Machine Learning, 28, 211–255 (1997)
© 1997 Kluwer Academic Publishers. Manufactured in The Netherlands.

Malicious Omissions and Errors in Answers to Membership Queries

DANA ANGLUIN angluin@cs.yale.edu
MĀRTIŅŠ KRIĶIS krikis@cs.yale.edu
Department of Computer Science, Yale University, P.O. Box 208285, New Haven, CT 06520

ROBERT H. SLOAN sloan@eecs.uic.edu
Dept. of Electrical Eng. and Computer Science, 851 S. Morgan St. Rm 1120, University of Illinois at Chicago, Chicago, IL 60607

GYÖRGY TURÁN u11557@uicvm.uic.edu
Dept. of Math., Stat., and Comp. Sci., 851 S. Morgan St. Rm 322, University of Illinois at Chicago, Chicago, IL 60607, Automata Theory Research Group Hungarian Academy of Sciences, Szeged

Editor: David Haussler

Abstract. We consider two issues in polynomial-time exact learning of concepts using membership and equivalence queries: (1) errors or omissions in answers to membership queries, and (2) learning finite variants of concepts drawn from a learnable class.

To study (1), we introduce two new kinds of membership queries: limited membership queries and malicious membership queries. Each is allowed to give incorrect responses on a maliciously chosen set of strings in the domain. Instead of answering correctly about a string, a limited membership query may give a special "I don't know" answer, while a malicious membership query may give the wrong answer. A new parameter L is used to bound the length of an encoding of the set of strings that receive such incorrect answers. Equivalence queries are answered correctly, and learning algorithms are allowed time polynomial in the usual parameters and L. Any class of concepts learnable in polynomial time using equivalence and malicious membership queries is learnable in polynomial time using equivalence and limited membership queries; the converse is an open problem. For the classes of monotone monomials and monotone k-term DNF formulas, we present polynomial-time learning algorithms using limited membership queries alone. We present polynomial-time learning algorithms for the class of monotone DNF formulas using equivalence and limited membership queries, and using equivalence and malicious membership queries.

To study (2), we consider classes of concepts that are polynomially closed under finite exceptions and a natural operation to add exception tables to a class of concepts. Applying this operation, we obtain the class of monotone DNF formulas with finite exceptions. We give a polynomial-time algorithm to learn the class of monotone DNF formulas with finite exceptions using equivalence and membership queries. We also give a general transformation showing that any class of concepts that is polynomially closed under finite exceptions and is learnable in polynomial time using standard membership and equivalence queries is also polynomial-time learnable using malicious membership and equivalence queries. Corollaries include the polynomial-time learnability of the following classes using malicious membership and equivalence queries: deterministic finite acceptors, boolean decision trees, and monotone DNF formulas with finite exceptions.

Keywords: Concept learning, queries, errors

1. Introduction

There is an impressive and growing number of polynomial-time algorithms, many of them quite beautiful and ingenious, to learn various interesting classes of concepts using equiva-

lence and membership queries. To apply such algorithms in practice, researchers need to overcome a number of problems.

One significant issue is the problem of omissions and errors in answers to queries. Previous learning algorithms in the equivalence and membership query model are guaranteed to perform well assuming that queries are answered correctly, but there is often no guarantee that the performance of the algorithm will "degrade gracefully" if that assumption is not exactly satisfied.

Lang and Baum (1992) report that this problem derailed their attempt to apply Baum's algorithm for learning neural nets from examples and membership queries (Baum, 1991) to the problem of recognizing hand-written digits. The attempt failed because the membership questions posed by the algorithm were too difficult for people to answer reliably. The algorithm typically asked membership queries on, say, a random-looking blur midway between a "5" and a "7," and the humans being queried gave very inconsistent responses. Studies in cognitive psychology indicate that this is the norm; people are typically quite inconsistent in deciding where the precise boundary of a concept lies. (See, e.g., Anderson (1980).)

1.1. Omissions and Limited Membership Queries

This motivated us to introduce the limited membership query. A limited membership query may be answered either correctly, or with an omission, that is, a special value signifying "I don't know." The answers are persistent; that is, repeated queries about the same example are given the same answer. The choice of the set of strings on which to give answers of "I don't know" is assumed to be made by a malicious adversary. We introduce a new parameter L to quantify the "amount" of omission—it is a bound on the table-size of the set of strings on which the adversary answers "I don't know." (The table-size of a set of strings is the number of strings in the set plus the sum of their lengths.) A polynomial-time learning algorithm is permitted time polynomial in the usual parameters and L.

For this model, we define a hypothesis to be "nonstrictly" correct if it agrees with the target concept on all examples except possibly ones for which a limited membership query was answered "I don't know." Thus, domain elements answered with "I don't know" are allowed to be classified arbitrarily by the final hypothesis of a learning algorithm. This corresponds to the intuition that since a person could not be sure of the classification of a "blur between 5 and 7", it does not matter how the final hypothesis classifies it.

