Learning Propositional Horn Sentences With Hints

Dana Angluin*, Yale University

YALEU/DCS/RR-590
December 1987

*Supported by the National Science Foundation, IRI-8404226

Learning Propositional Horn Sentences With Hints

Dana Angluin *
Yale University

December 1987

Abstract

We demonstrate a simple polynomial time algorithm for learning positive acyclic propositional Horn sentences using equivalence queries and a new type of query called a "request for a hint". The basic algorithm is modified to be incremental, and to handle the non-positive case. Suggestions for the cyclic case are offered. For the special case of positive Horn sentences with two literals per clause, the learning problem is recast as a problem of diagnosing "errors" in directed graphs.

1 Introduction

This paper is part of an effort to use reasonable kinds of additional information to construct efficient learning algorithms. In [5] we introduced a classification of types of queries for concept learning consisting of membership, equivalence, subset, superset, disjointness, and exhaustiveness queries. It is an open problem whether there is a polynomial time algorithm to learn concepts described by propositional Horn sentences, even if all of these types of queries are available. In this paper we show that there is a simple polynomial time algorithm to learn acyclic propositional Horn sentences if an additional type of information, called a "hint", is available.

A hint consists of an intermediate point in a derivation using the unknown sentence. It may be interpreted as a partial explanation of why the result of the derivation follows from the assumptions of the theory. The notion of a hint is described and defended at greater length below.

We also argue that it is reasonable to let positive and negative examples be clauses rather than truth-value assignments. In practice certain clauses are "meaningless"; we show how our learning algorithm avoids them. In the case of positive Horn sentences with exactly two literals per clause, we recast the problem as one of diagnosing "errors" in directed graphs. For acyclic directed graphs there is an efficient debugging algorithm, described in [1].

*Supported by NSF grant IRI-8404226

2 Preliminaries

2.1 Propositional Horn sentences

Let V be a finite set of variables. A *literal* is an element of V or the negation of an element of V. The negation of a is denoted $\neg a$.

A Horn clause is a disjunction of a set of literals that contains at most one positive literal. We will use the term "clause" to mean Horn clause throughout the paper. A Horn clause that contains a positive literal is called *positive*. A Horn clause that contains no positive literal is called *negative*.

A Horn sentence is a conjunction of a set of Horn clauses. A Horn sentence is *positive* if and only if it contains only positive clauses. The *size* of a Horn sentence is the number of occurrences of literals in the sentence.

If ϕ and ϕ' are Horn sentences, we use $\phi \vdash \phi'$ to mean that ϕ logically implies ϕ', that is, there is no assignment of truth values to the variables V such that ϕ is true and ϕ' is false.

The following is a Horn sentence over the variables $\{a, b, c, d, e\}$:

$$(a) \wedge (\neg a \vee \neg b \vee c) \wedge (\neg c \vee d) \wedge (\neg a \vee \neg d \vee e) \wedge (\neg d \vee \neg e).$$

Note that this sentence is not positive, since it contains the negative clause $(\neg d \vee \neg e)$. The size of this sentence is 11.

By the usual rules of logic, the clause $(\neg a \vee \neg b \vee c)$ is equivalent to the implication $(a \wedge b \rightarrow c)$. It is convenient to represent all the clauses as implications, so we introduce the constant symbol $\top$ for the value true, and $\bot$ for the value false. Then the clause (a) may be rewritten as $(\top \rightarrow a)$, and $(\neg d \vee \neg e)$ may be rewritten as $(d \wedge e \rightarrow \bot)$.

Thus we can rewrite the above formula as

$$(\top \rightarrow a) \wedge (a \wedge b \rightarrow c) \wedge (c \rightarrow d) \wedge (a \wedge d \rightarrow e) \wedge (d \wedge e \rightarrow \bot).$$

A positive Horn sentence can be rewritten as implications without using $\bot$.

If C is a Horn clause, the *antecedents* of C is the set of variables that occur negated in C. If C is a positive Horn clause, the *consequent* of C is the (unique) variable that occurs unnegated in C. In the clause $(a \wedge b \rightarrow c)$, the *antecedents* are the variables a and b, and the *consequent* is the variable c. In the clause $(\top \rightarrow a)$ the antecedents are the empty set and the consequent is a. If A is a set of variables and z is a variable, we denote the Horn clause with antecedents A and consequent z by $(A \rightarrow z)$.

We define the *graph* of a Horn sentence ϕ to be a directed graph whose nodes are the variables appearing in ϕ. There is an edge from variable a to variable b in the graph if and only if there is a clause of ϕ in which a is among the antecedents, and b is the consequent. A Horn sentence is *acyclic* if its graph of is acyclic, that is, contains no directed cycles.

If ϕ is an acyclic Horn sentence, the topological ordering of its graph is the ordering in which a *precedes* b if and only if there is a directed path of one or more edges from a to b in the graph. This ordering is transitive, anti-reflexive, and anti-symmetric. The *depth* of an acyclic Horn sentence is the maximum number of edges in any directed path in its graph.

The graph of the sentence

$$(\top \rightarrow a) \wedge (a \wedge b \rightarrow c) \wedge (c \rightarrow d) \wedge (a \wedge d \rightarrow e) \wedge (d \wedge e \rightarrow \bot).$$

has the edges

$$(a,c),(b,c),(c,d),(a,e),(d,e)$$

and is acyclic. In this graph, a precedes c, d, and e, but neither a nor b precedes the other. The depth of this sentence is 3; the directed path from a to c to d to e has three edges.

2.2 Molly's problem

What good are propositional Horn sentences? They are a toy model of the language Prolog, whose pure part consists of Horn clauses in the predicate logic. They are computationally more tractable than general conjunctive normal form propositional sentences, in that there are polynomial time sequential algorithms to decide the satisfiability and equivalence problems. (However, the satisfiability problem is log-space complete for polynomial time, so there is unlikely to be a poly-log parallel time algorithm for either of these problems [20].)

We give an example of the use of propositional Horn sentences. Imagine that you have become acquainted with an alien named Molly from the planet Orand, who is currently employed in a daycare center. She is quite good at propositional logic, but a bit weak on knowledge of Earth. So you decide to formulate the beginnings of a propositional theory to help her label things in her immediate environment. The clauses you devise are as follows.

1. two_wheels $\wedge$ one_seat $\wedge$ pedals $\rightarrow$ bike.
2. two_wheels $\wedge$ no_seat $\rightarrow$ scooter.
3. three_wheels $\wedge$ one_seat $\wedge$ pedals $\rightarrow$ trike.
4. four_wheels $\wedge$ doors $\wedge$ few_seats $\rightarrow$ car.
5. four_wheels $\wedge$ doors $\wedge$ many_seats $\rightarrow$ bus.
6. four_wheels $\wedge$ handle $\wedge$ no_seats $\rightarrow$ wagon.
7. scooter $\rightarrow$ toy.
8. wagon $\rightarrow$ toy.
9. trike $\rightarrow$ toy.
10. bike $\wedge$ small_size $\rightarrow$ toy.
11. bike $\wedge$ medium_size $\rightarrow$ vehicle.
12. car $\rightarrow$ vehicle.
13. bus $\rightarrow$ vehicle.
14. toy $\rightarrow$ object.
15. vehicle $\rightarrow$ object.

The conjunction of all these clauses is a Horn sentence, say ϕ_w. It is acyclic and positive, and has size 43 and depth 3. One clause logically implied by ϕ_w is

$$(\text{two\_wheels} \wedge \text{one\_seat} \wedge \text{pedals} \wedge \text{small\_size} \rightarrow \text{toy}).$$

A clause that is not logically implied by ϕ_w is

$$(\text{two\_wheels} \wedge \text{one\_seat} \rightarrow \text{toy}).$$

If you find that Molly calls motorcycles "toys", it might be because her incorrect version of the theory implies this clause.

It seems reasonable to present positive and negative examples of the theory ϕ_w in terms of clauses implied by and not implied by the theory, rather than by truth-value assignments that do or do not satisfy the sentence ϕ_w. A further discussion of this issue appears in Appendices A and B.

This little theory also illustrates the issue of "meaningfulness". Suppose at some point in acquiring the theory ϕ_w, Molly asks you whether the clause

$$(\text{two\_wheels} \wedge \text{three\_wheels} \rightarrow \text{toy})$$

is implied by the correct theory. Of course the correct answer is "no", but such queries are unsettling. The reason is that the conjunction of the antecedents,

$$(\text{two\_wheels} \wedge \text{three\_wheels}),$$

is in practice always false, though nothing in the logical theory presented captures that.

In the abstract treatment we assume that the structure of "meaningful" situations is specified by a collection of those sets of variables that can be simultaneously true in the intended semantic domain. Given some reasonable conditions on that collection, we show that our learning algorithm will stick to "meaningful" queries.

2.3 Deriving clauses from Horn sentences

We need a fairly detailed understanding of derivations.

The procedure *Forward-Chain*(ϕ, A)

Forward-Chain takes as input a positive Horn sentence ϕ and a set of variables A. The output is the set of those variables x such that ϕ logically implies the clause $(A \rightarrow x)$.

1. Initialize $T = A$.
2. If there is any clause C' of ϕ whose antecedents are all in T and whose consequent is not, add the consequent of C' to T and repeat this step.
3. Otherwise, stop and output T.

Lemma 1 *Let ϕ be any positive Horn sentence and let $C = (A \rightarrow z)$ be any positive Horn clause. Then $\phi \vdash C$ if and only if z is in the set output by Forward-Chain on inputs ϕ and A.*

We first show by induction that every x placed in T has the property that ϕ logically implies the clause $(A \rightarrow x)$. That is clearly true after the initialization of T, when it contains just the variables of A.

Suppose it is true just before the variable x is added to T. Then there is a clause C' in ϕ whose antecedents are in T and whose consequent is x. Thus, whenever ϕ and the variables in A are true, all the variables in T must be true, and so x must be true as well. Thus, ϕ logically implies the clause $(A \rightarrow x)$, which completes the induction.

Conversely, let X denote the set of variables output when *Forward-Chain* is run with inputs ϕ and A, and suppose $z \notin X$. Consider the truth-value assignment that assigns true to all the variables in X and false to all other variables in V. This assignment assigns true to all the variables of A, and false to z, so it assigns false to the clause C. If C' is any clause of ϕ, then either all its antecedents are assigned true and its consequent is assigned true, or some of its antecedents are assigned false. Thus, every clause of ϕ is assigned the value true by this assignment, so ϕ is also assigned true. Thus $\phi \not\vdash C$, since this assignment makes ϕ true and C false. Q.E.D.

It is clear that *Forward-Chain* can be implemented in a straightforward way to run in time polynomial in the size of ϕ and $|A|$. A modification of this procedure takes a clause C logically implied by ϕ and constructs a *derivation-DAG* of C from ϕ, defined next.

A *derivation-DAG* of a clause $C = (A \rightarrow z)$ from ϕ is a directed acyclic graph that satisfies the following properties. Its nodes are a subset of V. The variable z is the unique node of out-degree zero, called the *sink*. For every node x, there is a directed path from x to z. The nodes of in-degree zero, called the *sources*, are a subset of A. For every non-source node x, if the set of nodes with edges into x is S then the clause $(S \rightarrow x)$ is in ϕ.

Note that for every node x in a derivation-DAG of C from ϕ, ϕ logically implies the clause $(A \rightarrow x)$. The condition that there be a directed path from any node x in the graph to z means that all the intermediate nodes are actually used in the derivation of C.

Lemma 2 *Suppose ϕ is a positive acyclic propositional Horn sentence, and C a clause such that $\phi \vdash C$. Let D be a derivation-DAG of C from ϕ. If there is a directed path of length $k \geq 1$ from x to y in D, then there is a directed path of length k from x to y in the graph of ϕ.*

This is proved by induction on the length of the path in D. If the path is of length one, then there is a clause of ϕ with x in the set of antecedents and y as the consequent, so there is an edge from x to y in the graph of ϕ.

