviation on either side of the mean for these runs. The d-rules generated were useful for configurations other than the 4×3 grid ExEL learned from.

Air-Traffic Control (ATC) domain. This domain is a simplified version of Kanfer-Ackerman air-traffic control task.[1] The main task is to land a plane from any configuration. The task has a queue of incoming planes, holding patterns and runways. The planes are accepted into the holding patterns, and then are landed on the runways. The type of a plane, the current wind speed and direction, and the runway conditions impose restrictions on landing a plane. The operators select a plane to land, deposit a plane either on a runway or in a holding position, or move the cursor on the screen. There are 13 d-rules for this domain, including multiple d-rules for some goals. The main goal to land a plane from any holding pattern required three d-rules. Its subgoal to take the plane to the correct runway required four d-rules. Depositing a plane on a runway needed three d-rules. Moving a plane between holding patterns, and its subgoals moving the cursor and selecting a plane all required one d-rule each. Fig. 2 plots the number of training problems and their solutions versus percentage of test problems ExEL successfully solves in ATC. Each data point is a mean of

[*3] For more details on this, see the discussion on self-testing in Section §5.

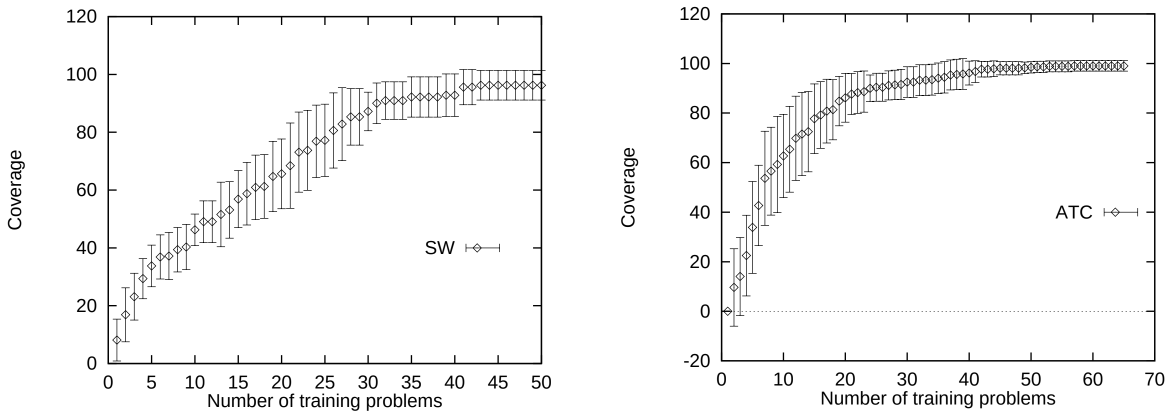

Fig. 2 Performance of ExEL in SW & ATC domains

20 runs. The error bars show one standard deviation on either side of the mean for these runs.

There are a few other systems that learn control knowledge for planning using ILP methods, such as DOLPHIN[34], Grasshopper[21] and SCOPE.[12] One crucial difference between the above learning systems and ExEL is in the form of control knowledge being learned. They learn control rules that choose among various operators to apply in solving a planning problem, whereas ExEL learns d-rules that transform a planning problem into a hierarchical one and specify rules on how to decompose hierarchies. Because of this difference, we cannot compare these systems against ExEL in a meaningful way. Also, these systems embed FOIL[28] for inducing control rules. FOIL, as well as another ILP system GOLEM[23], require negative examples in batch, instead of queries. Such negative examples are difficult to obtain in our setting. However, if we substitute queries by self-testing (see Section §5), the test examples generated can be thought of as negative examples, but they are of "near-miss" kind—that is, they are generated to test a particular hypothesis d-rule learner has constructed.

§5 Discussion

In this work, we have shown that first-order non-recursive Horn definitions are learnable utilizing reasonable amount of resources and queries. As a special case, it follows that first-order monotone DNF is PAC-learnable using membership queries. A generalization of the class of Horn definitions is Horn sentences (or Horn programs). Horn sentences allow different predicate symbols for head literals in a set of Horn clauses, as opposed to a single predicate symbol as in Horn definitions. In fact, as mentioned in Section 4.2, our d-rules are equivalent to first-order Horn programs. Angluin, Frazier and Pitt have shown that propositional Horn sentences are exactly learnable from equivalence and membership queries.[3] They use the model of *learning from interpretations*, where positive examples are models of the target, and negative examples are negative interpretations. We used the learning from entailment model studied by Frazier and Pitt.[15] When membership queries are available, learning from interpretations reduces to learning from entailment. This is because we can convert every negative interpretation of the target into a a positive example in the entailment model (a Horn clause entailed by the target). Since every negative interpretation violates some Horn clause, we can prune all but one negative literal in the interpretation, while making sure that the result is still a negative interpretation. We can then convert that reduced negative interpretation into a positive Horn clause by negating it. Hence, our results imply that Horn definitions are learnable from interpretations as well.

