Yale University
Department of Computer Science

Types of queries for concept learning

Dana Angluin, Yale University

YALEU/DCS/TR-479
June 1986

Research funded in part by the National Science Foundation, DCR-8404226.

Abstract

We consider the problem of identifying a concept, that is, an unknown subset L_* of a universal set U, using a variety of sources of information about L_*. The information sources we study are: random sampling and queries that test equivalence, membership, subset, superset, and disjointness. We give examples of identification problems that are efficiently solvable using different collections of these information sources, including regular languages, some classes of context-free languages, pattern languages, monotone DNF formulas, and CNF formulas with a restricted number of clauses or of literals per clause.

Contents

Types of queries for concept learning

Dana Angluin, Yale University

1 Introduction

We consider the problem of identifying an unknown set L_* from some countable population $L_1, L_2, \ldots$ of subsets of a universal set U. The primary examples we consider are sets of strings generated by grammars or recognized by automata, and sets of truth-value assignments that satisfy propostional formulas from some particular class. This problem has been considered in the context of inductive inference, in which the source of information about the unknown language is a sequence of arbitrarily chosen elements of U, each one marked as to whether it is an element of L_*. (The sequence may also be an arbitrarily chosen sequence of elements of L_*, in which case the problem is one of *identification from positive data.*) For a comprehensive survey of inductive inference, see [4].

In this paper, we explore the effects of other kinds of information about the unknown set on the feasibility of identification. This study is motivated by Shapiro's work on automated aids for debugging Prolog programs [15,16,17], by Valiant's work on learning concepts from randomly chosen examples and other types of information [19], and by the discovery of an efficient identification procedure for regular sets using equivalence and membership queries [2]. The goal is to classify several types of queries and give examples of identification problems in which they are useful. Such a classification cannot claim to be exhaustive, but it has already raised some interesting questions.

1.1 Types of queries

The types of queries we consider are:

1. Equivalence. The input is a set L and the output is *yes* if $L = L_*$ and *no* if $L \neq L_*$. In addition, if the answer is *no*, an element $x \in L \oplus L_*$ is returned.

Research funded in part by the National Science Foundation, DCR-8404226.

2. Membership. The input is an element $x \in U$ and the output is *yes* if $x \in L_*$ and *no* if $x \notin L_*$.

3. Subset. The input is a set L and the output is *yes* if $L \subseteq L_*$ and *no* otherwise. In addition, if the answer is *no*, an element $x \in L - L_*$ is returned.

4. Superset. The input is a set L and the output is *yes* if $L \supseteq L_*$ and *no* otherwise. In addition, if the answer is *no*, an element $x \in L_* - L$ is returned.

5. Disjointness. The input is a set L and the output is *yes* if $L \cap L_*$ is empty and *no* otherwise. In addition, if the answer is *no*, an element $x \in L \cap L_*$ is returned.

In the cases of equivalence, subset, superset, and disjointness queries, the returned element x is called a *counter-example.* The particular selection of a counter-example is assumed to be arbitrary, that is, a successful identification method must work no matter what counter-example is returned. We shall also consider *restricted* versions of each of these queries, for which the responses are just *yes* and *no*, with no counter-example provided.

In general, we think of the input L as restricted to one of the hypotheses $L_1, L_2, L_3 \ldots$ in the original hypothesis space. The effects of this restriction are considered in more detail below.

Is this a complete set of queries? One notion of "completeness" is the following. An *E-query* is defined to have two inputs: a set L and a boolean expression over L and L_*, considered as representing a set X of elements. The output is *empty* if X is empty and is an element $x \in X$ otherwise.

One E-query that cannot be constructed from the above queries is "Exhaustiveness", that is, whether the complement of $L \cup L_*$ is empty or not. If we add exhaustiveness to subset, superset, and disjointness, and L is permitted to be the empty set or U, then any E-query can be answered by a sequence of these four basic types of queries. (This is a kind of truth-table reducibility.) However, exhaustiveness does not seem to come up in natural contexts, so it is omitted from the basic list.

1.2 Exact identification

We shall consider two criteria of successful identification, exact and probabilistic. An identification method *exactly identifies* a set L_* with access to certain types of queries if it always halts and outputs a set L such that $L = L_*$. Note that this is

not a limiting criterion of identification – the identification method is allowed one guess, and that guess must be exactly right.

1.3 Probabilistic identification

Probabilistic identification is defined as follows. There is some probability distribution D on the universal set U. There is a *sampling oracle* $EX()$ which has no input. Whenever $EX()$ is called, it independently draws a element $x \in U$ according to the distribution D and returns the pair $\langle x, s \rangle$, where $s = +$ if $x \in L_*$ and $s = -$ otherwise. Successful identification is parameterized with respect to two small positive quantities, the accuracy parameter ϵ and the confidence parameter δ. We define a notion of the difference between two sets L_1 and L_2 with respect to the probability distribution as

$$d(L_1, L_2) = \sum_{x \in L_1 \oplus L_2} Pr(x).$$

The probability of getting an element in one but not the other of the two sets L_1 and L_2 in one call to $EX()$ is precisely $d(L_1, L_2)$.

An identification method is said to *probably approximately correctly identify* L_* if it always halts and outputs a set L such that

$$Pr(d(L, L_*) \leq \epsilon) \geq 1 - \delta.$$

That is, with high probability (quantified by δ) there is not too much difference (quantified by ϵ) between the conjectured set and the unknown set.

In general if we consider only deterministic information from a subset of the queries listed above we shall be interested in exact identification, but if the $EX()$ oracle is used, we shall be interested in probably approximately correct identification (abbreviated *pac*-identification hereafter). We now consider the different types of information individually.

2 Equivalence and sampling

2.1 Exhaustive search

If the only source of information available to us about the unknown language L_* is equivalence queries, then one strategy is to enumerate the hypotheses $L_1, L_2, \ldots$, querying each one until we get an answer of *yes* for some L_i, at which point we halt and output L_i. This achieves exact identification.

If queries are restricted to the hypothesis space, $L_1, L_2, L_3 \ldots$, then exhaustive search is in some cases the best that can be done. Suppose the hypothesis space is the set of singleton subsets of the set of all 2^n binary strings of length n. The following adversary will force any method of exact identification using equivalence, membership, subset, and disjointness queries to make $2^n - 1$ queries in the worst case.

The adversary maintains a set S of all the uneliminated binary strings of length n. Initially S contains all 2^n binary strings of length n. As long as S contains at least two distinct strings, the adversary proceeds as follows. If the next query is a membership query with the string x, then the adversary answers with *no*. If the next query is an equivalence or subset query with the singleton set $\{x\}$, then the adversary answers with *no* and the counter-example x. If the next query is a disjointness query with the singleton set $\{x\}$, then the adversary answers with *yes*. In each case, if x is a member of S, the adversary removes it from S.