Assume that for any pair x and y such that there is a path of length $n \geq 1$ from x to y in D, there is a path of length n from x to y in the graph of ϕ. Suppose that there is a path of length $n+1$ from x to y in D.

In D there must be a node w such that there is a path of length n from x to w and an edge from w to y. By the inductive assumption, there is a path of length n from x to w in the graph of ϕ. Also there must be a clause of ϕ such that w is in the set of antecedents and the consequent is y. Thus, there is an edge from w to y in the graph of ϕ, and a path of length $n+1$ from x to y in the graph of ϕ, which completes the induction. Q.E.D.

2.4 M-equivalence

We now introduce a modification of the notion of logical equivalence, to be used in formally specifying "meaningfulness". Let M be any collection of subsets of the set V of variables. A clause C is defined to be *M-meaningful* if and only if the set of antecedents of C is in M. A clause is *M-meaningless* if it is not M-meaningful.

Two Horn sentences ϕ_1 and ϕ_2 are defined to be *M-equivalent* if and only if for every M-meaningful clause C, $\phi_1 \vdash C$ if and only if $\phi_2 \vdash C$.

Clearly, if ϕ_1 and ϕ_2 are logically equivalent, then they are M-equivalent for any M. If M is the set of all subsets of V and ϕ_1 and ϕ_2 are M-equivalent, then they are logically equivalent. If M does not contain all subsets of V, then M-equivalence does not imply logical equivalence, as the following example illustrates.

Let

$$M = \{\emptyset, \{a\}, \{b\}, \{c\}, \{a, b\}\}.$$

Then $(a \rightarrow b)$ is M-equivalent to $(a \rightarrow b) \wedge (b \wedge c \rightarrow a)$, although they are not logically equivalent.

2.5 Meaning structures

We now describe the additional conditions a meaning structure must meet. A meaning structure is defined with respect to a positive propositional Horn sentence ϕ.

Let M be a collection of subsets of the set V of variables. M is a *meaning structure* for ϕ if and only if it is closed under subset and implication with respect to ϕ. That is, if $S \in M$ and S' is a subset of S, then $S' \in M$. And if $S \in M$ and there is a clause C implied by ϕ such that the antecedents of C are in S, then $(S \cup \{x\}) \in M$, where x is the consequent of C.

The intuition is that M specifies all the sets of variables that can be simultaneously true in practice. Then the first condition just says that a subset of a set of variables that can be simultaneously true can also be simultaneously true. The second condition says that if all the antecedents of a clause implied by ϕ can be simultaneously true, then they can be simultaneously true with the consequent.

In the example of Molly's problem, the meaning structure would contain all the subsets of {two_wheels, pedals, one_seat, small_size, bike, toy, object}, together with other sets, but would not contain any superset of {two_wheels, three_wheels}.

Lemma 3 *Let ϕ be a positive propositional Horn sentence and M be a meaning structure for ϕ. If ϕ' is obtained from ϕ by removing all the M-meaningless clauses, then ϕ and ϕ' are M-equivalent.*

Clearly ϕ implies ϕ', so assume that $\phi \vdash C$ for some M-meaningful clause $C = (A \rightarrow z)$. Then there is a derivation-DAG D for C from ϕ. We argue that each of the clauses of ϕ used in D is M-meaningful, and hence in ϕ'.

We argue by induction on the topological ordering of D that the set of all its nodes is M-meaningful. The sources are a subset of A, which is M-meaningful because C is a meaningful clause and M is closed under subset.

Inductively assume S is the set of all nodes preceding x in the topological ordering of D, and assume that S is M-meaningful. Then there is a clause C' of ϕ such that the consequent of C' is x and the antecedents of C' are contained in S, so by closure of M under implication with respect to ϕ, $S \cup \{x\}$ is also in M.

This completes the induction and shows that every clause used in D is M-meaningful, and hence in ϕ'. Thus D is also a derivation-DAG of C from ϕ', so $\phi' \vdash C$, as claimed. Thus, for every M-meaningful clause, $\phi \vdash C$ if and only if $\phi' \vdash C$, so ϕ and ϕ' are M-equivalent. Q.E.D.

3 Formalization of the learning problem

We assume that there is an unknown propositional Horn sentence ϕ_* to be learned using the information available from two types of queries: M-equivalence queries and requests for a hint. We assume initially that ϕ_* is acyclic and positive, and show how to remove these restrictions later.

We assume also that there is a fixed meaning structure M associated with the sentence ϕ_* to be learned. The meaning structure is unknown to the learning algorithm. The goal of the learning algorithm is to output a sentence ϕ that is M-equivalent to the unknown sentence ϕ_*.

3.1 The two types of queries

M-equivalence queries. The first type of query is an *M-equivalence* query. The learning algorithm proposes a positive Horn sentence ϕ. If ϕ is M-equivalent to ϕ_*, the answer to this query is *yes*. Otherwise, the answer is *no*. In this case the answer also contains a *counterexample*, that is, an M-meaningful clause C that is implied by ϕ_* and not by ϕ or vice versa. If C is implied by ϕ_*, it is called a *positive* counterexample. If C is implied by ϕ, it is called a *negative* counterexample. The choice of counterexample is arbitrary; the learning algorithm must work no matter what counterexample is returned.

Request for hint queries. The second type of query is a *request for a hint*. In this case, the learning algorithm proposes a positive clause $C = (A \rightarrow z)$. If C is M-meaningful, then the query is answered as follows. If ϕ_* does not logically imply C, the reply is simply *no*. If there is a clause C' of ϕ_* such that its consequent is z and its antecedents are a subset of A, the reply is *one step*. Finally, if neither of these conditions obtains, the reply is a variable x which is neither in A nor equal to z and which occurs in some derivation-DAG for C from ϕ_*. In this last case, the variable x is called the *hint* returned by the query. The choice of hint is arbitrary; the learning algorithm must work no matter what hint is returned. If the clause C is not M-meaningful, then any answer is legal: *no* or *one step* or any variable from V. Thus the reply is informative only in case C is a meaningful clause.

Comments. An M-equivalence query is simply an equivalence query in the sense of [5], if Horn sentences are taken to denote the set of M-meaningful clauses they imply. However, a request for a hint is a new type of query, intended to evoke a kind of partial explanation of why some consequence z follows from some set of assumptions A in the theory ϕ_*. The

reply *one step* means that the implication follows in one step from the theory. Otherwise, we are given a hint, that is, an intermediate point in the derivation of z from A using ϕ_*.

A discussion of the relationship between equivalence queries and stochastic equivalence may be found in [5]. Littlestone [26] shows the close relationship between bounds on equivalence queries and errors of prediction.

4 The learning algorithm

4.1 A brute-force method

One simple approach to learning in this situation, which dispenses with the need for requests for hints, is the following. Initially set the current hypothesis to the empty sentence, $\top$. Do an M-equivalence query with the current hypothesis. If the answer is *yes*, stop and output the current hypothesis. Otherwise, there will be a positive counterexample C; conjoin C to the current hypothesis and repeat. However, it is shown in [5] that such an approach may take time exponential in the size of the unknown sentence ϕ_*.

Moreover, we show in Appendix B that if A is any algorithm that learns Horn sentences from equivalence queries (with clauses as counterexamples), then A may be transformed with little overhead to predict acyclic circuits. This makes it unlikely that any such A can run in polynomial time.

4.2 The learning algorithm *HL*

The procedure we now describe, *HL*, uses M-equivalence queries and requests for hints to find a sentence M-equivalent to ϕ_* in time polynomial in $|V|$ and the size of ϕ_*. We begin by describing two useful subprocedures.

The procedure *Reduce*(C)

The input is an M-meaningful clause $C = (A \rightarrow z)$ such that a request for a hint with C as input returns the answer *one step*. The output is a subset of C that is equal to a clause of ϕ_*. The method is a greedy search for a minimal subset of the antecedents of C which preserves the condition that the clause be derivable in one step using ϕ_*.

1. Let $T = A$.
2. While there exists an element $x \in T$ such that a request for a hint query with the clause $((T - \{x\}) \rightarrow z)$ is answered *one step*, set $T = T - \{x\}$.
3. Return the clause $(T \rightarrow z)$.

Lemma 4 *Suppose the input to Reduce is an M-meaningful clause $C = (A \rightarrow z)$ such that a request for a hint with C as input returns the answer one step. Then the output of Reduce is an M-meaningful clause $C' = (A' \rightarrow z)$ such that A' is a subset of A and C' is a clause of ϕ_*. Reduce runs in time bounded by a polynomial in the size of C, and every request for a hint query it makes is with an M-meaningful clause.*

T is initially A and remains a subset of A. Since C is M-meaningful and M is closed under subset, every request for a hint is with an M-meaningful clause. Thus every request for a hint is answered correctly.

Since every element of A is removed from T at most once, this procedure must terminate after at most $|A|$ iterations. The value returned is an M-meaningful clause $C' = (T \rightarrow z)$ such that a request for a hint with this clause as input is answered *one step*, and if T' is obtained from T by removing any element, then a request for a hint with $(T' \rightarrow z)$ is not answered *one step*. Thus C' is equal to a clause of ϕ_*. A straightforward implementation of *Reduce* clearly runs in time polynomial in the size of C. Q.E.D.

The procedure *Find-Missing*(C)

The procedure *Find-Missing* takes as input an M-meaningful clause $C = (A \rightarrow z)$ that is implied by ϕ_* but not by ϕ, and returns an M-meaningful clause C' that is in ϕ_* but not implied by ϕ.

1. Make a request for a hint with the clause C. If the reply is *one step*, then return *Reduce*(C).
2. Otherwise, if the reply is a hint x, call *Forward-Chain*(ϕ, A), and let X denote the set of variables returned.
3. If x is not in X, then let $C' = (A \rightarrow x)$, and return *Find-Missing*(C').
4. If x is in X, then let $C' = ((A \cup \{x\}) \rightarrow z)$, and return *Find-Missing*(C').

Lemma 5 *If the procedure Find-Missing is called with an M-meaningful clause C such that ϕ_* implies C and ϕ does not imply C, then it will return a clause C' in ϕ_* that is not implied by ϕ. Find-Missing runs in time bounded by a polynomial in $|V|$ and the size of ϕ. Every request for a hint query made by Find-Missing is with an M-meaningful clause.*

Assume that *Find-Missing* is called with an M-meaningful clause $C = (A \rightarrow z)$ such that $\phi_* \vdash C$ and $\phi \nvdash C$. Thus A is in M. We show that if there is a recursive call, it is with an M-meaningful clause C' such that $\phi_* \vdash C$ and $\phi \nvdash C$.

Suppose the hint x is returned by the request for a hint in step (1). We know that ϕ_* implies the clause $(A \rightarrow x)$. Since A is in M, $A \cup \{x\}$ is in M by closure under implication with respect to ϕ_*. Thus, both A and $A \cup \{x\}$ are in M, so a recursive call in step (3) or step (4) must be with an M-meaningful clause C'.

If a recursive call occurs from step (3), it is with the clause $C' = (A \rightarrow x)$. Clearly $\phi_* \vdash C'$. However, $x \notin X$, so by the correctness of the *Forward-Chain* procedure, $\phi \nvdash C'$.

If a recursive call occurs from step (4), then it is with the clause $C' = ((A \cup \{x\}) \rightarrow z)$. Clearly $\phi_* \vdash C'$, since $\phi_* \vdash C$ and $C \vdash C'$. In this case, $x \in X$, which means that ϕ implies the clause $(A \rightarrow x)$. If ϕ also implied C' then ϕ would imply the clause $(A \rightarrow z)$, that is, would imply C, contrary to hypothesis. Thus in this case also, $\phi \nvdash C'$.