Learnability of first-order Horn sentences from entailment queries is an important open question. Arimura[4)] showed that acyclic and simple Horn programs with predicates of fixed arity are learnable from entailment using equivalence and membership queries. The acyclicity constraint implies that the literals in the language can be ordered such that the derivation of any literal depends only on the literals which occur before it. Elsewhere[31)], we have shown that first-order acyclic Horn sentences that have polynomial-time subsumption algorithms are exactly learnable. As in the case of Arimura, we too use entailment membership queries and assume that the derivation order of literals is known. Unlike Arimura's case, we do not require that the clauses be simple, but require that they be "non-generative"—only the terms that appear in the body of a clause can appear in the head of a clause.

Most of the positive results for PAC-learnability in ILP, so far, depended either on polynomial-time transformations of first-order clauses to propositional logic and propositional PAC-learning algorithms[10, 5, 26)], or on exhaustively enumerating polynomially sized hypothesis spaces.[14)] Therefore, the classes considered were very restrictive. In comparison, the hypothesis space for the language class we considered is unbounded. Because the arity of the predicates is not constant, converting the first order target to a propositional one yields an exponentially large target, which requires exponentially large number of equivalence queries to learn it. The negative results by Cohen[7)] for PAC-learning interesting classes suggest that membership queries are necessary. Along with our work, the work by Page[25)] and the work by Haussler[17)] are significant efforts in making learning tractable using membership and subset queries.

The algorithm in Fig. 1 is similar in spirit to an ILP system called CLINT[9)] in the sense that they both are incremental and interactive. Like in our algorithm, CLINT uses queries to eliminate irrelevant literals. CLINT raises the generality of hypotheses by proposing more complex hypothesis clauses, whereas our algorithm uses lgg.

Several pieces of research have used the lgg idea in different ways for the purpose of generalization. Haussler[17)] considers learning multiple conjunctive concepts. In the hypothesis language considered by him where the number of objects per scene is fixed, lgg of two hypotheses is at most as big as one of the hypotheses, but is not unique. Haussler uses queries to select an lgg which is in the target. On the other hand, in our case, lgg is unique, but its size grows exponentially in the number of hypotheses of which it is lgg. We use queries to reduce the size of the hypothesis generated by the *lgg* (see Generalize procedure in Fig. 1). Frazier and Pitt[16] also use a pruning procedure similar to our Generalize procedure to limit the size of *lgg* in CLASSIC learning. GOLEM[23] is another system that uses *lgg* to generalize hypotheses. GOLEM mitigates the problem of combinatorial explosion due to *lgg* in two ways: (1) by restricting the hypothesis language to *ij*-determinate Horn clauses which guarantee polynomial-sized *lgg*; and (2) by using negative examples to eliminate literals from the hypotheses. In the case of the work by Page[25], the simplicity and the fixed-arity restrictions make the size of *lgg* polynomial in the sizes of the hypotheses being generalized.

The use of queries assumes that a teacher who knows the target concept is available. Requiring such a teacher in practice is unreasonably demanding. This presents a dilemma—since, as mentioned in earlier, without queries only a restricted classes are polynomially PAC-learnable. Elsewhere[29, 30], we present a practical alternative for membership queries, called *self-testing.* In self-testing, the d-rule learner automatically answers the queries by generating planning problems that must use the hypothesized d-rules, and then trying out these d-rules in solving the generated problems. If the generated problems can be solved using the hypothesized d-rules, then the queries are answered yes; otherwise, the queries fail. This solution is practical if we restrict ourselves to learning a subclass of Horn definitions for which subsumption is tractable—such as determinate, k-free or k-local Horn definitions.[18] Determinate Horn definitions are not PAC-learnable without queries, under cryptographic assumptions.[5] Thus, even when limiting learning to languages for which subsumption is tractable, this method is strictly more general.

For self-testing to be usable when learning d-rules for several goals (equivalent to learning Horn programs) or learning recursive d-rules (equivalent to learning recursive Horn definitions), the learner should know how to solve the the subgoals addressed in a hypothesized d-rule. That means that the learner should have known d-rules for those subgoals. Thus, learning has to be ordered such that d-rules for subgoals appearing in a particular d-rule are learned before learning that d-rule. One way to order is following goal-subgoal hierarchies in the domain. In case of recursive d-rules for a goal (intra-goal), this, however, does not work. A solution we adopted in our work[29, 30] is learning according to the number of recursive "calls" to a goal in a d-rule. The base-case d-rules (the ones with zero recursive calls) are learned first. Then the ones with one recursive call are learned next, and so forth. The idea is that while learning d-rules with one recursive call, base-case d-rules can be used by the self-testing method, and so on.

This work is one of the first to theoretically demonstrate that the first-order learning is more powerful than propositional learning. We believe that structural domains such as planning and scheduling are inherently relational and are better suited to first-order learning. However, to build practical systems, we need to generalize our results in many directions. These include learning more general Horn sentences, handling noise, probabilities and real numbers, and incorporating background theory.

Acknowledgment 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 this special issue and of ILP-97 for their insightful comments.