It is not difficult to see that the responses of the adversary are compatible with the unknown hypothesis being $\{x\}$ for any element x still in S, and at most one element is removed from S with each query. Thus to be correct the algorithm must make at least $2^n - 1$ queries. If superset queries are permitted, a single one will disclose the unknown set.

2.2 Some general lower bound techniques

In this section we generalize the techiques of the preceding section to give some properties of subclasses of the hypothesis space that force exact identification algorithms to do exhaustive search over those subclasses if only certain types of queries are available.

Lemma 1 *Suppose the hypothesis space contains a class of distinct sets $L_1, \ldots, L_N$, and a set $L_\cap$ such that for any pair of distinct indices i and j,*

$$L_i \cap L_j = L_\cap.$$

Then any algorithm that exactly identifies each of the hypotheses L_i using restricted equivalence, membership, and subset queries (for the whole hypothesis space) must make at least $N - 1$ queries in the worst case.

We construct an adversary as follows. The set S is initially the set of all L_i for $i = 1, 2, \ldots, N$. As long as S contains at least two elements, the adversary answers queries according to the following strategy. If the next query is a restricted equivalence query with the hypothesis L, the reply is *no* and L is removed from S

if it is present. If the next query is a membership query with the element x, then if $x \in L_\cap$ the reply is *yes*. Otherwise, the reply is *no*, and the (at most one) element L_i of S containing x is removed from S. If the next query is a subset query with the hypothesis L, then if L is a subset of $L_\cap$, the reply is *yes*. Otherwise, the reply is *no* and any element $x \in L - L_\cap$ is selected as the counter-example. The (at most one) element L_i of S containing x is removed from S.

It is clear that at any point the replies to the queries are compatible with any of the remaining L_i in S being the unknown hypothesis, so a correct exact identification algorithm must reduce the cardinality of S to at most one. Moreover, each query removes at most one element from the set S, so $N - 1$ queries are required in the worst case, which proves Lemma 1.

If it happens that the "intersection set" $L_\cap$ is not itself an element of the hypothesis space, then the above result can be strengthened to cover unrestricted equivalence queries.

Lemma 2 *Suppose the hypothesis space contains a class of distinct sets $L_1, \ldots, L_N$, and there exists a set $L_\cap$ which is NOT a hypothesis, such that for any pair of distinct indices i and j,*

$$L_i \cap L_j = L_\cap.$$

Then any algorithm that exactly identifies each of the hypotheses L_i using equivalence, membership, and subset queries (for the whole hypothesis space) must make at least $N - 1$ queries in the worst case.

Let S initially contain all the sets L_i for $i = 1, 2, \ldots, N$. As long as S contains at least two elements, the adversary answers queries as follows. If the next query is an equivalence query with the hypothesis L then the reply is *no* and a counter-example x is chosen from $L \oplus L_\cap$. (This is possible since $L_\cap$ is not a hypothesis, so L cannot be equal to it.) If x is from $L - L_\cap$, the (at most one) element L_i of S containing x is removed from S. If the next query is a membership query with the element x, then if $x \in L_\cap$ the reply is *yes*. Otherwise, the reply is *no*, and the (at most one) element L_i of S containing x is removed from S. If the next query is a subset query with the hypothesis L then if L is a subset of $L_\cap$ then the reply is *yes*. Otherwise, the reply is *no* and an element $x \in L - L_\cap$ is selected to be the counter-example. The (at most one) element L_i in S containing x is removed from S.

It is clear that at any point the replies to the queries are compatible with any of the remaining L_i in S being the unknown hypothesis, so a correct exact identification algorithm must reduce the cardinality of S to at most one. Moreover, each query removes at most one element from the set S, so $N - 1$ queries are required in the worst case, which proves Lemma 2.

There are also dual results for equivalence, membership, and superset queries, which we state without proof.

Lemma 3 *Suppose the hypothesis space contains a class of distinct sets $L_1, \ldots, L_N$, and a set $L_\cup$ such that for any pair of distinct indices i and j,*

$$L_i \cup L_j = L_\cup.$$

Then any algorithm that exactly identifies each of the hypotheses L_i using restricted equivalence, membership, and superset queries (for the whole hypothesis space) must make at least $N - 1$ queries in the worst case.

Lemma 4 *Suppose the hypothesis space contains a class of distinct sets $L_1, \ldots, L_N$, and there exists a set $L_\cup$ which is NOT a hypothesis, such that for any pair of distinct indices i and j,*

$$L_i \cup L_j = L_\cup.$$

Then any algorithm that exactly identifies each of the hypotheses L_i using equivalence, membership, and superset queries (for the whole hypothesis space) must make at least $N - 1$ queries in the worst case.

2.3 A logarithmic strategy

If the input L to an equivalence query is NOT required to be a member of the hypothesis space, there is a general strategy using only equivalence queries that requires many fewer queries than exhaustive search. Suppose the hypothesis space is finite, of cardinality N. Then instead of making $N - 1$ queries, we may make $\lfloor \log N \rfloor$ queries, as follows.

A set C is maintained of all the hypotheses compatible with all the counter-examples seen so far. (C initially contains all N hypotheses.) If C contains just one element, then halt and output that element. Otherwise, define

$$M_C = \{x \in U : \text{ at least } |C|/2 \text{ elements of } C \text{ contain } x\}.$$

Thus, an element x is included in M_C if and only if at least half the remaining compatible hypotheses include x. Then we make an equivalence query with M_C as input. If the answer is *yes*, we halt and output M_C. Otherwise, we receive a counter-example x.

If $x \in M_C$, we remove from C every hypothesis L_i that includes x; otherwise, we remove from C every hypothesis L_i that does not include x. Then the process above is iterated. Note that we only make queries when C contains at least two

distinct elements and every query answered *no* must reduce the cardinality of C by at least one half, so a total of $\lfloor \log N \rfloor$ queries suffices. This result is a special case of the results of Barzdin and Freivalds on minimizing the number of changes of hypothesis [5].

This indicates that the structure of the hypothesis space has a strong influence on the number of equivalence queries required to do exact identification, when the queries are restricted to the hypothesis space. Of course we shall also be interested in the computational feasibility of the query strategy.

2.4 Comparison of equivalence and sampling

An identification method that uses equivalence queries and achieves exact identification may be modified to achieve *pac*-identification using calls to $EX()$ instead of equivalence queries. The idea is instead of asking an equivalence query about L, the identification method calls $EX()$ a number of times and checks to see that L is compatible with each pair $\langle x, s \rangle$ returned by $EX()$. If not, then the identification method proceeds as though the equivalence query had returned the answer *no* with x as a counter-example. If L is compatible with all the samples drawn, then the identification method proceeds as though the equivalence query had returned the answer *yes*.