We give a polynomial-time learning algorithm using just limited membership queries for the class of monotone monomials and a lower bound on the query complexity of this problem. We also give a polynomial-time learning algorithm in this model for the class of k-term monotone DNF formulas.

We also consider combining limited membership queries with equivalence queries. We assume that the answers to equivalence queries remain correct, that is, any counterexample given is truly a counterexample to the hypothesis of the learning algorithm. In the nonstrict model, the equivalence query is answered "yes" if the hypothesis is nonstrictly correct; otherwise a counterexample must be returned from among examples not answered with "I don't know." In the strict model, equivalence queries remain as usual, that is, an equivalence query is answered with "yes" if the hypothesis is exactly equivalent to the target concept;

otherwise an arbitrary counterexample is returned (including possibly an example previously answered with “I don’t know.”)

We show that the same classes of concepts can be learned in polynomial time using limited membership queries and equivalence queries in the nonstrict and strict models. We give a polynomial-time algorithm to learn monotone DNF formulas using limited membership queries and equivalence queries in the nonstrict model.

1.2. Malicious Membership Queries and Errors

In the case of limited membership queries, the answers that are not omissions are guaranteed to be correct. We also consider the situation in which the answers to membership queries may be wrong. In a malicious membership query, the answer given may be correct, or it may be an error. As in the case of limited membership queries, we use the parameter L to bound the table-size of the set of strings whose malicious membership queries are answered erroneously, the choice of that set of strings is assumed to be made by a malicious adversary, and the answers to queries are persistent. We assume that equivalence queries remain correct, which corresponds to the strict model introduced above. That is, the final hypothesis of the learning algorithm must be exactly equivalent to the target concept.

We give a polynomial-time algorithm to learn monotone DNF formulas using malicious membership queries and equivalence queries. It is not difficult to see that any concept class learnable in polynomial time using malicious membership queries and equivalence queries is learnable in polynomial time using limited membership queries and equivalence queries, but the converse direction is an interesting open question. We exhibit a class of concepts for which the query complexity for equivalence and malicious membership queries is not bounded by any polynomial in the query complexity for equivalence and limited membership queries, but both complexities are exponential in the number of strings that receive incorrect answers.

1.3. Finite Exceptions

A related issue is the assumption that the target concept is drawn from a particular class of concepts, for example, monotone DNF formulas. Even if the target concept is “nearly” a monotone DNF formula, there is typically no guarantee that the learning algorithm will do anything reasonable. We approach this question by considering finite variants of the concepts in a given class, using the table-size of the set of exceptions as a measure of “how different” the target concept is from one in the specified class.

We define what it means for a concept class to be polynomially closed under finite exceptions. Some concept classes, for example, DFA’s and decision trees, are polynomially closed under finite exceptions, while others, like monotone DNF formulas, are not. For the latter, we define a natural operation of adding exception tables to the concept class to make it polynomially closed under exceptions. We give a polynomial-time learning algorithm for the resulting class of monotone DNF formulas with finite exceptions, using equivalence queries and standard membership queries.

We then give a general transformation that shows that any class of concepts that is polynomially closed under exceptions and polynomial-time learnable using equivalence que-

ries and standard membership queries is also polynomial-time learnable using equivalence queries and malicious membership queries. Corollaries include polynomial-time learning algorithms using equivalence queries and malicious membership queries for the concept classes of DFA's, decision trees, and monotone DNF formulas with finite exceptions.

The notion of a finite variant of a concept, that is, a concept with a finite set of exceptions, is a unifying theme between the models of learning with equivalence and malicious membership queries and of learning a concept class with finite exceptions. Our model of errors in membership queries can be viewed as combining an equivalence oracle for the target concept and a membership oracle for a finite variant of the target concept. In the case of learning a concept class with finite exceptions, the equivalence and membership oracles present the same finite variant of a concept in the base class. In both cases, the goal is to identify exactly the concept presented by the equivalence oracle.

1.4. Related Work

There is a considerable body of literature on errors in examples in the PAC model, starting with the first error-tolerant algorithm in the PAC model, given by Valiant (1985). In this case the goal is PAC-identification of the target concept, despite the corruption of the examples by one or another kind of error, for example, random or malicious misclassification errors, random or malicious attribute errors, or malicious errors (in which both attributes and classification may be arbitrarily changed).

There has been not as much work on omissions and errors in learning models in which membership queries are available, and the issues are not as well understood. One relevant distinction is whether the omissions of errors in answers to membership queries are persistent or not. They are *persistent* if repeated queries to the same domain element always return the same answer. In general, the case of persistent omissions or errors is more difficult, since non-persistent omissions or errors can yield extra information, and can always be made persistent simply by caching and using the first answer for each domain point queried.