Since every recursive call is with an M-meaningful clause, every request for a hint query made by *Find-Missing* is with an M-meaningful clause.

If *Find-Missing* halts, it is with the value *Reduce*(C), where the request for a hint with input C in step (1) returns the value *one step*. By the correctness of the procedure *Reduce*, *Reduce*(C) is a clause C' of ϕ_*. By the input conditions, $\phi \not\vdash C$, so $\phi \not\vdash C'$ (since $C' \vdash C$). Hence if *Find-Missing* terminates, it returns a clause that is in ϕ_* but not implied by ϕ.

We must establish that *Find-Missing* will terminate. Here we finally use the assumption that ϕ_* is acyclic. When a hint x is returned, it is a node in some derivation-DAG of $C = (A \rightarrow z)$ from ϕ_*, so there is a directed path of positive length from x to z in the derivation-DAG. Hence x precedes z in the graph of ϕ_* by Lemma 2. Then x becomes either the consequent of the new clause C' or one of the antecedents of the new clause C'.

In either case, x cannot be returned as a hint again, since the set of antecedents is non-decreasing (with respect to set inclusion) and the consequent is non-increasing (with respect to the topological ordering of the graph of ϕ_*.) Hence each variable can be returned as a hint at most once per top-level call of *Find-Missing*, so a top-level call of *Find-Missing* must terminate after making at most $|V| - 1$ recursive calls.

It is clear that the procedure *Find-Missing* can be implemented to run in time polynomial in $|V|$. Q.E.D.

Note that one call to *Forward-Chain* suffices per top-level call of *Find-Missing*, since the variables added to the antecedents are implied by the antecedents according to ϕ. We now come to the top-level learning algorithm.

The procedure *HL*

1. Initialize $\phi = \top$.
2. Make an M-equivalence query with ϕ. If the reply is *yes*, output ϕ and halt.
3. Otherwise, the reply is an M-meaningful clause C implied by ϕ_* but not by ϕ.
4. Let C' be the clause returned by *Find-Missing*(C), set $\phi = \phi \wedge C'$, and go to step 2.

Theorem 6 *The procedure HL finds a positive Horn sentence ϕ that is M-equivalent to ϕ_* and runs in time polynomial in the size of ϕ_* and the number of variables, $|V|$. Every request for a hint query made by HL is with an M-meaningful clause.*

We first show by induction that ϕ always consists of a conjunction of a subset of the clauses of ϕ_*. This is clear when $\phi = \top$, since $\top$ is the conjunction of no clauses, which is vacuously a subset of the clauses of ϕ_*.

Assume ϕ is a conjunction of a subset of the clauses of ϕ_*, and assume a counterexample C is returned by the M-equivalence query in step (2). Then C must be M-meaningful and implied by ϕ_* but not by ϕ (since $\phi_* \vdash \phi$.) Thus, by the correctness of the procedure *Find-Missing*(C), the clause C' added to ϕ must be in ϕ_* but not in ϕ. Hence ϕ remains a conjunction of a subset of the clauses of ϕ_*.

Once a clause is added to ϕ, it is never removed, so new clauses can be added to ϕ at most r times, where r is the number of clauses in ϕ_*. Hence, *HL* must terminate after at

most r M-equivalence queries and calls to *Find-Missing*, and when it terminates its output must be M-equivalent to ϕ_*.

Since *Find-Missing* runs in time polynomial in $|V|$ and the size of ϕ, and the size of ϕ is bounded above by the size of ϕ_* at all times, *HL* runs in time bounded by a polynomial in $|V|$ and the size of ϕ_*.

Each subprocedure called by *HL* makes request for hint queries with M-meaningful clauses only, so this is also true of *HL*. Q.E.D.

4.3 Example of *HL*

We give an example to illustrate how *HL* and its sub-procedures work. Imagine that Molly is using the algorithm *HL* to acquire the theory ϕ_w from you, and the value of ϕ is currently the conjunction of the following clauses.

1. three_wheels $\wedge$ one_seat $\wedge$ pedals $\rightarrow$ trike.
2. two_wheels $\wedge$ one_seat $\wedge$ pedals $\rightarrow$ bike.
3. trike $\rightarrow$ toy.
4. toy $\rightarrow$ object.

In response to an M-equivalence theory, you propose the following clause as a positive counterexample, that is, implied by ϕ_w but not by ϕ.

$$\text{two\_wheels} \wedge \text{one\_seat} \wedge \text{pedals} \wedge \text{small\_size} \rightarrow \text{object.}$$

The procedure *Find-Missing* is invoked with this clause as input. A request for a hint is made with this clause as input. Since the clause is derived from ϕ_w, but not in one step, you reply with "bike" as an intermediate point in the derivation. The *Forward-Chain* procedure is used to derive from the antecedents

$$\{\text{two\_wheels, one\_seat, pedals, small\_size}\}$$

the consequences

$$\{\text{two\_wheels, one\_seat, pedals, small\_size, bike}\}$$

according to the theory ϕ. Since "bike" is already in this set, there is a request a hint with the clause

$$\text{two\_wheels} \wedge \text{one\_seat} \wedge \text{pedals} \wedge \text{small\_size} \wedge \text{bike} \rightarrow \text{object.}$$

You reply with an intermediate point in the derivation of this clause, for example, "toy". Since this is not in the set returned by *Forward-Chain*, there is a request for a hint with the clause

$$\text{two\_wheels} \wedge \text{one\_seat} \wedge \text{pedals} \wedge \text{small\_size} \wedge \text{bike} \rightarrow \text{toy.}$$

This is derived in one step using ϕ_*, since it is subsumed by the clause

$$\text{bike} \wedge \text{small\_size} \rightarrow \text{toy}.$$

Thus the reply is *one step*, and the sub-procedure *Reduce* is called with the clause

$$\text{two\_wheels} \wedge \text{one\_seat} \wedge \text{pedals} \wedge \text{small\_size} \wedge \text{bike} \rightarrow \text{toy}.$$

The next request for a hint is with the clause

$$\text{one\_seat} \wedge \text{pedals} \wedge \text{small\_size} \wedge \text{bike} \rightarrow \text{toy}.$$

This is answered *one step*.

The process of dropping variables from the antecedents continues until the clause

$$\text{small\_size} \wedge \text{bike} \rightarrow \text{toy}$$

is returned by *Reduce*, then by *Find-Missing*, and finally is conjoined to the theory ϕ.

5 Improvements to *HL*

In this section we discuss various methods to make *HL* more robust and to extend the scope of its applicability.

5.1 Incremental learning

Imagine again that Molly is using the algorithm *HL* to learn the propositional Horn theory ϕ_w from you. Things are going along fine until you suddenly realize that you've made an error in answering her queries, and she has mistakenly acquired the clause

$$(\text{two\_wheels} \wedge \text{one\_seat} \rightarrow \text{toy}).$$

How to get rid of the offending clause? There is no provision for removing clauses in the algorithm *HL*, so you reluctantly tell her to forget all she's learned about ϕ_w and begin the learning process from scratch again.

A realistic learning procedure needs to be incremental in the sense that if it is started with a "nearly" correct theory, the amount of work to get to a correct theory is "small". Hence we add a new procedure to find incorrect clauses, analogous to Shapiro's procedure to find incorrect clauses in Prolog programs [35].

The procedure *Find-Incorrect*(C)

The input is an M-meaningful clause C that is implied by the current hypothesis ϕ, but not by ϕ_*. The output is an M-meaningful clause C' that is in ϕ but is not implied by ϕ_*. The method is to test the clauses in some derivation of C from ϕ one by one until an incorrect clause is found.

1. Find a derivation-DAG D of C from ϕ. Let T be the source nodes of D and G the graph D with its source nodes removed.

2. Remove any source node x from G and let S_x be the set of nodes of D with edges into x.

3. Use a request for a hint query to determine whether $\phi_* \vdash (S_x \rightarrow x)$.

4. If $\phi_* \not\vdash (S_x \rightarrow x)$, then return the clause $(S_x \rightarrow x)$.

5. If $\phi_* \vdash (S_x \rightarrow x)$, then add x to T and continue at step (2) with the next element of G.

Lemma 7 *If C is an M-meaningful clause implied by ϕ but not ϕ_*, then Find-Incorrect(C) returns an M-meaningful clause C' that is in ϕ but not implied by ϕ_*. The procedure runs in time bounded by a polynomial in $|V|$ and the size of ϕ. Every request for a hint query made by Find-Incorrect is with an M-meaningful clause.*

Assume that C is an M-meaningful clause implied by ϕ and not by ϕ_*. A modification of the procedure *Forward-Chain* will find a derivation-DAG D of C from ϕ in time bounded by a polynomial in the $|V|$ and the size of ϕ. D contains at most $|V|$ nodes.

We show by induction that the set T is in M throughout the running of the procedure. The source nodes of D are a subset of the antecedents of C, and since C is M-meaningful, the set of source nodes of D is in M. Thus T is in M after the initialization in step (1).

Inductively assume that T is in M and suppose another element, x, is to be added to T in step (5). Because the nodes are processed in topological order with respect to D, the set S_x of nodes with edges into x is a subset of T at this point. Moreover, $\phi_* \vdash (S_x \rightarrow x)$. Since T is in M by the induction hypothesis, and M is closed under implication with respect to ϕ_*, $(T \cup \{x\})$ is also in M. Thus, T remains in M after the execution of step (5).

Since every request for a hint is made with a clause whose antecedents are a subset of T, every request for a hint is made with an M-meaningful clause, so the answers are correct.

Each non-source node x of D is considered at most once. If x is considered, the clause $C' = (S_x \rightarrow x)$ is formed, where S_x is the set of nodes of D with edges into x. This is a clause of ϕ because D is a derivation-DAG with respect to ϕ. C' is tested to see if it is implied by ϕ_*. If it is not, then it is a clause of ϕ that is not implied by ϕ_*, and it is correctly returned in step (4).

Suppose this procedure tests all the non-source nodes of D and finds that every clause tested is implied by ϕ_*. Then the clause C is implied by ϕ_*, contrary to the input conditions. Thus, *Find-Incorrect* must eventually return a clause C' that is in ϕ but not implied by ϕ_*.

Since each non-source node of D is considered at most once, there are at most $|V| - 1$ clauses tested before this procedure returns a clause C'. Hence the total running time of this procedure is bounded by a polynomial in the size of ϕ and $|V|$. Q.E.D.

Note that *Find-Incorrect* uses the request for hint queries in a restricted way; in the terminology of [5], restricted superset queries suffice in this procedure.

We now modify the procedure *HL* so that it takes as input a positive Horn sentence ϕ and uses M-equivalence queries and calls to *Find-Missing* and *Find-Incorrect* to "correct" ϕ until it is M-equivalent to ϕ_*. The modified procedure is called *IHL*, for Incremental Horn Learner.

The procedure $IHL(\phi)$

1. Make an M-equivalence query with ϕ. If the reply is *yes*, output ϕ and halt.

2. Otherwise, the reply is an M-meaningful clause C that is a counterexample.

3. If C is a positive counterexample then let C' be the clause returned by *Find-Missing*(C), set $\phi = \phi \wedge C'$, and go to step 1.

4. If C is a negative counterexample then let C' be the clause returned by *Find-Incorrect*(C), remove C' from ϕ, and go to step 1.

We quantify the performance of this algorithm using a measure of how "close" the initial theory ϕ is to the correct theory ϕ_*.