Suppose that when an equivalence query returns *yes*, the identification method simply halts and outputs the language L. If the identification method makes

$$q_i = \lceil \frac{1}{\epsilon}(\ln \frac{1}{\delta} + i \ln 2) \rceil$$

calls to the $EX()$ oracle in place of the i-th equivalence query then the probability that the identification method will output a set L such that $d(L, L_*) \geq \epsilon$ is at most $(1-\epsilon)^{q_i}$. Thus, the probability that at any stage the identification method will output a hypothesis that is not an ϵ-approximation of L_* is at most

$$\begin{aligned} \sum_{i=1}^{\infty}(1-\epsilon)^{q_i} &\leq \sum_{i=1}^{\infty} e^{-\epsilon q_i} \\ &\leq \sum_{i=1}^{\infty} \frac{\delta}{2^i} \\ &\leq \delta. \end{aligned}$$

Thus the modified method achieves *pac*-identification of L_*.

An example of the modification of an exact method to identify regular sets using equivalence and membership queries to a probabilistic method using $EX()$

and membership queries may be found in [2]. There it is argued that if the source of information about L_* is a domain expert, then it may be unreasonable to expect correct answers to equivalence queries, but this probabilistic equivalence-testing may be a practical substitute. Stochastic equivalence-testing was a feature of the method of Knobe and Knobe for identifying context-free grammars [11].

What about the converse? Can the use of an $EX()$ oracle and *pac*-identification criterion be replaced by the use of equivalence queries and exact identification? Not if it is required to preserve the efficiency of the method. Consider again the problem of identifying singleton subsets of the set of all binary strings of length n. Recall that an adversary could force a strategy using only equivalence queries to make $2^n - 1$ queries in the worst case.

However, to achieve *pac*-identification in this domain, it suffices to make

$$q = \lceil \frac{2}{\epsilon}(\ln \frac{1}{\delta} + n \ln 2) \rceil$$

calls to $EX()$. (Note that q grows linearly in n for fixed ϵ and δ.) Either at least one of these calls will return a positive example $\langle x, + \rangle$, in which case the correct answer is $\{x\}$, or none of them do, in which case we output any set $\{y\}$ such that the string y has not appeared among the negative examples.

To see that this procedure achieves *pac*-identification we show that with high probability every string of probability at least $\epsilon/2$ is drawn in q calls to $EX()$. Thus if the "correct" string x is not drawn and we instead output y (which is also not drawn), with high probability

$$d(\{y\}, \{x\}) \leq \epsilon/2 + \epsilon/2 \leq \epsilon.$$

Suppose x is a string with probability at least $\epsilon/2$. The probability that x is not drawn in q calls to $EX()$ is at most $(1 - \epsilon/2)^q$, so the probability that there exists any string of probability at least $\epsilon/2$ that is not drawn in q calls to $EX()$ is at most

$$\begin{aligned} 2^n(1 - \epsilon/2)^q &\leq 2^n e^{\frac{-\epsilon q}{2}} \\ &\leq \delta. \end{aligned}$$

Thus *pac*-identification with oracle $EX()$ is in general easier to achieve than exact identification with equivalence queries.

2.5 k-CNF formulas and k-DNF formulas

Upper bounds. Let $CNF(n, k, c)$ denote the set of propositional formulas over the n variables $x_1, \ldots, x_n$ with at most k literals per clause and at most c clauses.

(If k is $*$ then there is no bound on the number of literals per clause, and similarly for c.) The $EX()$ oracle returns pairs $\langle a, s \rangle$ where a is an assignment of truth-values to the variables $x_1, \ldots, x_n$ and $s = +$ if this assignment satisfies an unknown formula ϕ_* and $s = -$ otherwise.

Valiant [19] has shown that there is a method that runs in time polynomial in n^k, $1/\epsilon$, and $\log 1/\delta$ that achieves *pac*-identification of the formulas in $CNF(n, k, *)$. The method is to draw a certain number of samples from $EX()$ and then to output the conjunction of all those clauses C over n variables with at most k literals per clause such that for every positive sample $\langle a, + \rangle$, $a(C)$ is true.

There is an analogous method that runs in time polynomial in n^k, uses equivalence queries, and achieves exact identification of the formulas in $CNF(n, k, *)$. Initially let ϕ be the conjunction of all clauses C over n variables with at most k literals per clause. (There are no more than $(2n + 1)^k$ such clauses.) Iterate the following process until the equivalence query returns *yes*, at which point halt and output ϕ. Test ϕ for equivalence with ϕ_*. If it is inequivalent, the response will be a counter-example a, which must satisfy ϕ_* but not ϕ. There will be at least one clause C in ϕ such that $a(C)$ is false. Remove from ϕ all such clauses C and iterate. It is clear that we remove at least one clause for each negative answer to an equivalence query and must arrive at a formula equivalent to ϕ_* by the time we remove all the clauses, so the claim follows.

By logical duality, there are similar methods for exact and *pac*-identification of disjunctive normal form formulas over n variables with at most k literals per term. For a number of interesting examples of *pac*-identification, and a general characterization of when it is possible, see the paper of Blumer et al.[7].

Lower bounds. We may apply Lemma 1 to show an exponential lower bound on any algorithm that exactly identifies all 1-CNF formulas with n variables using restricted equivalence, membership, and subset queries. To do this, we consider the class of all formulas of the form

$$P_1 \wedge P_2 \wedge \ldots \wedge P_n,$$

where each P_i is either x_i or $\neg x_i$. There are 2^n such formulas, each one a 1-CNF formula satisfied by exactly one assignment, and no two formulas are satisfied by the same assignment. This class satisfies the hypotheses of Lemma 1, which implies that any algorithm that exactly identifies every formula in this class using restricted equivalence, membership, and subset queries must make $2^n - 1$ queries in the worst case.

Dually, we may consider the class of 1-DNF formulas of the form

$$P_1 \vee P_2 \vee \ldots \vee P_n,$$

where each P_i is either x_i or $\neg x_i$. Every formula in this class is satisfied by all but one assignment which is unique to that formula, so the hypotheses of Lemma 3 are satisfied. This implies that any algorithm that exactly identifies every formula in this class using restricted equivalence, membership, and superset queries must make $2^n - 1$ queries in the worst case.

3 Membership queries

A membership query returns one bit of information: whether or not the queried element is a member of the unknown set L_*. If the source of information is a domain expert, it seems reasonable to ask the expert to classify cases that the system may generate in the course of trying to learn a particular concept. (However, in a practical case, say X-rays of potential tumors, it may be difficult for the system to generate fully instantiated cases (simulated X-rays) that embody the particular features the system has decided are relevant. In such a case, subset, superset, or disjointness queries using a higher-level description language may actually prove more reasonable.)

Shapiro uses both membership and disjointness queries in his system for automatic debugging of Prolog programs [15,16,17]. See Section 5.1 for a description of the use of queries in his system.