References

1) Ackerman, P., Kanfer, R., *Kanfer-Ackerman Air Traffic Control Task© CD-ROM Database, Data-Collection Program, and Playback Program*, Dept. of Psychology, Univ. of Minn., Minneapolis, MN, 1993.

2) Angluin, D., "Queries and Concept Learning," *Machine Learning 2*, 319–342, 1988.

3) Angluin, D., Frazier, M., Pitt, L., "Learning conjunctions of Horn clauses," *Machine Learning 9*, 147–164, 1992.

4) Arimura, H., "Learning Acyclic First-order Horn Sentences From Entailment," in *Proceedings of the Eigth International Workshop on Algorithmic Learning Theory*. Ohmsha/Springer-Verlag, 1997.

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

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

7) Cohen, W., "Pac-learning recursive logic programs: negative results," *Jl. of AI Research 2*, 541–573, 1995.

8) De Raedt, L., "Logical Settings for Concept Learning," *Artificial Intelligence 95*, 1, 187–201, 1997.

9) De Raedt, L., Bruynooghe, M., "Interactive concept learning and constructive induction by analogy," *Machine Learning 8*, 2, 107–150, 1992.

10) Džeroski, S., Muggleton, S., Russell, S., "PAC-learnability of Determinate Logic Programs," in *Proceedings of the Fifth Annual ACM Workshop on Computational Learning Theory*, (pp. 128–135), 1992.

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

12) Estlin, T., Mooney, R., "Multi-strategy Learning of Search Control for Partial-Order Planning," in *Proceedings of the Thirteenth National Conference on Artificial Intelligence (AAAI-96)*, (pp. 843–848). AAAI/MIT Press, 1996.

13) Fikes, R., Hart, P., Nilsson, N., "Learning and executing generalized robot plans," *Artificial Intelligence 3*, 251–288, 1972.

14) Frazier, M., Page, C. D., "Learnability in Inductive Logic Programming: Some Basic Results and Techniques," in *Proceedings of the Eleventh National Conference on Artificial Intelligence (AAAI-93)*, (pp. 120–127). AAAI Press, 1993.

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

16) Frazier, M., Pitt, L., "CLASSIC learning," *Machine Learning 25*, 151–194, 1995.

17) Haussler, D., "Learning Conjunctive Concepts in Structural Domains," *Machine Learning 4*, 7–40, 1989.

18) Kietz, J.-U., Lübbe, M., "An efficient subsumption algorithm for inductive logic programming," in *Proceedings of the Eleventh International Conference on Machine Learning*, (pp. 130–138), 1994.

19) Kowalski, R., "The case for using equality axioms in automatic demonstration," in *Lecture Notes in Mathematics*, volume 125, Springer-Verlag, 1970.

20) Lassez, J.-L., Maher, M., Marriott, K., "Unification revisited," in J. Minker (ed.), *Foundations of Deductive Databases and Logic Programming*, Morgan Kaufmann, 1988.

21) Leckie, C., Zukerman, I., "An inductive approach to learning search control rules for planning," in *Proceedings of the 13th IJCAI*, (pp. 1100–1105), 1993.

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

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

24) Nienhuys-Cheng, S.-H., de Wolf, R., "Least Generalizations and Greatest Specializations of Sets of Clauses," *Jl. of AI Research 4*, 341–363, 1996.

25) Page, C. D., *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, 1993.

26) Page, C. D., Frisch, A. M., "Generalization and Learnability: A Study of Constrained Atoms," in S. H. Muggleton (ed.), *Inductive Logic Programming*, (pp. 29–61), Academic Press, 1992.

27) Plotkin, G., "A Note on Inductive Generalization," in B. Meltzer, D. Michie (eds.), *Machine Intelligence*, volume 5, (pp. 153–163), Elsevier North-Holland, New York, 1970.

28) Quinlan, J., "Learning logical definitions of from relations," *Machine Learning 5*, 239–266, 1990.

29) Reddy, C., Tadepalli, P., "Inductive logic programming for speedup learning," in L. DeRaedt, S. Muggleton (eds.), *Proceedings of the IJCAI-97 workshop on Frontiers of Inductive Logic Programming*, 1997.

30) Reddy, C., Tadepalli, P., "Learning Goal-Decomposition Rules using Exercises," in *Proceedings of the 14th International Conference on Machine Learning*. Morgan Kaufmann, 1997.

31) Reddy, C., Tadepalli, P., "Learning First-Order Acyclic Horn Programs from Entailment," in *Proceedings of the 15th International Conference on Machine Learning; (and Proceedings of the 8th International Conference on Inductive Logic Programming)*. Morgan Kaufmann, 1998.

32) Reddy, C., Tadepalli, P., Roncagliolo, S., "Theory-Guided Empirical Speedup Learning of Goal-Decomposition Rules," in *Proceedings of the 13th International Conference on Machine Learning*, (pp. 409–417). Morgan Kaufmann, 1996.

33) Schapire, R., "The strength of weak learnability," *Machine Learning 5*, 197–227, 1990.

34) Zelle, J., Mooney, R., "Combining FOIL and EBG to speedup logic programs," in *Proceedings of the 13th IJCAI*, (pp. 1106–1111), 1993.