Define a clause C to be ***incorrect*** if it is M-meaningful and not implied by ϕ_*. Clearly no such clause may appear in ϕ if it is to be M-equivalent to ϕ_*. Let $i(\phi)$ be the number of incorrect clauses in ϕ. Define the ***correct part*** of ϕ, denoted $c(\phi)$ to be the sentence which is the conjunction of all the clauses C in ϕ that are not incorrect. Note that $\phi \vdash c(\phi)$.

Define a clause C to be ***missing with respect to*** ϕ if it is M-meaningful and implied by ϕ_* but not by $c(\phi)$. Let $m(\phi)$ be the number of clauses in ϕ_* that are missing with respect to ϕ. The reason for defining this with respect to just the correct part of ϕ is that there may be some very "powerful" incorrect clauses in ϕ, which would mask just how much is missing from ϕ.

The measure of how "close" ϕ is to the correct theory ϕ_* is

$$d(\phi) = i(\phi) + m(\phi).$$

The smaller this number is, the closer ϕ is to ϕ_*. In particular, if this number is zero, the two theories are M-equivalent.

Lemma 8 *If ϕ and ϕ_* are positive Horn sentences then $d(\phi) = 0$ if and only if ϕ is M-equivalent to ϕ_*.*

If ϕ is M-equivalent to ϕ_*, then for every M-meaningful clause C, $\phi \vdash C$ if and only if $\phi_* \vdash C$. For every M-meaningful clause C in ϕ, $\phi_* \vdash C$, so there are no incorrect clauses in ϕ and $i(\phi) = 0$. Moreover, $c(\phi) = \phi$, and for every M-meaningful clause C in ϕ_*, $\phi \vdash C$, so $c(\phi) \vdash C$ and $m(\phi) = 0$. Thus, $d(\phi) = 0$.

Conversely, suppose ϕ is not M-equivalent to ϕ_*. Then there is an M-meaningful clause C such that $\phi_* \vdash C$ and $\phi \nvdash C$, or vice versa.

If $\phi_* \vdash C$ and $\phi \nvdash C$, then by the correctness of the *Find-Missing* procedure, if it is called with C as input, the output is an M-meaningful clause C' in ϕ_* such that $\phi \nvdash C'$. Since $\phi \vdash c(\phi)$, $c(\phi) \nvdash C'$, so $m(\phi) > 0$.

If $\phi_* \nvdash C$ and $\phi \vdash C$, then by the correctness of the *Find-Incorrect* procedure, if it is called with C as input, the output is an M-meaningful clause C' in ϕ such that $\phi_* \nvdash C'$, so $i(\phi) > 0$.

Thus in either case, $d(\phi) > 0$. Hence if $d(\phi) = 0$ then ϕ is M-equivalent to ϕ_*. Q.E.D.

Now we come to the analysis of *IHL*

Theorem 9 *Suppose the correct sentence ϕ_* is a positive acyclic Horn sentence. Let the input to IHL be the positive Horn sentence ϕ. The output will be a positive Horn sentence M-equivalent to ϕ_*. IHL runs in time bounded by a polynomial in the sizes of ϕ and ϕ_* and the number of variables, $|V|$. IHL makes at most $d(\phi)+1$ M-equivalence queries. Every request for a hint is with an M-meaningful clause.*

If *IHL* ever halts, its output is a positive Horn sentence because ϕ is initially a positive Horn sentence, and *Find-Missing* returns only clauses from ϕ_* to be added to ϕ. It is clear that if *IHL* ever halts, its output is M-equivalent to ϕ_*. Since *IHL* calls *Find-Missing* and *Find-Incorrect* with M-meaningful clauses that satisfy their respective input conditions, every request for a hint is with an M-meaningful query.

To bound the running time of *IHL*, note that for each M-equivalence query that is answered *no*, there will be a call to *Find-Incorrect* or to *Find-Missing*.

A call to *Find-Incorrect* returns an M-meaningful clause C' that is in ϕ but not implied by ϕ_*, and C' is then removed from ϕ. That is, an incorrect clause if removed from ϕ. This operation decreases $i(\phi)$ by one.

A call to *Find-Missing* returns an M-meaningful clause C' of ϕ_* that is not implied by ϕ, and this clause is added to ϕ. This operation decreases the number of M-meaningful clauses in ϕ_* that are not implied by $c(\phi)$ by at least one. That is, it decreases $m(\phi)$ by at least one.

Since no incorrect clause is ever added to ϕ by *IHL*, the number $i(\phi)$ can never increase. The only clauses ever deleted from ϕ are incorrect, so the set of clauses in the correct part of ϕ is monotonically non-decreasing with respect to set inclusion. Thus, once an M-meaningful clause of ϕ_* is implied by $c(\phi)$, it will continue to be implied by $c(\phi)$ in subsequent iterations. Hence $m(\phi)$ can never increase.

Thus, the quantity $d(\phi)$ can never increase, and is decreased by at least one for each M-equivalence query that is answered *no*. Hence, after at most $d(\phi)$ calls to *Find-Missing* and *Find-Incorrect*, the procedure *IHL* must arrive at formula ϕ that is M-equivalent to ϕ_*. Thus there are a total of at most $d(\phi)+1$ M-equivalence queries made before *IHL* terminates with a correct answer.

Each call to *Find-Missing* runs in time polynomial in $|V|$, and each call to *Find-Incorrect* runs in time polynomial in the size of ϕ and $|V|$. Since $m(\phi)$ is bounded by the size of ϕ_* and $i(\phi)$ is bounded by the size of ϕ, it is clear that the total running time of *IHL* is bounded by a polynomial in the sizes of ϕ_*, ϕ, and $|V|$. Q.E.D.

Note that in the case that the input sentence ϕ is the empty sentence, $\top$, *Find-Incorrect* is never called, and this procedure reduces to *HL*.

5.2 An example of *IHL*

A brief example will illustrate *IHL* and *Find-Missing*. Suppose the target theory is ϕ_w and the current value of ϕ is the conjunction of the following clauses.

1. three_wheels $\wedge$ one_seat $\wedge$ pedals $\rightarrow$ trike.

2. two_wheels $\wedge$ one_seat $\wedge$ pedals $\rightarrow$ bike.

3. trike $\rightarrow$ toy.

4. bike $\rightarrow$ toy.

5. toy $\rightarrow$ object.

Note that clause (4) is incorrect according to the theory ϕ_w. An M-equivalence query with ϕ could return the following clause as a negative counterexample, that is, implied by ϕ but not ϕ_w.

$$\text{two\_wheels} \wedge \text{one\_seat} \wedge \text{pedals} \rightarrow \text{toy}.$$

Find-Incorrect is called with this clause, and uses a variant of *Forward-Chain* to find a derivation-DAG of this clause from ϕ. The DAG has nodes

$$\text{two\_wheels, one\_seat, pedals, bike, toy}$$

and edges

$$\text{(two\_wheels, bike), (one\_seat, bike), (pedals, bike), (bike, toy).}$$

The first query is a request for a hint with the clause

$$\text{two\_wheels} \wedge \text{one\_seat} \wedge \text{pedals} \rightarrow \text{bike}.$$

The reply is *one step*, so ϕ_w implies this clause. The next query is a request for a hint with the clause

$$\text{bike} \rightarrow \text{toy}.$$

The reply is *no*, signifying that this clause is not implied by ϕ_w, so this clause is returned by *Find-Incorrect* and removed from ϕ.

5.3 Non-positive Horn sentences

The algorithm *IHL* can be modified fairly naturally to learn acyclic Horn sentences that are not positive. But first, what is the meaning of a negative Horn clause? A negative clause such as

$$C = (\text{two\_wheels} \wedge \text{three\_wheels} \rightarrow \bot)$$

stipulates that the variables "two_wheels" and "three_wheels" are never simultaneously true.

Recall that the meaning structure M is intended to designate those sets of variables that can be simultaneously true. Hence in some sense, M is performing the same function as negative clauses. However, the difference is that the meaning structure is implicit, while the inclusion of negative clauses is explicit. That is, if the set containing the variables "two_wheels" and "three_wheels" is omitted from M, then we don't care how the final theory ϕ treats situations in which both of these variables are true. However, if we stipulate that the final theory ϕ should contain or imply the clause C, then the theory must yield $\bot$ when both variables are true. The cost associated with this explicit treatment is an extra clause included in the correct theory.

In the hypothetical situation of Molly the alien, we might know that her procedures for counting wheels were a little shaky, so that she might mistakenly believe that both "two_wheels" and "three_wheels" were true in some situation. In this case, we might want to explicitly include the clause C, with the understanding that Molly would detect an error and re-check her procedures if any actual situation yielded $\perp$. Thus, in practice, one might want to be able to use both the meaning structure (for "don't care" situations), and explicit non-positive clauses (for error detection).

To do this, the meaning of the meaning structure must be changed a bit, to those sets of variables such that we do "care" about the outcome of the theory if all the variables in the set are true. Thus, in the above example, the meaning structure should include the set consisting of the two variables "two_wheels" and "three_wheels".

It turns out to be sufficient to expand the notion of a derivation-DAG in a controlled way. Let ϕ be a not necessarily positive Horn sentence. If ϕ implies the non-positive clause $(A \rightarrow \perp)$, then for every variable z, ϕ implies the clause $(A \rightarrow z)$. In some sense, these latter clauses are "silly" consequences of the theory, analogous to the proposition:

If one equals two then the moon is made of green cheese.

Hence the following definition. Let $C = (A \rightarrow z)$ be any positive Horn clause. We say that C is *silly* for ϕ if and only if ϕ implies $(A \rightarrow \perp)$.

For a positive clause C such that $\phi \vdash C$, a derivation-DAG of C from ϕ will be just what it was before. Note that if C is silly for ϕ, a derivation-DAG of C from ϕ may not exist.

If $C = (A \rightarrow \perp)$ is a negative clause such that $\phi \vdash C$, then a derivation-DAG of C from ϕ will be a directed acyclic graph with the following properties. Its nodes are a subset of the variables V together with a special node, $\perp$, which is the unique sink node. There is a directed path from every node of the graph to the sink node. The source nodes are a subset of A. For each non-source node x in the graph, if S_x is the set of nodes with edges into x, then $(S_x \rightarrow x)$ is a clause of ϕ.

The key property of this definition is the following.

Lemma 10 *Let ϕ be a Horn sentence. For any clause C, if there is a derivation-DAG of C from ϕ, then ϕ implies C. For any clause C that is negative or positive and not silly for ϕ, if ϕ implies C then there is a derivation-DAG of C from ϕ.*

Thus the proof procedure is still sound, although it may not be complete for silly clauses. This is proved by modifying the *Forward-Chain* procedure to check negative as well as positive clauses. As soon as the procedure detects that all the antecedents of some negative clause of ϕ are in T, it returns the special value $\perp$. We omit the details.

A modification is required to the *IHL* procedure, to check whether a positive counterexample $C = (A \rightarrow z)$ is silly for ϕ_*. The check is done by using a request for a hint to determine whether ϕ_* implies the clause $C_n = (A \rightarrow \perp)$. If so, the clause C is silly for ϕ_*, and the clause C_n is substituted for it in step (3) of the procedure. This guarantees that there will be a derivation-DAG to use in the request for a hint in *Find-Missing*. The rest of the procedure and sub-procedures are unchanged.

5.4 Cyclic Horn sentences

In contrast to non-positive sentences, a significant modification is necessary to deal with cyclic Horn sentences. An example of a cyclic Horn sentence is

$$\phi_1 = (a \rightarrow b) \wedge (b \rightarrow c) \wedge (c \rightarrow a).$$

The import of this construct is that if any of a, b, or c is true then they all are. This meaning cannot be captured by any acyclic sentence. Thus it is useful to be able to deal with cyclic Horn sentences.