3.1 Monotone DNF formulas

We consider the class of monotone DNF formulas, that is, disjunctive normal form formulas over n variables that contain no negative literals. The main result to be proved in this section is the following.

Theorem 5 *There is an algorithm that exactly identifies every monotone DNF formula ϕ_* over n variables that uses equivalence and membership queries and runs in time polynomial in n and the number of terms of ϕ_*.*

Valiant [19] gives an algorithm to *pac*-identify an unknown DNF formula ϕ_* over n variables using sampling and restricted subset queries, and shows that its running time can be bounded by a polynomial in n and the number of prime implicants of ϕ_*. (A *prime implicant* of a formula ϕ is a satisfiable product t of literals such that t implies ϕ but no term obtained from t by deleting one literal implies ϕ.) Since the number of prime implicants of a monotone DNF formula ϕ_* is bounded by the number of terms of ϕ_*, it remains to replace sampling by equivalence queries and restricted subset queries by membership queries.

It is easy to modify Valiant's method to achieve exact identification using equivalence queries in place of sampling. For completeness, the resulting algorithm is briefly described.

Initially the hypothesis ϕ is the empty formula, equivalent to false. We test ϕ using an equivalence query. If the reply is *yes*, we halt and output ϕ. Otherwise, the counter-example will be an assignment a that satisfies ϕ_* and not ϕ.

From a we search for a new prime implicant of ϕ_*. Let t be the unique minterm satisfied by a, that is, a product containing each variable or its complement, where t contains x_i if and only if $a(x_i)$ is true. Then we attempt to "reduce" t. For each t' obtained by deleting one literal from t, we use a subset query to test whether t' implies ϕ_*, i.e., whether the set of assignments satisfying t is a subset of the set of assignments satisfying ϕ_*. If so, we replace t by t' and continue the reduction process. Eventually we arrive at a term t such that t implies ϕ_*, but no term obtained from t by deleting one literal implies ϕ_*, that is, t is a prime implicant of ϕ_*. Note that the counter-example a still satisfies t. We now replace the hypothesis ϕ by $\phi + t$ and iterate from the equivalence test.

Clearly each prime implicant requires one equivalence query and at most n subset queries, so the running time of the algorithm is clearly bounded by a polynomial in n and the number of prime implicants of ϕ_*.

In the case of a monotone DNF formula ϕ_*, the process of searching for a prime implicant starts from a different initial term t, namely, the product of all those positive literals x_i such that $a(x_i)$ is true. From there the search is the same, but note that every subset query now concerns a monotone term t.

Finally, we note that in the case of monotone DNF formulas, restricted subset queries are reducible to membership queries. To test whether a monotone DNF formula ϕ implies ϕ_*, it suffices to be able to test whether each of the terms t of ϕ imply ϕ_*. To test whether the monotone term t implies ϕ_*, construct the assignment a that is true on just those variables x_i that appear in t and test a for membership in the set of assignments that satisfy ϕ_*. If the reply is *yes* then t implies ϕ_*, otherwise t does not imply ϕ_*.

This concludes the proof of Theorem 5.

The importance of counter-examples. The counter-examples provided by the equivalence queries are essential to the efficiency of the above algorithm. (Recall that a restricted equivalence query returns only *yes* or *no* and no counter-example.)

Theorem 6 *For each positive integer n there is a class $\mathcal{D}$ of monotone DNF formulas with $2n$ variables and $n + 1$ terms such that any algorithm that exactly*

4.1 The pattern languages

The pattern languages were introduced in [1]. Let A be a finite alphabet of constant symbols, and X a countably infinite alphabet of variable symbols. (We assume that A contains at least two distinct symbols.) A *pattern* is a finite string of symbols from $A \cup X$. The *language of a pattern* p, denoted $L(p)$, is the set of all strings over the alphabet A obtained by substituting non-empty strings of constant symbols for the variable symbols of p. For example, if $p = 122x5yyx3$, then the language of p includes the strings 12205111103 and 122001512120013, but not the strings 12253 or 1221560601113.

Efficient exact identification with superset queries. Superset queries alone suffice for efficient identification of the pattern languages; equivalence queries are not required for correct termination.

Theorem 7 *There is an algorithm that exactly identifies the class of languages defined by patterns of length n that uses restricted superset queries and runs in time polynomial in n.*

We assume that there is an unknown pattern p_*. The goal is to find a pattern equivalent to p_* by asking queries of the form $L(p) \supseteq L(p_*)$? for any pattern p. The replies will be either *yes* or *no*, with no counter-example supplied.

Note that if p is a pattern of length n then $L(p)$ contains at least one string of length n and contains only strings of length n or greater. Also, $L(x_1x_2\cdots x_n)$ is precisely all those strings of symbols from A of length n or greater. Thus we can determine the length of the unknown pattern p_* by using superset queries on the patterns x_1, x_1x_2, $x_1x_2x_3$, and so on, to find the least $k+1$ such that $L(x_1x_2\cdots x_{k+1})$ is not a superset of $L(p_*)$. Then the length of p_* is k.

Having determined the length of p_* is k, we can determine the positions and values of its constant symbols as follows. For each $a \in A$ and $i = 1, 2, \ldots, k$, query whether

$$L(x_1 \cdots x_{i-1} a x_{i+1} \cdots x_k) \supseteq L(p_*).$$

If so, then the i-th symbol of p_* is the constant symbol a. If for no a is this query answered *yes*, then the i-th symbol of $L(p_*)$ is a variable symbol.

Knowing the length of p_* and the positions and values of its constant symbols, it remains only to determine for each pair of positions containing variables whether the variables are the same or not. For each pair $i < j$ of positions of variable symbols in p_*, we query whether $L(p_{i,j})$ is a superset of $L(p_*)$, where $p_{i,j}$ is obtained from $x_1x_2\cdots x_k$ by substituting the new variable x for both x_i and x_j. If the

answer is *yes*, then positions i and j of p_* contain the same variable; otherwise, they contain different variables.

Once all these tests have been completed, a pattern p equivalent to p_* can be constructed and output. The number of queries and the computation time for this method are easily seen to be bounded by a polynomial in the length of p_*.

This concludes the proof of Theorem 7.

A lower bound for exact identification of pattern languages. We may apply Lemma 2 to obtain an exponential lower bound on exact algorithms to identify the pattern languages.

Theorem 8 *Any algorithm that exactly identifies all the patterns of length n using equivalence, membership, and subset queries must make at least $2^n - 1$ queries in the worst case.*

If p is a pattern that contains no variables, then $L(p) = \{p\}$. Consider the class of singleton sets of binary strings of length n. This is a class of 2^n pattern languages with the property that the intersection of any distinct pair of them is the empty set, which is not a pattern language. Hence this class satisfies the hypotheses of Lemma 2, which implies that any algorithm that exactly identifies all the patterns in this class using unrestricted equivalence, membership, and subset queries must make at least $2^n - 1$ queries, which proves Theorem 8.

Note that these two results on the pattern languages leave open the question of how useful disjointness queries might be to their identification.

4.2 k-term DNF formulas and k-clause CNF formulas

We consider the class $DNF(n, *, k)$ of DNF formulas over n variables with at most k terms. As we showed in Section 3.1, there is an algorithm that exactly identifies any DNF formula ϕ_* over n variables using equivalence and restricted subset queries that runs in time polynomial in n and the number of prime implicants of ϕ_*. The following bound on the number of prime implicants of any formula from $DNF(n, *, k)$ is proved in the Appendix.

Lemma 9 *Every formula ϕ_* from $DNF(n, *, k)$ has fewer than 3^k prime implicants.*

An immediate corollary is the following.

Theorem 10 *The class $DNF(n,*,k)$, consisting of DNF formulas over n variables with at most k terms, can be exactly identified using equivalence and restricted subset queries in time polynomial in n^k. Dually, $CNF(n,*,k)$ can be exactly identified using equivalence and restricted superset queries in time polynomial in n^k.*

Lower bounds on these two problems are given in the following two theorems.

Theorem 11 *There exists a class $\mathcal{D}$ of DNF formulas with n variables and one term such that any algorithm that exactly identifies every formula from $\mathcal{D}$ using restricted equivalence, membership, and subset queries must make at least $2^n - 1$ queries in the worst case.*

The class $\mathcal{D}$ consists of the 2^n formulas of the form

$$P_1 \wedge P_2 \wedge \ldots \wedge P_n,$$

where each P_i is either x_i or $\neg x_i$. Each such formula is satisfied by exactly one assignment, and no two formulas are satisfied by a common assignment. Applying Lemma 1, the result follows.

Theorem 12 *There exists a class $\mathcal{C}$ of CNF formulas with n variables and one clause such that any algorithm that exactly identifies every formula from $\mathcal{C}$ using restricted equivalence, membership, and superset queries must make at least $2^n - 1$ queries in the worst case.*

The class $\mathcal{C}$ consists of the 2^n formulas of the form

$$(P_1 \vee P_2 \vee \ldots \vee P_n),$$

where each P_i is either x_i or $\neg x_i$. Each such formula is satisfied by all but one assignment, and for any pair of formulas, the union of their sets of satisfying assignments is all assignments. Applying Lemma 3, the result follows.

5 Disjointness queries

Valiant [19] describes a *possibility-oracle*, which takes as input a term t and determines whether or not t has any satisfying instances in common with the unknown formula ϕ_*. The possibility-oracle answers restricted disjointness queries.

Shapiro has also made use of disjointness queries in his work on automatic debugging of Prolog programs [15,16,17]. A brief description will give the reader some feeling for the uses of queries in his system; please refer to the original papers for full details.

5.1 Queries in Shapiro's debugging system

In Shapiro's system the user is assumed to have in mind a model of the correct behavior of his or her program, consisting of a collection of named procedures and, for each procedure, the set of tuples of ground terms on which it is true. For example, the user might be writing a procedure $memb(X,Y)$ that should be true whenever X is a member of the list Y, or a procedure $rev(X,Y)$ that should be true whenever the list Y is the reverse of the list X.

In addition, there is a current program, represented as a set of axioms in Prolog, which may or may not be correct for the intended model. It is assumed that if the program is not correct, this will eventually be discovered and a counter-example provided. (This is in essence an equivalence query.)

The counter-example may be an atom $P(t_1,\ldots,t_k)$ which is provable from the program and not true in the correct model. In this case, the system takes any computation that derives the incorrect atom from the program and, using membership queries, locates an incorrect axiom in the program. The membership queries take the form of asking whether ground atoms are true or false in the intended model. For example, the user might be queried whether $memb(3,[1,2])$ is true or not.

If the counter-example is an atom $P(t_1,\ldots,t_k)$ which is true in the intended model but is such that the program terminates without proving it, then a different diagnosis algorithm is applied to locate an "incomplete procedure", that is, a procedure that requires further axioms. This diagnosis algorithm uses what Shapiro calls *existential queries*, which give the user an atom containing variables and ask if there is any instantiation of the variables that makes the atom true in the intended model. If the user answers *yes*, he or she is then asked to supply instantiations that make the atom true in the intended model.

An example given in [15] in the course of debugging an incorrect insertion sort is the query: $isort([2,1],Z)$?, that is, is there any value of Z for which the insertion sort procedure $isort([2,1],Z)$ is true? The user answers *yes* and is queried for a value of Z. The user supplies $Z=[1,2]$, and the diagnosis algorithm continues.

In the terminology of this paper, this is a disjointness query, testing whether the set of ground instances of the atom $isort([2,1],Z)$ has any elements in common with the ground atoms making up the correct behavior of *isort*, and if so, to supply one. (In Shapiro's system, the user must be prepared to enumerate all of the common instances if there are more than one.)

As thus described, Shapiro's system makes use of equivalence, membership, and disjointness queries. There is an additional type of query that is used in the case that the current program fails to terminate, which is beyond the scope of this

discussion.

5.2 A very small class of context-free languages

In this section we give a fairly artificial example to illustrate the use of disjointness queries in an efficient identification algorithm. Let A be a fixed finite alphabet. Let f be any mapping of A into the integers. We extend f additively to any string $a_1 a_2 \cdots a_n$ of symbols from A:

$$f(a_1 a_2 \cdots a_n) = \sum_{i=1}^{n} f(a_i).$$

Now we define a set of strings, $L(f) \subseteq A^*$, determined by f as follows. The string

$$w = a_1 a_2 \cdots a_n$$

is in $L(f)$ if and only if w is not the empty string, $f(w) = 0$, and for each $i = 1, 2, \ldots, n-1$,

$$f(a_1 a_2 \cdots a_i) \geq 0.$$

Let $\mathcal{C}$ denote the class of all languages $L(f)$ as f ranges over all functions from A to the integers.

As an example, if A consists of a left and a right parenthesis and f assigns 1 to the left parenthesis and -1 to the right parenthesis, then $L(f)$ is just the language of balanced parentheses. It is clear that languages in this class can be recognized by a very simple kind of deterministic push-down automaton with one stack symbol, and are therefore context-free, but the present description suffices for our purposes. Note that $\mathcal{C}$ is very small, in particular, it does not include any finite subset of A^* except the empty set.

Theorem 13 *There is an algorithm that exactly identifies every language in the class $\mathcal{C}$ that uses only disjointness queries and runs in time polynomial in the longest counter-example provided.*

Assume that there is an unknown function f_* mapping A to the integers. For any function f mapping A to the integers, we may ask whether $L(f)$ is disjoint from $L(f_*)$, and if not, a counter-example $w \in L(f) \cap L(f_*)$ will be provided.