The difficulty with *IHL* for cyclic Horn sentences is that nothing prevents the request for hint queries from cycling. As a concrete example, consider the Horn sentence

$$\phi_* = (x \rightarrow y) \wedge (y \rightarrow a) \wedge (y \rightarrow b) \wedge (a \rightarrow b) \wedge (b \rightarrow a).$$

For the hypothesis $\top$ an M-equivalence query could return the counterexample $(x \rightarrow a)$, implied by ϕ_* but not by $\top$.

Since $(x \rightarrow a)$ is not derived in one step from ϕ_*, a request for a hint produces a hint, say b. This is a legal hint, since there is a derivation

$$x \rightarrow y \rightarrow b \rightarrow a.$$

Now a request for a hint with $(x \rightarrow b)$ can legally be answered a, since it is not derived in one step from ϕ_* and there is a derivation

$$x \rightarrow y \rightarrow a \rightarrow b.$$

And we are right back where we started, asking for a hint for $(x \rightarrow a)$. Thus an unsympathetic teacher could keep *IHL* cycling forever in this situation.

One possible approach is to try to equip *IHL* to detect and correct for this situation; I don't know how to do that in general. Another approach, taken here, is to restrict the malice of the teacher by placing additional constraints on how a request for a hint may be answered.

The simplest restriction is to require that the hint come from a derivation-DAG with the smallest possible depth for C from ϕ. If $C = (A \rightarrow z)$ is a clause implied by ϕ_*, let $d(A, z)$ denote the minimum depth of any derivation-DAG of C from ϕ. Note that this is bounded by $|V|$.

To see that this restriction prevents cycling, consider the sequence of clauses tested with request for hint queries in one top-level call to *Find-Missing*. If the hint x is returned for the clause $(A \rightarrow z)$, then the next clause tested is either $(A \cup \{x\} \rightarrow z)$ or $(A \rightarrow x)$. Now $d(A \cup \{x\}, z) \leq d(A, z)$ and $d(A, x) < d(A, z)$. Thus the antecedents form a non-decreasing sequence (with respect to set containment) and the $d(A, z)$ measure is non-increasing. Moreover, either the set of antecedents increases or the $d(A, z)$ measure decreases with each hint. Hence there are fewer than $2|V|$ hints returned before the top-level call of *Find-Missing* returns.

How onerous to the teacher is this restriction on hints? *Forward-Chain*, since it searches breadth-first, will produce a derivation-DAG with the smallest possible depth, but other,

more practical, proof procedures are likely not to be guaranteed to produce a shallowest derivation-DAG. Thus, this restriction is probably too severe.

It would be sufficient, and in some sense more reasonable, if the teacher simply stuck to one derivation-DAG for the duration of one top-level call to *Find-Missing*, instead of hopping between derivations as in the above example. We could ensure this by adding another parameter to the request for a hint query which in effect specifies the derivation-DAG to be used. A top-level call to *Find-Missing* then makes an initial request for a hint query with no derivation-DAG specified, and if the answer is a hint x, then an identifier is also returned to specify the derivation-DAG it came from. Further requests for hints made before this top-level call terminated would then specify this identifier, constraining further hints to come from the same derivation-DAG.

This would prevent cycling, since the successive hints must come from a monotonically shrinking portion of the single derivation-DAG being used to produce them. This approach guarantees that the number of hints needed is bounded by the number of nodes in the initial derivation chosen by the teacher, which however need not be minimal. This modification is not too unreasonable in the presence of a sympathetic teacher, since it amounts to being able to name and refer to an explanation.

5.5 A closer look at complexity

The algorithms and analyses have been given above in their simplest form, for purposes of exposition. Now it is time to look a little more closely at running times.

The procedure *Forward-Chain* is called to determine whether the current theory implies a clause C, and to produce a derivation-DAG for C if so. As described, it uses a simple breadth-first search, is not incremental, and is pretty unworkable for large theories. However, the problem of maintaining and drawing inferences from a theory in the presence of incremental changes is beyond the scope of this paper. It suffices to use any method that will produce a correct derivation-DAG if one exists. We examine the complexity of the procedures that are specific to learning: *Reduce*, *Find-Missing*, and *Find-Incorrect*.

Reduce. The input to *Reduce* is a clause $C = (A \rightarrow z)$ that is derivable from ϕ_* in one step. Without additional information, ϕ_* may consist of the single clause $(A' \rightarrow z)$ for any subset A' of A. Each request for a hint returns only one bit of information ("not implied", or "one step"), so at least $|A|$ queries will be required. Since the search is a greedy search for a minimal set A' such that $(A' \rightarrow z)$ is derivable in one step from ϕ_*, $|A|$ queries are also sufficient. Any substantial improvement to *Reduce* would probably have to make use of additional information available from the structure or semantics of the domain under consideration.

Find-Missing. *Find-Missing* is called with a clause $C = (A \rightarrow z)$ that is implied by ϕ_* and not by ϕ. It makes calls to *Forward-Chain* to test whether ϕ implies $(A \rightarrow x)$ for each hint x, but any proof procedure would suffice instead.

If requests for hints are answered in an unfavorable way, there may be many more requests for hints than nodes in a derivation-DAG for C from ϕ_*. The reason for this is the

"derivation hopping" problem mentioned in the section on cyclic sentences. To show how this can complicate even the acyclic case, consider the following example.

Let $n \geq 1$ and let ϕ_* be the conjunction of the clause $(x \rightarrow y)$ and the clauses $(a \rightarrow b_i)$ and $(y \wedge b_i \rightarrow z)$, for $i = 1, \ldots, n$. There are n different derivation-DAGs with 5 nodes and depth 2 for the clause $(a \wedge x \rightarrow z)$. Suppose the current theory ϕ consists of the conjunction of the clauses $(a \rightarrow b_i)$ for $i = 1, \ldots, n$. Then an M-equivalence query might return the positive counterexample $(a \wedge x \rightarrow z)$. A request for a hint might return b_1. Since this is derivable from a using ϕ, the next request for a hint would be with $(a \wedge x \wedge b_1 \rightarrow z)$. A legal answer to this is b_2. Then a request for a hint with $(a \wedge x \wedge b_1 \wedge b_2 \rightarrow z)$ could be answered with b_3, and so on, producing n hints, even though each possible derivation has only 5 nodes.

Note that this problem affects even the version of the algorithm in which the hints are restricted to come from a derivation of minimum depth. However, in the version in which requests for hints can specify the derivation-DAG to be used, the maximum number of requests for hints in one call to *Find-Missing* is bounded by the number of nodes in the derivation-DAG chosen by the teacher, in this case, 5.

Find-Incorrect. *Find-Incorrect* is called with a clause C such that ϕ implies C but ϕ_* does not. It constructs a derivation-DAG of C from ϕ and then systematically makes request for hint queries for the clauses of ϕ used in the derivation-DAG until an incorrect one is found. They are queried in topological order with respect to the derivation-DAG, since this guarantees the M-meaningfulness of the clauses queried.

The analogous procedure of Shapiro [35] admits an improvement which consists of finding a query that substantially reduces the number of clauses to be tested, instead of testing them one at a time. Can we make use of this improvement?

Shapiro uses derivation trees instead of derivation-DAGs. His improvement is to check a large subtree of the derivation tree for correctness, and thus to restrict further attention to the subtree or the tree minus the subtree.

However, a derivation tree may be exponentially larger than the corresponding DAG. As an example, let $n \geq 1$ and consider the sentence that is the conjunction of the clause $(a_n \wedge b_n \rightarrow z)$, and the clauses $(a_i \wedge b_i \rightarrow a_{i+1})$ and $(a_i \wedge b_i \rightarrow b_{i+1})$ for $i = 1, \ldots, n-1$. Then for each $n \geq 1$, a derivation-DAG for the clause $(a_1 \wedge b_1 \rightarrow z)$ from this sentence has $2n+1$ nodes, while the corresponding derivation tree has $2^{n+1} - 1$ nodes.

We prefer DAGs to tree for their conciseness, but Shapiro's improvement is then not directly applicable. We must generalize the notion of a subtree of a derivation tree into a self-contained "piece" of a derivation-DAG. This is done using the notion of a closed subset of a DAG, see Appendix C for more details.

6 A graph problem: debugging a DAG

For a different view of the learning algorithm presented above, we consider the special case of positive Horn sentences with exactly two literals per clause, and recast the learning problem as one of "debugging" a directed graph. If G is a directed graph, the assertion that there is a directed path of length at least zero from x to y in G is denoted $x \rightsquigarrow y$ *in* G. The assertion that there is no path from x to y in G is denoted $x \not\rightsquigarrow y$ *in* G.

6.1 Positive Horn 2-CNF sentences as directed graphs

Let ϕ be a positive Horn sentence with exactly two literals per clause. Every clause of ϕ is of the form $(x \rightarrow y)$. We define a related directed graph, $G(\phi)$, as follows. The nodes are the variables appearing in ϕ, and there is an edge (x, y) from x to y in ϕ if and only if there is a clause $(x \rightarrow y)$ in ϕ. The following lemma is immediate from the correctness of *Forward-Chain* in this case.

Lemma 11 *The sentence ϕ implies the clause $(x \rightarrow z)$ if and only if there is a directed path from x to z in $G(\phi)$. That is, $\phi \vdash (x \rightarrow z)$ if and only if $x \rightsquigarrow y$ in $G(\phi)$.*

Let G_1 and G_2 be two directed graphs on the same set of nodes V. Then G_1 is *transitively equivalent* to G_2 if and only if for every pair x and y of elements of V, there is a directed path from x to y in G_1 if and only if there is a directed path from x to y in G_2. That is, for all $x, y \in V$, $x \rightsquigarrow y$ in G_1 if and only if $x \rightsquigarrow y$ in G_2.

Lemma 12 *The sentences ϕ_1 and ϕ_2 are logically equivalent if and only if the graphs $G(\phi_1)$ and $G(\phi_2)$ are transitively equivalent.*

Suppose ϕ_1 and ϕ_2 are logically equivalent. Suppose $x \rightsquigarrow y$ in $G(\phi_1)$. Then $\phi_1 \vdash (x \rightarrow y)$, so $\phi_2 \vdash (x \rightarrow y)$, and $x \rightsquigarrow y$ in $G(\phi_2)$. Thus a path from x to y in $G(\phi_1)$ implies a path from x to y in $G(\phi_2)$. By symmetry the converse holds, so if ϕ_1 and ϕ_2 are logically equivalent, $G(\phi_1)$ and $G(\phi_2)$ are transitively equivalent.

Conversely, suppose $G(\phi_1)$ and $G(\phi_2)$ are transitively equivalent. Consider any clause $(x \rightarrow y)$ in ϕ_1. Clearly $\phi_1 \vdash (x \rightarrow y)$, so $x \rightsquigarrow y$ in $G(\phi_1)$. Thus $x \rightsquigarrow y$ in $G(\phi_2)$, and $\phi_2 \vdash (x \rightarrow y)$. Thus ϕ_2 implies every clause of ϕ_1, and by symmetry, ϕ_1 implies every clause of ϕ_2. Hence ϕ_1 and ϕ_2 are logically equivalent. Q.E.D.

6.2 A "debugging" problem on directed graphs

Every directed graph is isomorphic to $G(\phi)$ for some positive Horn sentence with exactly two literals per clause. If we omit the question of "meaningfulness", the two lemmas in the previous section allow us to interpret the learning problem in this restricted case as a problem on directed graphs.

Let V be a known set of nodes. There is an unknown directed graph G_* on the nodes V. The problem is to find a directed graph G on the nodes V that is transitively equivalent to G_* using equivalence queries and requests for hints.

What is an equivalence query in this setting? The learner proposes a directed graph G. If G is transitively equivalent to G_*, the reply is *yes*; otherwise, the reply is *no* and a counterexample.