For any subset $B \subseteq A$, the language B^+ of all nonempty strings of symbols over the alphabet B is in $\mathcal{C}$ via the function f that maps every element of B to 0 and every element of $A - B$ to 1.

We can easily determine the set Z of symbols $a \in A$ such that $f_*(a) = 0$. For every $a \in A$, we test whether a^+ is disjoint from $L(f_*)$. If so, $f_*(a) \neq 0$; otherwise, $f_*(a) = 0$.

For every pair a_1 and a_2 of distinct symbols from $A - Z$, we test whether the language $\{a_1, a_2\}^+$ is disjoint from $L(f_*)$. If so, then $f_*(a_1)$ and $f_*(a_2)$ are both positive or both negative.

Otherwise, the counter-example will be a nonempty string w in $L(f_*)$ that contains r occurrences of a_1 and s occurrences of a_2. Without loss of generality, assume that the first symbol in w is a_1. Then we know that $f_*(a_1) > 0$ and $f_*(a_2) < 0$, and moreover,

$$rf_*(a_1) = -sf_*(a_2),$$

and neither r nor s is 0.

If we find any function f such that $f(a) = 0$ for all $a \in Z$ and f satisfies all the inequalities and equations above for all pairs a_1 and a_2 from $A - Z$, then $L(f) = L(f_*)$. One way of finding such an f is sketched.

If no equations

$$rf_*(a_1) = -sf_*(a_2)$$

are discovered, then for every a in $A - Z$, $f_*(a)$ has the same sign, so it suffices to take $f(a) = 1$ for all $a \in A - Z$. Otherwise at least one pair a_1, a_2 will be found with $f_*(a_1)$ positive and $f_*(a_2)$ negative, so we will be able to classify every element a of $A - Z$ as to whether $f_*(a)$ is positive or negative, and there will be equations relating every pair consisting of a positive and a negative. Since all the values of r and s are positive, we can express the absolute value of each $f_*(a)$ as a multiple of one of them, say $f_*(a_1)$.

That is, for each $a_i \in A - Z$ we have

$$|f_*(a_i)| = (p_i/q_i)|f_*(a_1)|,$$

for positive integers p_i and q_i. If we let $|f(a_1)|$ be the least common multiple of all the q_i's that appear in these equations and define

$$|f(a_i)| = (p_i/q_i)|f(a_1)|,$$

then combining this with the sign information we have for each $f_*(a_i)$, f is completely defined and $L(f) = L(f_*)$.

Note that the number of queries to determine f is bounded by the square of $|A|$, and the computations involved are bounded by a polynomial in $|A|$ and the lengths of the counter-examples provided, which concludes the proof of Theorem 13

If we also have equivalence queries, the alphabet A need not be known in advance, but can be discovered through equivalence queries.

6 Summary, remarks, and open questions

We have described efficient algorithms and lower bounds for the use of queries in several specific domains. These results are summarized in Figure 1. In the first column are listed the specific domains discussed, including the special cases of context-free language identification described in Section 3.2 (marked with (*)) and in Section 5.2 (marked with (**)). In the second column, labelled *sufficient*, is the smallest set of query-types that has been shown to suffice for efficient exact identification in the specified domain. In the third column, labelled *insufficient*, is the largest set of query-types for which an exponential lower bound on exact identification has been shown for the specified domain. The types of queries are identified by numbers according to the following scheme:

1. Equivalence.
2. Membership.
3. Subset.
4. Superset.
5. Disjointness.

A number superscripted with a minus sign denotes the restricted version of the corresponding query, that is, with no counter-examples returned. For example, 1^- denotes restricted equivalence queries. Recall that in general the use of equivalence queries for exact identification can be replaced with the sampling oracle $EX()$ and *pac*-identification.

A large number of open problems are implicit in Figure 1, for example, is there a domain for which an exponential lower bound on exact identification is provable for the full set, $\{1,2,3,4,5\}$, of query types? Can membership queries be shown to be essential to efficient identification of the regular sets or monotone DNF formulas? Are disjointness queries of any help in identifying the pattern languages?

In addition, there are other domains that seem worth exploring. One, suggested by Phil Laird, is the domain of propositional Horn sentences[10]. A propositional Horn sentence is a propositional formula in CNF with at most one positive literal per clause. Equivalently, it is the conjunction of a set of formulas each of which is either a single positive literal, or an implication of the form

$$A_1 \wedge A_2 \wedge \ldots A_k \rightarrow B,$$

domain	*sufficient*	*insufficient*
singleton languages	4	1,2,3,5
k-CNF formulas	1	1^-,2,3
k-DNF formulas	1	1^-,2,4
monotone DNF formulas	1,2	1^-,2,3
regular languages	1,2	1^-,2,3
context-free languages (*)	1,2	1^-,2,3
pattern languages	4^-	1,2,3
k-term DNF formulas	$1,3^-$	1^-,2,3
k-clause CNF formulas	$1,4^-$	1^-,2,4
context-free languages (**)	5	

Figure 1: Summary of results for specific domains

where each A_i is a positive literal and B is either *false* or a positive literal.

There is a polynomial time algorithm to decide satisfiability of propositional Horn sentences. Moreover, a satisfiable propositional Horn sentence has a unique satisfying instance with a minimum number of *true* variables, and the satisfiability algorithm can be arranged to return this unique instance in the case that the formula is satisfiable. The approach of using equivalence and superset queries to locate all the prime implicands (the dual of prime implicants) of the unknown formula does not appear to be efficient, since Theorem 20 can be modified to show that there are propositional Horn sentences with an exponential number of prime implicands. Thus, despite the computational tractability of this domain, no efficient identification algorithm is known, even if we permit all five types of queries.

Another domain to consider is that of deterministic bottom-up tree acceptors, since many of the theorems characterizing states and distinguishing strings in the domain of deterministic finite acceptors have useful analogs in the case of tree-acceptors [8,12,13].

In any practical setting, the answers to queries of all types are likely to be contaminated with errors. The errors may reflect some consistent ignorance or bias of the informant, or may be generated by some random process. Valiant [18] has found an efficient algorithm for *pac*-identification of k-CNF and k-DNF formulas that is robust in the presence of small but possibly malicious errors in the $EX()$ oracle. Angluin and Laird [3] have found an efficient algorithm for *pac*-identification of k-CNF and k-DNF formulas that is robust in the presence of

larger but randomly generated errors in the $EX()$ oracle. These represent just the beginning of an understanding of the effect of errors on identification and learning.

Another area of open questions is how the logarithmic "majority vote" strategy described in Section 2.3 can actually be used or approximated in practice. This seems to lead to difficult questions of how to sample "fairly" from the set of hypotheses that are compatible with the replies to previous queries.

The existence of domains for which there are lower bounds for identification WITHOUT restricting the queries to elements of the hypothesis space (but assuming the computational intractability of relevant number-theoretic problems) appears to follow fairly directly from the work of Blum, Goldreich, Goldwasser, Micali, and others on the construction of cryptographically secure pseudo-random bit generators and functions [6,9].

The classification of queries given in this report is appropriate to the problem of identifying a single unknown subset of a given universe, and can be readily extended to the case of identifying a collection of named subsets, as in Shapiro's work. In practical cases one is sometimes interested in identifying functions rather than sets, and the classification of queries for this situation deserves to be rethought rather than just interpreted as a special case.

Valiant [19] makes use of two additional types of queries specific to DNF formulas: *relevant possibility* and *relevant accompaniment.* Shapiro [15,16,17] makes use of one additional type of query to help diagnose nonterminating Prolog programs. Still other sources of information will prove to be relevant for other specific domains. The classification of types of queries in this report should in no sense be thought of as exhaustive or canonical, but it does permit comparison of existing methods and poses some new and interesting questions about learning and identification problems.

7 Appendix: Bounds on the number of prime implicants of a k-term DNF formula

In this section we prove Lemma 9, that is, we show that a formula in DNF with k terms and n variables has fewer than 3^k prime implicants. We also prove an exponential lower bound on the number of prime implicants. For the proofs it is more convenient to consider the dual problem.

7.1 An upper bound

Theorem 14 *If ϕ is a CNF formula with n variables and k clauses, then the number of prime implicands of ϕ is less than 3^k.*

A *prime implicand* of a formula ϕ is a clause c such that c is not a tautology, ϕ implies c, and for every clause c' obtained by deleting one literal from c, ϕ does not imply c'. The number of prime implicands of ϕ is equal to the number of prime implicants of $\neg\phi$, so Lemma 9 is an immediate corollary of Theorem 14.

The proof of Theorem 14 is based on the fact that a prime implicand of a CNF formula ϕ can be obtained by a resolution proof from the clauses of ϕ, and consists essentially of a close look at any such resolution proof. An introduction to proof by resolution may be found in [14]. The proof of Theorem 14 proceeds by a sequence of lemmas.

The following lemma says that every prime implicand of ϕ can be obtained by a resolution proof from the clauses of ϕ.

Lemma 15 *If $\phi = c_1 \wedge c_2 \wedge \ldots \wedge c_k$ is a formula in CNF and c is a prime implicand of ϕ, then there is a resolution proof of c from the clauses $c_1, c_2, \ldots, c_k$.*

Suppose $c = (L_1 \vee L_2 \vee \ldots \vee L_m)$, where the L_i are the literals appearing in c. Since ϕ implies c, the formula

$$c_1 \wedge c_2 \wedge \ldots c_k \wedge \neg L_1 \wedge \neg L_2 \wedge \ldots \neg L_m$$

is unsatisfiable, so there is a resolution proof of the empty clause from the clauses $c_1, \ldots, c_k$ and $\neg L_1, \ldots, \neg L_m$. If we modify the resolution proof by omitting the steps involving the input clauses $\neg L_1, \ldots, \neg L_m$, then the output clause, instead of being empty, will be a clause c' consisting of a subset of the literals $L_1, \ldots, L_m$. Thus ϕ implies c' and c' is a subset of c. Since c is a prime implicand of ϕ, this means $c = c'$, so there is a resolution proof of c from $c_1, \ldots, c_k$, as claimed.

The next lemma permits us to ignore clauses c_j such that $(c \vee c_j)$ is a tautology.

Lemma 16 *If $\phi = c_1 \wedge c_2 \wedge \ldots \wedge c_k$ is a formula in CNF and c is a prime implicand of ϕ, and*

$$I = \{i : (c_i \vee c) \text{ is not a tautology}\},$$

then c is a prime implicand of the reduced formula ϕ', where

$$\phi' = \bigwedge_{i \in I} c_i.$$

Since ϕ implies ϕ', it suffices to show that ϕ' implies c. Assume to the contrary that ϕ' does not imply c, i.e., there exists a truth-value assignment a that satisfies ϕ' but not c. Since ϕ implies c, a must not satisfy ϕ, so there exists an index j not in I such that a does not satisfy c_j. But $(c_j \vee c)$ is by hypothesis a tautology, so

every assignment, including a, must satisfy c_j or c or both, a contradiction. This proves Lemma 16.

If ϕ is a formula in CNF, we partition the literals appearing in ϕ into two sets, depending on whether the complement of the literal also appears in ϕ. Let $U(\phi)$ contain all those literals L such that L appears in ϕ and the complement of L does not appear in ϕ. Let $V(\phi)$ contain all those literals L such that both L and its complement appear in ϕ.

The following lemma says that the U-portions of clauses move as "unsplittable blocks" in a resolution proof.

Lemma 17 *If $\phi = c_1 \wedge c_2 \wedge \ldots \wedge c_k$ is a formula in CNF and c is any clause obtained by a resolution proof from $c_1, \ldots, c_k$, then there exists a set I contained in $\{1, 2, \ldots, k\}$ such that*

$$c \cap U(\phi) = \bigcup_{i \in I} c_i \cap U(\phi).$$

The proof is by induction on the resolution proof. If the proof has no resolution steps, then $c = c_i$ for some i and the result follows. Assume that c is obtained by resolving the clauses c' and c'', and that for some sets I' and I'',

$$c' \cap U(\phi) = \bigcup_{i \in I'} c_i \cap U(\phi),$$

and

$$c'' \cap U(\phi) = \bigcup_{i \in I''} c_i \cap U(\phi).$$

Since elements of $U(\phi)$ do not appear complemented anywhere in ϕ, they cannot be resolved upon, so c simply inherits them from c' and c'', that is,

$$c \cap U(\phi) = \bigcup_{i \in I' \cup I''} c_i \cap U(\phi).$$

This concludes the proof of Lemma 17.

The next lemma characterizes prime implicands in terms of the U-portions of the original clauses.

Lemma 18 *If $\phi = c_1 \wedge c_2 \wedge \ldots \wedge c_k$ is a formula in CNF and c is a prime implicand of ϕ such that no $(c \vee c_i)$ is a tautology, then for some subset I of $\{1, 2, \ldots, k\}$,*

$$c = \bigcup_{i \in I} c_i \cap U(\phi).$$

By Lemma 15, c can be obtained by a resolution proof from the clauses of ϕ. Thus, c contains only literals from $U(\phi) \cup V(\phi)$. Note that if c contains any literal from $V(\phi)$, say L, then there is some clause c_i in which L appears complemented, so $(c \vee c_i)$ is a tautology, contrary to our hypothesis on c. Thus c is a subset of $U(\phi)$. Applying Lemma 17, there exists some subset I of $\{1, 2, \ldots, k\}$ such that

$$c \cap U(\phi) = \bigcup_{i \in I} c_i \cap U(\phi).$$

Since c is a subset of $U(\phi)$, $c = c \cap U(\phi)$, and Lemma 18 is proved.