In the general case, a counterexample is any Horn clause C implied by the correct sentence but not by the proposed sentence, or vice versa. In terms of graphs, such a counterexample would be a subset S of V and a node $y \in V$ such that there is a path from some node of S to y in G_* and no path from any node of S to y in G, or vice versa. However, for the graph problem it seems more natural to restrict the counterexample to be a pair of nodes x and y such that there is a path from x to y in G_* but not in G, or vice versa. The implications of this restriction are considered briefly in Appendix A.

Thus, for this problem, equivalence queries and requests for hints are defined as follows.

1. An equivalence query: propose a directed graph G. If G is transitively equivalent to G_* the answer is *yes*. Otherwise, the answer is *no*, and a counterexample is provided, that is, an ordered pair of nodes (x, y) such that there is a path from x to y in G but not in G_* or vice versa.

2. A request for a hint: propose a pair of nodes (x, y). If there is no path from x to y in G_*, the answer is *no*. If (x, y) is an edge of G_*, the answer is *edge*. If there is a path but no edge from x to y in G, the answer is a node w that is not equal to x or y but appears in some path from x to y in G_*.

One method is to make a request for a hint with every pair (x, y) of nodes, record which ones are edges in G_*, and finally output G exactly equal to G_*. This approach necessarily uses $|V|(|V|-1)$ queries in the worst case, even if G_* is an extremely sparse graph. It also is not "incremental" in the sense that if it is given a graph that is "close" to G_*, it does not use that information to reduce the work of finding a graph exactly equivalent to G_*.

However, if we restrict G and G_* to be acyclic, then there is an efficient solution to this debugging problem. In particular, there is an algorithm that runs in time polynomial in $|V|$ and uses equivalence queries and request for hint queries to find incorrect and missing edges in an initial graph G in $O(\log |V|)$ queries per edge found to be missing or incorrect. The algorithm is based on the ideas of *IHL*, with some combinatorial optimizations. Further description appears in [1].

7 Appendix A: Clauses versus assignments as examples

For clarity we ignore the issue of "meaningfulness" in this discussion. Equivalently, we could take M to be all subsets of V.

In this paper a Horn sentence ϕ is taken to denote the set of Horn clauses logically implied by ϕ. An example consists of a clause C and an indication of whether it is implied by the target sentence ϕ_* or not. Such a set of examples is sufficient in the sense of Laird [25], since the set of Horn clauses implied by a Horn sentence characterize it up to logical equivalence.

An alternative approach is to take a Horn sentence ϕ to denote the set of all those assignments of truth values to the variables V that satisfy ϕ. (This is the approach taken in [5].) In this case an example consists of an assignment of truth values to the variables and an indication of whether this assignment makes the target sentence ϕ_* true or false. Assignments seem less natural as examples than clauses in this domain. We examine the interconvertibility of these two schemes below.

A third approach is to specialize the setting used by Shapiro [35] to the case of propositional Horn sentences. Then examples are literals and two theories are considered equivalent if they imply the same set of literals. This notion of equivalence is weaker than the present one; in particular, the empty sentence $\top$ is equivalent to the example sentence ϕ_w in this sense. This approach will not be considered further.

How inter-convertible are clauses and assignments as examples? We consider both the non-Horn and Horn cases.

7.1 The general (non-Horn) case.

The sentences ϕ_* and ϕ are permitted to be general conjunctive normal form propositional sentences, and clauses are permitted to be arbitrary disjunctions of literals.

Suppose t is an assignment of truth values to the variables that is a positive counterexample, that is, satisfies ϕ_* but not ϕ. Then by evaluating each clause of ϕ with t, we can find a clause C in ϕ that is falsified by t. Thus $\phi \vdash C$ and $\phi_* \nvdash C$, so C is a negative example in the clause scheme.

Suppose t is an assignment that is a negative counterexample, that is, t falsifies ϕ_* and satisfies ϕ. We construct a (general) clause C by including the literal a if $t(a)$ is false, and the literal $\neg a$ if $t(a)$ is true. Then t is the unique assignment that falsifies C, and t also falsifies ϕ_*, so $\phi_* \vdash C$. Moreover, since t satisfies ϕ and falsifies C, $\phi \nvdash C$. Thus C is a positive counterexample in the clause scheme.

Hence in the non-Horn case, assignment counterexamples can be efficiently converted into clause counterexamples. The reverse conversions seem to cause trouble, as we now see.

Suppose C is a (general) clause that is a positive counterexample, that is, $\phi_* \vdash C$ and $\phi \nvdash C$. Then there is an assignment t that satisfies ϕ and not C. We can find this assignment, though it entails the computational problem of testing general conjunctive normal form sentences for satisfiability, which is an NP-complete problem. (In particular, if the sentence ϕ_* is contradictory and the clause C is the empty clause, $\perp$, then finding an assignment counterexample is precisely the problem of finding a satisfying instance of ϕ.) Then t is a negative counterexample in the assignment scheme, since t falsifies ϕ_* and satisfies ϕ.

Suppose C is a clause that is a negative counterexample, that is, $\phi_* \nvdash C$ and $\phi \vdash C$. Suppose we also have an oracle for logical implication, that is, each query "*Does* ϕ_* *imply* C*?*" will be correctly answered *yes* or *no* for any (general) clause C. If the variable a does not occur in C then at least one of the two clauses $(C \vee a)$ and $(C \vee \neg a)$ must fail to be implied by ϕ_*, so by querying the oracle we can add either a or $\neg a$ to C and preserve the conditions that $\phi_* \nvdash C$ and $\phi \vdash C$. Continuing in this way, we can find a clause C that contains every variable or its negation and is such that $\phi_* \nvdash C$ and $\phi \vdash C$. There is a unique truth value assignment t that makes all the literals in C false. Then t must satisfy ϕ_* but not ϕ, so it is a positive counter-example in the assignment scheme.

Thus, converting (general) clause counterexamples into assignment counterexamples is an NP-complete problem in one case, and seems to require an oracle for logical implication in the other.

7.2 The Horn case.

Now we consider what happens if the sentences and clauses are restricted to be Horn sentences and clauses.

The method described above for converting an assignment t that is a positive counterexample into a clause that is a negative counterexample by searching for a clause of ϕ falsified by t works in this case also.

Suppose C is a Horn clause that is a positive counterexample, that is, the target sentence ϕ_* implies C, but the current hypothesis ϕ does not. Then, using the (polynomial-time) satisfiability procedure for Horn sentences, we can find an assignment t of truth values to

the variables such that t satisfies ϕ and falsifies C. Then t must falsify ϕ_*, so it is a negative counterexample in the assignment scheme, that is, it falsifies ϕ_* and satisfies ϕ.

Thus in the Horn case, a positive counterexample in either scheme can be efficiently converted into a negative counterexample in the other scheme.

Now suppose we have an assignment t that is a negative counterexample, that is, falsifies ϕ_* and satisfies ϕ. In the general case it was easy to convert t into a (general) clause implied by ϕ_* but not by ϕ. However, that clause is not necessarily in Horn form. To solve this problem, we assume the existence of an oracle for logical implication for Horn sentences, that is, each query "*Does* ϕ_* *imply* C*?*" will be correctly answered provided C is a Horn clause.

Let A denote the set of variables assigned true by t. There must be a clause C of ϕ_* that is falsified by t, that is, its antecedents are a subset of A and it either has no consequent or a consequent $z \in V - A$. We use the oracle to test each of the clauses $(A \rightarrow \bot)$, and $(A \rightarrow z)$ for each $z \in V - A$. Note that t falsifies each of these clauses. At least one of these clauses is subsumed by C, so there must be at least one of these clauses C' such that $\phi_* \vdash C'$. Since t falsifies C' and satisfies ϕ, we know that $\phi \nvdash C'$. Hence we have found a clause C' that is a positive counterexample in the clause scheme.

In the last case, assume that we have a clause C that is a negative counterexample, that is, $\phi_* \nvdash C$ and $\phi \vdash C$. The approach in the general case constructs an assignment that is a positive counterexample by using queries to an oracle for (general) logical implication. It does not seem that an oracle for (Horn) logical implication is sufficient in this case.

Summarizing, in the Horn case, assignment counterexamples can be converted into clause counterexamples efficiently provided there is an oracle to test logical implication of Horn clauses by ϕ_*. In the case when M is all subsets of V, a request for a hint can be used to test whether ϕ_* implies an arbitrary Horn clause. Thus, the algorithms *HL* and *IHL* could be modified to deal with assignments rather than clauses as counterexamples, though the number of requests for hints would be increased somewhat.

7.3 Restricted sets of clauses as counterexamples

Consider the problem of learning a positive acyclic Horn sentence ϕ_* with exactly two literals per clause. We could take ϕ_* to denote the set of all Horn clauses that it implies, or we could take it to denote the set of all Horn clauses with exactly two literals that it implies. The choice results in two different definitions of an equivalence query. In both, the learner proposes a positive acyclic Horn sentence ϕ with exactly two literals per clause, and if ϕ is logically equivalent to ϕ_* the reply is *yes*, and otherwise, the answer is *no* and a counterexample. However, the definition of a counterexample differs in the two cases, as follows.

1. The counterexample may be any Horn clause C that is implied by ϕ_* and not by ϕ, or vice versa.

2. The counterexample may be any Horn clause C with exactly two literals that is implied by ϕ_* but not by ϕ, or vice versa.

For the graph debugging problem stated above and considered in [1] we have chosen the second alternative. However, the choice is not indifferent, as we now indicate.

There is a simple polynomial time learning algorithm for this problem that uses only equivalence queries of type 2. The method is to begin with $\phi = \top$, and query ϕ. The counterexample will be a clause C with exactly two literals such that ϕ_* implies C, but ϕ does not. Modify ϕ by conjoining C to it, and iterate. This must converge to a positive acyclic Horn sentence ϕ with exactly two literals per clause after fewer than $|V|^2$ iterations, since this is a bound on the number of distinct clauses that can be returned as counterexamples.

However, if $P \neq NP$ then there is no polynomial time learning algorithm for this problem that uses only equivalence queries of type 1. The reason for this is the NP-completeness of the following problem. Given two sets S and T of Horn clauses, determine whether there is a positive acyclic Horn sentence ϕ with exactly two literals per clause such that for every clause $C \in S$, $\phi \vdash C$, and for every clause $C \in T$, $\phi \nvdash C$. Call this problem ***constrained 2-Horn consistency***. If there were a polynomial time exact learning algorithm in this setting that used only equivalence queries of type 1, we could use it to solve the constrained 2-Horn consistency problem in polynomial time, as follows.

Given S and T, run the learning algorithm until it makes an equivalence query with a sentence ϕ. Check to see whether $\phi \vdash C$ for all $C \in S$ and $\phi \nvdash C$ for all $C \in T$. (This can be done in polynomial time because ϕ is a Horn sentence and each C is a Horn clause.) If ϕ is consistent with all these requirements, halt and output ϕ. Otherwise, if there is a clause $C \in S$ such that $\phi \nvdash C$, reply *no* to the equivalence query, and return C as the counterexample. Similarly for the case of a clause $C \in T$ such that $\phi \vdash C$. If the learning algorithm exceeds its running time bound (some polynomial in $|V|$) without finding a ϕ agreeing with all the requirements, halt and output "no".

The proof that the constrained 2-Horn consistency problem is NP-complete is via a reduction from exact cover by 3-sets, and is omitted. (See [14] for a definition of X3C.)

8 Appendix B: A lower bound for learning Horn sentences

In this section we show that learning Horn sentences using equivalence queries with clauses as counterexamples appears to be impossible with a polynomial time algorithm. This is an application of the approach of Pitt and Warmuth to finding reductions between prediction problems [29].