Thus, every prime implicand of ϕ of this kind consists of a selection of U-portions from the clauses of ϕ. It follows immediately that there are no more than 2^k such prime implicands. We state this formally.

Lemma 19 *If $\phi = c_1 \wedge c_2 \wedge \ldots \wedge c_k$ is a formula in CNF then there are at most 2^k prime implicands c of ϕ with the property that for no i is $(c \vee c_i)$ a tautology.*

The proof of Theorem 14 may now be concluded. Suppose $\phi = c_1 \wedge c_2 \wedge \ldots \wedge c_k$ is a formula in CNF. If c is any prime implicand of ϕ, then there is a unique set $I(c)$ consisting of all those indices i such that $(c \vee c_i)$ is not a tautology. Then by Lemma 16, c is also a prime implicand of the reduced formula

$$\phi' = \bigwedge_{i \in I(c)} c_i.$$

By Lemma 19, there are at most $2^{|I(c)|}$ prime implicands c of ϕ' with the property that $(c \vee c')$ is not a tautology for any c' in ϕ'.

To bound the number of prime implicands of ϕ, it suffices to consider every formula ϕ' obtained by taking a nonempty subset of the clauses of ϕ and summing up the bounds on the number of prime implicands c of ϕ' such that $(c \vee c')$ is not a tautology for any c' in ϕ'. (The empty subset gives us a formula equivalent to *true*, which has no prime implicands, since a prime implicand is required not to be a tautology.) This gives us a bound of

$$\sum_{m=1}^{k} \binom{m}{k} 2^m < 3^k,$$

which concludes the proof of Theorem 14.

7.2 A lower bound

We also prove the following exponential lower bound on the number of prime implicands of a CNF formula with k clauses.

Theorem 20 *For every positive integer k there is a formula ϕ_k in CNF with $2k+1$ variables and $4k+1$ clauses that has at least 2^k prime implicands.*

The formula ϕ_k contains variables $x_1, \ldots, x_k$ and $p_0, p_1, ..., p_k$. It may be thought of as computing the parity of the true x_i's. That is, p_i should be true if and only if an odd number of the variables $x_1, x_2, \ldots, x_i$ are true.

The clauses of ϕ_k are $(\neg p_0)$ and for each $i = 1, 2, \ldots, k$ the four clauses:

$$(p_{i-1} \wedge x_i \rightarrow \neg p_i),$$

$$(\neg p_{i-1} \wedge x_i \rightarrow p_i),$$

$$(p_{i-1} \wedge \neg x_i \rightarrow p_i),$$

$$(\neg p_{i-1} \wedge \neg x_i \rightarrow \neg p_i).$$

Let c be any clause of the form

$$(Y_1 \wedge Y_2 \wedge \ldots \wedge Y_k) \rightarrow P,$$

where each Y_i is either x_i or $\neg x_i$, and P is p_k if an odd number of unnegated x_i's appear and $\neg p_k$ otherwise. Then we show that c is a prime implicand of ϕ_k. Since there are 2^k such clauses, the result follows.

It is not difficult to see that ϕ_k implies c. To see that c is a prime implicand, first note that it is not a tautology. If c' is derived from c by deleting Y_i, consider the assignment that makes Y_i false, all the other Y_j's in c true, and P false. This assignment makes ϕ_k true but c' false, so ϕ_k does not imply c'. If c' is derived from c by deleting P, consider the assignment that makes all the Y_i's true and P true. This assignment makes ϕ_k true, but makes c' false (since c' is the disjunction of the complements of all the Y_i's), so ϕ_k does not imply c'. Hence no clause obtained from c by deleting one literal is implied by ϕ_k, so c is a prime implicand. This concludes the proof of Theorem 20.

Note that we can modify the formula ϕ_k to be in Horn form by introducing variables $y_1, y_2, \ldots, y_k$ and $q_0, q_1, \ldots, q_k$ and substituting them for the negations of the x_i's and p_i's respectively. Thus, an exponential lower bound holds also for the number of prime implicands of a propositional Horn sentence.

8 Acknowledgements

I have enjoyed conversations about this material with Manuel Blum, Bill Gasarch, David Haussler, Phil Laird, Lenny Pitt, Udi Shapiro, and Carl Smith. The financial support of the National Science Foundation under grant number DCR-8404226 is gratefully acknowledged.

References

[1] D. Angluin. Finding patterns common to a set of strings. *J. Comp. Sys. Sci.*, 21:46–62, 1980.

[2] D. Angluin. *Learning regular sets from queries and counter-examples.* Technical Report, Yale University Computer Science Dept. TR-464, 1986.

[3] D. Angluin and P. D. Laird. *Identifying k-CNF formulas from noisy examples.* Technical Report, Yale University Computer Science Dept. TR-478, 1986.

[4] D. Angluin and C. Smith. Inductive inference: theory and methods. *Comput. Surveys*, 15:237–269, 1983.

[5] J. M. Barzdin and R. V. Freivalds. On the prediction of general recursive functions. *Sov. Math. Dokl.*, 13:1224–1228, 1972.

[6] M. Blum and S. Micali. How to generate cryptographically strong sequences of pseudo-random bits. *SIAM J. Comput.*, 13:850–864, 1984.

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

[8] L. Fass. *Inference of skeletal automata.* Technical Report, Georgetown University Computer Science Dept. TR-2, 1984.

[9] O. Goldreich, S. Goldwasser, and S. Micali. How to construct random functions. In *Proc. 25th Annual Symposium on Foundations of Computer Science*, pages 464–479, IEEE, 1984.

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

[11] B. Knobe and K. Knobe. A method for inferring context-free grammars. *Inform. Contr.*, 31:129–146, 1976.

[12] B. Levine. Derivatives of tree sets with applications to grammatical inference. *IEEE Trans. on Pattern Analysis and Machine Intelligence*, PAMI-3:285–293, 1981.

[13] B. Levine. The use of tree derivatives and a sample support parameter for inferring tree systems. *IEEE Trans. on Pattern Analysis and Machine Intelligence*, PAMI-4:25–34, 1982.

[14] H. R. Lewis and C. H. Papadimitriou. *Elements of the Theory of Computation.* Prentice-Hall, Inc., 1981.

[15] E. Shapiro. *Algorithmic program debugging.* PhD thesis, Yale University Computer Science Dept., 1982. Published by MIT Press, 1983.

[16] E. Shapiro. Algorithmic program diagnosis. In *Proc. Ninth ACM Symposium on Principles of Programming Languages*, pages 299–308, ACM, 1982.

[17] E. Shapiro. A general incremental algorithm that infers theories from facts. In *Seventh IJCAI*, pages 446–451, IJCAI, 1981.

[18] L. G. Valiant. Learning disjunctions of conjunctions. In *Proceedings of IJCAI*, pages 560–566, IJCAI, 1985.

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