8.1 Representing circuits by Horn sentences

Consider any acyclic boolean circuit κ with n inputs $x_1, \ldots, x_n$ and one output, using AND, OR, and NOT gates. The *size* of κ is the number of gates in κ; we assume that the size is at least 1. We may construct an associated acyclic Horn sentence ϕ_κ that "represents" κ as follows.

Let the inputs and gates of κ be numbered 1 through M, and assume that 1 through n represent x_1 through x_n and $n+1$ represents the gate whose output is the output of κ. For each $i = 1, \ldots, M$, there are two variables: Y_i and Z_i, representing the value of the input or gate numbered i. Y_i is true if the value is 1, and Z_i is true if the value is 0. For each gate in κ there are two or three clauses in ϕ_κ, as follows.

1. For each NOT gate in κ, if the input is numbered j and the output is numbered k

then there are two clauses:

$$(Y_j \rightarrow Z_k),$$
$$(Z_j \rightarrow Y_k).$$

2. For each AND gate in κ, if the inputs are numbered j and k, and the output is numbered l, then there are three clauses:

$$(Y_j \wedge Y_k \rightarrow Y_l),$$
$$(Z_j \rightarrow Z_l),$$
$$(Z_k \rightarrow Z_l).$$

3. For each OR gate in κ, if the inputs are numbered j and k and the output is numbered l, then there are three clauses:

$$(Z_j \wedge Z_k \rightarrow Z_l),$$
$$(Y_j \rightarrow Y_l),$$
$$(Y_k \rightarrow Y_l).$$

This completely specifies ϕ_κ; note that it is acyclic. Now we indicate how ϕ_κ represents κ. Let a be any assignment to the variables x_i for $i = 1, \ldots, n$. Define $\kappa(a) = 1$ if and only if the output of κ is 1 when the values specified by a are applied as inputs. With a we associate A_a, a set of n variables defined as follows.

$$A_a = \{Y_i : a(x_i) = 1\} \cup \{Z_i : a(x_i) = 0\}.$$

Then ϕ_κ represents κ in the following sense.

Lemma 13 *For every assignment a to the variables x_i, $\kappa(a) = 1$ if and only if ϕ_κ implies the clause $(A_a \rightarrow Y_{n+1})$.*

We omit the simple inductive proof of this lemma. Note that the number of clauses in ϕ_κ is bounded by three times the number of gates in κ, and each clause has at most three occurrences of literals. Thus $size(\phi_\kappa) \leq 9 \times size(\kappa)$.

8.2 A reduction

Suppose A is an algorithm that learns propositional Horn sentences using equivalence queries with clauses as counterexamples. We first use Littlestone's result [26] to transform A into an algorithm A' that predicts whether a Horn clause is implied by an unknown Horn sentence. We then use A' to construct an algorithm A'' to predict acyclic boolean circuits.

In a prediction setting, there is an unknown function f_* mapping some domain X to $\{0, 1\}$. In one *trial*, the prediction algorithm is given an arbitrary element $x \in X$ and must predict whether $f_*(x) = 1$ After the prediction is made, the correct value of $f_*(x)$ is supplied. The prediction process consists of an indefinite number of trials.

Each time there is an incorrect prediction, it is counted as one *error of prediction.* Littlestone [26] has shown that a polynomial time algorithm for exact identification using equivalence queries can be transformed into a polynomial time prediction algorithm such that the number of errors of prediction is bounded by the number of equivalence queries used by the original algorithm. We make use of that transformation here to obtain A' from A.

Suppose that A is a polynomial time algorithm to identify propositional Horn sentences using equivalence queries with clauses as counterexamples. In other words, suppose that there is some (non-decreasing) polynomial $p_1(x)$ such that for any propositional Horn sentence ϕ, if A has access to equivalence queries about ϕ (returning clauses as counterexamples), then in time and queries bounded by $p_1(size(\phi))$, A will halt and output a Horn sentence logically equivalent to ϕ.

Then Littlestone's result shows that there is a polynomial time algorithm A' to solve the problem of predicting whether Horn clauses are implied by an unknown Horn sentence. More specifically, for each propositional Horn sentence ϕ there is an associated prediction problem f_ϕ defined on the set of all Horn clauses C over the variables of ϕ by

$$f_\phi(C) = 1 \text{ iff } \phi \vdash C.$$

And there is a polynomial time algorithm A' to solve these prediction problems. That is, there is some (non-decreasing) polynomial $p_2(x, y)$ such that for any propositional Horn sentence ϕ, if we run A' on the prediction problem f_ϕ, a trial with clause C will take time at most $p_2(size(C), size(\phi))$, and no more than $p_1(size(\phi))$ errors of prediction will be made.

From A' we obtain a polynomial time algorithm to predict boolean circuits. Given an acyclic circuit κ with n inputs $x_1, \ldots, x_n$, the associated prediction problem is given by f_κ defined on all assignments a to the variables $x_1, \ldots, x_n$ by

$$f_\kappa(a) = 1 \text{ iff } \kappa(a) = 1.$$

In the following algorithm, κ_* denotes the unknown circuit and n denotes the number of inputs to κ_*.

The circuit prediction algorithm A''

1. Run algorithm A' until it requests a clause to predict.
2. Request the next assignment to predict; let a denote it.
3. Give A' the clause $(A_a \rightarrow Y_{n+1})$ to predict. Output its prediction as the prediction for a.
4. Request the correct value of $\kappa_*(a)$. Give that reply to A' and go to step (1).

We claim that A'' must be a polynomial time algorithm to predict circuits. That is, there are polynomials $p_3(x, y)$ and $p_4(x)$ such that for any acyclic circuit κ, if A'' is run with the prediction problem f_κ then in any trial with assignment a the time used by A'' is bounded by $p_3(size(a), size(\kappa))$, and over all trials the total number of errors of prediction is bounded by $p_4(size(\kappa))$.

To see this, note that when A'' is run with the prediction problem f_κ, then it in effect runs A' with the prediction problem f_{ϕ_κ}. Only a subset of the domain of f_{ϕ_κ} is used, but the bound on errors of prediction holds for any sequence of trials.

If we consider a trial of A'' with assignment a, then the trial of A' is with the clause $C = (A_a \rightarrow Y_{n+1})$. The time for this trial of A' is bounded by $p_2(size(C), size(\phi_\kappa))$. Moreover, the size of C is bounded by a polynomial in the size of a, and the size of ϕ_κ is bounded by a polynomial in the size of κ. Thus the time used by A'' in this trial is bounded by a polynomial in the sizes of a and κ, as required. The total number of errors of prediction for A'' will be equal to the total number of errors of prediction for A', which was shown above to be bounded by $p_1(size(\phi_\kappa))$, which in turn is bounded by a polynomial in the size of κ, as required. Thus we have shown the following.

Lemma 14 *If there were a polynomial time algorithm to learn acyclic propositional Horn sentences using equivalence queries with clauses as counterexamples, there would be a polynomial time algorithm to predict acyclic circuits.*

(Pitt and Warmuth [29] have given a formal treatment of reductions in the context of prediction problems.)

What are we to make of this lemma? Work in cryptography suggests that some acyclic circuits are very hard to predict indeed. For example, Goldreich, Goldwasser, and Micali [15] have shown that the assumption that a polynomial time 1-way function exists implies that there are pseudo-random functions with "small" (that is, polynomial size) circuits that are very difficult to predict.

For a more concrete example, we consider the problem of testing whether a number x is a square modulo a number N that is a product of two large (unknown) primes P and Q. This problem, called "quadratic residuosity", is the basis of several cryptographic schemes.

Define the prediction problem $f_N(x)$ for all $x \in [1, N-1]$ by $f_N(x) = 1$ if and only if x is a square modulo N. The *size* of the problem is n, the number of bits in N. If P and Q are known, the problem reduces to testing whether $x^{(P-1)/2} \equiv 1$ modulo P and $x^{(Q-1)/2} \equiv 1$ modulo Q, which can easily be done with a circuit of size polynomial in n.

Blum [9] has shown that if quadratic residuosity could be predicted in polynomial time with a polynomial number of errors of prediction, then quadratic residuosity could be computed by a randomized polynomial time algorithm with a very high probability of correctness. This suggests (though does not prove) that there is no such polynomial time algorithm to predict quadratic residuosity.

Note that the problem of whether equivalence queries with assignments as counterexamples suffice for polynomial time identification of propositional Horn sentences is open. Intuitively, assignments would provide "a lot more" information than clauses in the reduction above, namely, the output values of all the gates in the circuit in addition to the input value and the final output value.

9 Appendix C: Closed sets and self-contained "pieces" of derivations

We show that a closed subset of a derivation-DAG corresponds to a self-contained "piece" of the derivation-DAG. Let D be any directed acyclic graph. Then S is a *closed subset* of

D if and only if S is a subset of the nodes of D such that for every node x in S, if (y, x) is an edge of D, then y is also in S. Thus S is closed under the relation of predecessor in D. A closed subset S of D is called *nontrivial* if it is not empty and does not contain all the nodes of D. (An efficient algorithm for finding a closed subset of maximum weight in a weighted DAG is given in [19].)

By induction, if there is a directed path from node y to an element x in S, then y must also be in S. Hence if S is a nontrivial closed subset of D, then S must contain at least one of the source nodes of D, and must omit at least one of the sink nodes of D.

If S is a closed subset of D, the *boundary* of S, denoted $b(S)$, is the set of all those nodes x in S such that there is an edge from x to some node of D that is not in S. Intuitively, the boundary nodes are the only means of communication between the subset S and the rest of the graph.

As an example, consider a rooted tree in which each edge is directed from child to parent. Then each closed subset corresponds to a forest of subtrees of the tree, whose boundary is the set of roots of the subtrees in the forest.

Suppose D is a derivation-DAG of a clause $C = (A \rightarrow z)$ from a Horn sentence ϕ. Suppose S is a nontrivial closed subset of D. Then S contains at least one source node of D and omits the unique sink node, z. We argue that S specifies a "self-contained" portion of the derivation-DAG.

Lemma 15 *Let D be a derivation-DAG of $C = (A \rightarrow z)$ from ϕ. Let S be a closed nontrivial subset of D. Let D_1 denote the subgraph of D induced by the nodes S. Let $x \in b(S)$, and consider the subgraph $D_1(x)$ induced by the set of nodes y such that there is a directed path from y to x in D_1. $D_1(x)$ is a derivation-DAG of the clause $(A \rightarrow x)$ from ϕ.*

This means that every boundary node has a proof within D_1. To see this, note that the source nodes in $D_1(x)$ are a subset of the source nodes in D, that is, are a subset of A. Moreover, by construction there is a unique sink node x in $D_1(x)$, and for every node y in $D_1(x)$ there is a directed path from y to x in $D_1(x)$. Finally, suppose y is any non-source node of $D_1(x)$, and let P be the set of nodes in D with an edge into y and let P' be the set of nodes in $D_1(x)$ with an edge into y. Since D is a derivation-DAG, $(P \rightarrow x)$ is a clause of ϕ. Clearly, P' is a subset of P, since $D_1(x)$ is an induced subgraph of D. Moreover, P' is equal to P. If w has an edge into y in D, then since y is in S and S is closed, w is in S, and moreover, since there is a directed path from y to x in D_1, there is a directed path from w to x in D_1, so w, and the edge from w to y, are part of $D_1(x)$. Thus $(P' \rightarrow y)$ is a clause of ϕ, concluding the proof that $D_1(x)$ is a derivation-DAG of $(A \rightarrow x)$ from ϕ. Q.E.D.

What about the rest of the graph D? Is the information contained in the boundary nodes, together with A, enough to complete the proof of z from A? The following lemma shows this is true.

Lemma 16 *Let D be a derivation-DAG of $C = (A \rightarrow z)$ from ϕ. Let S be a nontrivial closed subset of D. Let D_2 denote the subgraph of D obtained by removing all the nodes in $S - b(S)$ and all their incident edges, and removing any edges between elements of $b(S)$. Then D_2 is a derivation-DAG of $((A \cup b(S)) \rightarrow z)$ from ϕ.*

Note that every boundary node $x \in b(S)$ is a source node in D_2. The reason for this is that we have removed edges between the boundary nodes, and no node not in S can have an edge into a node of S. The node z is in D_2 because S does not contain z.

Consider a node x in D_2. If x is not in $b(S)$, there is a directed path in D to z, and this path cannot contain any node of S, since otherwise x would be in S. Hence this path from x to z also exists in the graph D_2. If x is in $b(S)$, then there is an edge in D from x to some node y not in S. This edge also exists in D_2, and we have already shown that there must be a path from y to z in D_2, so there is a path from x to z in D_2.

Let x be any source node of D_2, and suppose x is not a source node in D. Then there must be a node y in D such that there is an edge (y, x) in D, and this edge is not in D_2. Such an edge is only deleted if y is an element of $S - b(S)$ or if y and x are both elements of $b(S)$. If y is an element of $S - b(S)$, then by the definition of a boundary node, y has no edges into elements outside S, so x must be in $b(S)$. Thus, in either case, $x \in b(S)$, so the source nodes are a subset of $(A \cup b(S))$.

Finally, suppose x is any non-source node of D_2. Let P denote the set of nodes with edges into x in D, and let P' denote the set of nodes with edges into x in D_2. Because D is a derivation-DAG with respect to ϕ, $(P \rightarrow x)$ is a clause of ϕ. Clearly, P' is a subset of P, because D_2 is a subgraph of D. Morever, P' is equal to P. To see this, note that since all the nodes in $b(S)$ are sources in D_2, x must not be an element of S. Hence if y is any node in P, y must be in $b(S)$ or not in S, so the edge (y, x) is not removed in constructing D_2. Thus $(P' \rightarrow x)$ is a clause of ϕ, which completes the proof that D_2 is a derivation-DAG of $((A \cup b(S)) \rightarrow z)$. Q.E.D.

Suppose we have a derivation-DAG D for an M-meaningful incorrect clause $C = (A \rightarrow z)$ from ϕ, and we divide D into two pieces D_1 and D_2 as above. For each $x \in b(S)$, we determine whether ϕ_* implies the clause $A \rightarrow x$, using a request for a hint query. Note that A is in M, so each of these clauses is M-meaningful. If the query for $(A \rightarrow x)$ is answered *no*, then we may recursively continue with the derivation-DAG $D_1(x)$. If none of these queries is answered *no*, then by closure of M under implication with respect to ϕ_*, the set $A \cup b(S)$ is in M, and we continue recursively with the derivation-DAG D_2.

Thus a reasonable generalization of Shapiro's approach is to try to divide D using a closed subset S such that $b(S)$ is relatively small, and the pieces, $D_1(x)$ for each $x \in b(S)$ and D_2, are each substantially smaller than D. Of course, this may not always be possible, but intuition suggests that any derivation in which it is not possible is pretty hard for a human to understand.

For the example in Section 5.5, the derivation-DAG of $(a_1 \wedge b_1) \rightarrow z$ splits into the subgraph D_1 induced on the nodes

$$a_1, b_1, a_2, b_2, \ldots, a_{n/2}, b_{n/2},$$

and the subgraph D_2 induced by the nodes

$$a_{n/2}, b_{n/2}, \ldots, a_n, b_n, z$$

with the boundary set $\{a_{n/2}, b_{n/2}\}$.

Further discussion of this problem will appear elsewhere.

10 Comments

As Mark Fulk independently observed, the approach of using hints can be applied to the problem of identifying context-free grammars using equivalence and nonterminal membership queries, defined in [2] In this case, a "hint" for the derivation of a terminal string from a nonterminal is the name of an intermediate nonterminal in the derivation. "Cyclic" grammars are the rule rather than the exception, so it will be important to find a reasonable way to deal with cyclicity.

In [2], there is the incorrect statement that Shapiro's logarithmic improvement applied to parse-DAGs instead of just to parse trees. Appendix C makes clear that the problem is more complicated for DAGs than for trees.

In this paper we have demonstrated a simple incremental polynomial time algorithm for learning acyclic propositional Horn sentences using two types of queries:

1. equivalence queries with clauses as counterexamples, and

2. requests for hints.

We have shown that the algorithm could be modified to use equivalence queries with assignments as counterexamples in place of queries of type (1).

We have shown that our learning algorithm restricts its queries to "meaningful" clauses, assuming that the set of "meaningful" clauses is given implicitly and satisfies some reasonable restrictions.

We emphasize that learning algorithms, to be practical, should be incremental in the sense that if they are started with a hypothesis that is "close" to the final one, the total learning time should be proportionately "small". We have given one method of quantifying this requirement in the domain of propositional Horn sentences.

We have shown how Pitt and Warmuth's ideas for reductions between prediction problems can be used to show that learning acyclic propositional Horn sentences using just equivalence queries with clauses as counterexamples is as hard (modulo a polynomially bounded reduction) as predicting arbitrary acyclic circuits.

11 Acknowledgements

My thanks to the quartet in grey: Udi Shapiro, Bob Nix, Lenny Pitt, and Phil Laird, for keeping me thinking all these years. Conversations with Manuel Blum and Lenny Pitt led to the material in Appendix B. The support of the National Science Foundation, grant IRI-8404226, is gratefully acknowledged.

References

[1] D. Angluin. *Debugging a DAG efficiently.* Technical Report, Yale University Computer Science Dept., TR-591, 1987.

[2] D. Angluin. *Learning k-bounded context-free grammars.* Technical Report, Yale University Computer Science Dept., TR-557, 1987.

[3] D. Angluin. *Learning k-term DNF formulas using queries and counterexamples.* Technical Report, Yale University Computer Science Dept., TR-559, 1987.

[4] D. Angluin. Learning regular sets from queries and counterexamples. *Information and Computation*, 75:87–106, 1987. Preliminary version appeared as YALEU/DCS/RR-464.

[5] D. Angluin. *Types of queries for concept learning.* Technical Report, Yale University Computer Science Dept., TR-479, 1986. Revised version, *Queries and concept learning*, submitted for publication.

[6] D. Angluin, W. Gasarch, and C. Smith. *Training sequences.* Technical Report, University of Maryland, CS-TR-1894, UMIACS-TR-87-37, 1987. Submitted for publication.

[7] D. Angluin and P. Laird. *Identifying k-CNF formulas from noisy examples.* Technical Report, Yale University Computer Science Dept., TR-478, 1986. Revised version, *Learning from noisy examples*, to appear in *Machine Learning.*

[8] P. Berman and R. Roos. Learning one-counter languages in polynomial time. In *Proc. 28th IEEE Symposium on Foundations of Computer Science*, pages 61–67, IEEE, 1987.

[9] M. Blum. Result on predicting quadratic residuosity. 1987. Personal communication.

[10] A. Blumer, A. Ehrenfeucht, D. Haussler, and M. Warmuth. Classifying learnable geometric concepts with the Vapnik-Chervonenkis dimension. In *Proc. 18th ACM Symposium on Theory of Computing*, pages 273–282, ACM, 1986.

[11] J. Cherniavsky and C. Smith. *Using telltales in developing program test sets.* Technical Report, Georgetown University, Dept. of Computer Science, TR-4, 1986.

[12] J. Cherniavsky and R. Statman. Testing and inductive inference: abstract approaches. 1987. Preprint, Georgetown University.

[13] U. Feige and A. Shamir. Learning in permutation groups (extended abstract). 1987. Preprint, Applied Mathematics Dept., The Weizmann Institute of Science.

[14] M. Garey and D. Johnson. *Computers and Intractability: A Guide to the Theory of NP-Completeness.* W. H. Freeman and Company, San Francisco, 1979.

[15] O. Goldreich, S. Goldwasser, and S. Micali. How to construct random functions. *J. ACM*, 33:792–807, 1986.

[16] D. Haussler. *Learning conjunctive concepts in structural domains.* Technical Report, University of California at Santa Cruz, Dept. of Computer Science, UCSC-CRL-87-1, 1987.

[17] D. Haussler. *Quantifying inductive bias in concept learning.* Technical Report, University of California at Santa Cruz, Dept. of Computer Science, UCSC-CRL-86-25, 1986.

[18] D. Haussler, N. Littlestone, and M. Warmuth. Expected mistake bounds for on-line learning algorithms. Preprint, University of California at Santa Cruz, April 1987.

[19] R. W. Irving, P. Leather, and D. Gusfield. An efficient algorithm for the optimal stable marriage. *J. ACM*, 34:532–543, 1987.

[20] N. D. Jones and W. T. Laaser. Complete problems for deterministic polynomial time. *Theor. Comp. Sci.*, 3:107–113, 1977.

[21] M. Kearns and M. Li. *Learning in the presence of malicious errors.* Technical Report, Harvard University, Center for Research in Computing Technology, TR-03-87, 1987.

[22] M. Kearns, M. Li, L. Pitt, and L. Valiant. On the learnability of boolean formulae. In *Proc. 19th ACM Symposium on Theory of Computing*, pages 285–295, ACM, 1987.

[23] K. Kelly and C. Glymour. *On convergence to the truth and nothing but the truth.* Technical Report, Carnegie Mellon University, Laboratory for Computational Linguistics, CMU-LCL-87-4, 1987.

[24] P. Laird. Inductive inference by refinement. In *Proc. of AAAI-86*, pages 472–476, AAAI, 1986.

[25] P. Laird. *Learning From Good Data and Bad.* PhD thesis, Yale University, 1987. Computer Science Dept. TR-551.

[26] N. Littlestone. Learning quickly when irrelevant attributes abound: a new linear-threshold algorithm. In *Proc. 28th IEEE Symposium on Foundations of Computer Science*, pages 68–77, IEEE, 1987.

[27] B. K. Natarajan. On learning boolean functions. In *Proc. 19th ACM Symposium on Theory of Computing*, pages 296–304, ACM, 1987.

[28] L. Pitt and L. Valiant. *Computational limitations on learning from examples.* Technical Report, Harvard University, Center for Research in Computing Technology, TR-05-86, 1986.

[29] L. Pitt and M. Warmuth. Reductions among prediction problems: on the difficulty of predicting automata (extended abstract). 1987. Preprint.

[30] R. Rivest and R. Schapire. Diversity-based inference of finite automata. In *Proc. 28th IEEE Symposium on Foundations of Computer Science*, pages 78–87, IEEE, 1987.

[31] R. Rivest and R. Schapire. Inference of visible simple assignment automata with planned experiments. 1987. Preprint, MIT Laboratory for Computer Science.

[32] R. Rivest and R. Schapire. A new approach to unsupervised learning in deterministic environments. In *Proc. of the 4th International Workshop on Machine Learning*, pages 364–375, Morgan Kaufmann Publishers, Inc., 1987.

[33] S. Rudich. Inferring the structure of a Markov chain from its output. In *Proc. 26th IEEE Symposium on Foundations of Computer Science*, pages 321–326, IEEE, 1985.

[34] C. Sammut and R. Banerji. Learning concepts by asking questions. In *Machine Learning, Vol. II*, pages 167–191, Morgan Kaufmann Publishers, Inc., 1986.

[35] E. Shapiro. *Algorithmic Program Debugging.* PhD thesis, Yale University, 1982. Published by MIT Press, 1983.

[36] L. Valiant. Learning disjunctions of conjunctions. In *Proc. 9th IJCAI*, pages 560–566, IJCAI, 1985.

[37] L. G. Valiant. A theory of the learnable. *C. ACM*, 27:1134–1142, 1984.