1.5. Non-persistent Errors in Queries

Sakakibara defines one model of non-persistent errors, in which each answer to a query may be wrong with some probability, and repeated queries constitute independent events (Sakakibara, 1991). He gives a general technique of repeating each query sufficiently often to establish the correct answer with high probability. This yields a uniform transformation of existing query algorithms. The method also works for both of Bultman's models (Bultman, 1991). This could be a reasonable model of a situation in which the answers to queries were being transmitted through a medium subject to random independent errors; then the technique of repeating the query is eminently sensible.

A related model is considered by Dean et al. (1995) for the case of a robot learning a finite-state map of its environment using faulty sensors and reliable effectors. This model assumes that observation errors are independent as long as there is a nonempty action sequence separating the observations. This means that there is no simple way to "repeat the same query", since a nonempty action sequence may take the robot to another state, and no reset operation is available. A polynomial-time learning algorithm is given for the

situation in which the environment has a known distinguishing sequence. It achieves exact identification with high probability.

1.6. Persistent Errors in Membership Queries

The method of "repeating the query" is insufficient for the more difficult case of persistent omissions or errors in membership queries. In this case, we must exploit the error-correcting properties of groups of "related" queries. In an explicit and very interesting application of the ideas of error-correcting algorithms, Ron and Rubinfeld use the criterion of PAC-identification with respect to the uniform distribution, and give a polynomial-time randomized algorithm using membership queries to learn DFA's with high rates of random persistent errors in the answers to the membership queries (Ron & Rubinfeld, 1995).

Algorithms that use membership queries to estimate probabilities (in the spirit of the statistical queries defined by Kearns (1993)) are generally not too sensitive to small rates of random persistent errors in the answers to queries. For example, Goldman, Kearns, and Schapire give polynomial-time algorithms for exactly learning read-once majority formulas and read-once positive NAND formulas of depth $O(\log n)$ with high probability using membership queries with high rates of persistent random noise or modest rates of persistent malicious noise (Goldman, Kearns & Schapire, 1993). As another example, Kushilevitz and Mansour's algorithm that uses membership queries and exactly learns logarithmic-depth decision trees with high probability in polynomial time seems likely to be robust under nontrivial rates of persistent random noise in the answers to queries (Kushilevitz & Mansour, 1993).

However, learning algorithms for other classes of concepts using equivalence and membership queries may depend more strongly on the correctness of the answers to individual queries; in these cases, there is no guarantee of a learning algorithm for the class that can tolerate omissions or errors in the answers to membership queries.

One model that addressed these questions was introduced by Angluin and Slonim: equivalence queries are assumed to be answered correctly, while membership queries are either answered correctly or with "I don't know" and the answers are persistent. The "I don't know" answers are determined by independent coin flips the first time each query is made (Angluin & Slonim, 1994). They give a polynomial-time algorithm to learn monotone DNF formulas with high probability in this setting. They also show that a variant of this algorithm can deal with one-sided errors, assuming that no negative point is classified as positive. Goldman and Mathias also consider this model (Goldman & Mathias, 1992). Our current models and results differ in that the omissions and errors are chosen by a malicious adversary instead of a random process, and the rate of incorrect answers that can be tolerated is consequently much lower.

Frazier et al. (1994) have introduced a model of omissions in answers to membership queries, called learning from a consistently ignorant teacher. The basic idea is to require that if the teacher gives answers to certain queries that would imply a particular answer to another query, the teacher cannot answer the latter query with "I don't know." For example, in the domain of monotone DNF formulas, if the teacher classifies a particular point as positive, then the teacher cannot answer "I don't know" about any of the ancestors of the point. The goal of the learner is to learn exactly the ternary classification of points into

positive, negative, and “I don’t know” that is presented by the teacher. Such a ternary classification may be represented by the *agreement* of a set of binary-valued concepts; the agreement classifies a point as positive (respectively, negative) if all the concepts in the set classify it as positive (respectively, negative), otherwise, the agreement classifies the point as “I don’t know.” Efficient learning algorithms are given in this model for monomials with at least one positive example, concepts represented as the agreement of a constant number of monotone DNF formulas, k-term DNF formulas, DFA’s, or decision trees, and concepts represented by an agreement of boxes with samplable intersection. Compared to our model, this model has a different measure of the representational complexity of a concept with omissions, which allows a much higher rate of omissions to be compactly represented. It also differs in requiring the learner to reproduce exactly the “I don’t know” labels of the teacher, whereas in our (nonstrict) model of omissions such examples can be classified arbitrarily.

2. Preliminaries

2.1. Concepts and Concept Classes

Our definitions for concepts and concept classes are a bit non-standard. We have explicitly introduced the domains of concepts in order to try to unify the treatment of fixed-length and variable-length domains. We take Σ and Γ to be two finite alphabets. Examples are represented by finite strings over Σ and concepts are represented by finite strings over Γ.

A *concept* consists of a pair (X, f), where $X \subseteq \Sigma^*$, and f maps X to $\{0, 1\}$. The set X is the *domain* of the concept. The *positive examples* of (X, f) are those $w \in X$ such that $f(w) = 1$, and the *negative examples* of (X, f) are those $w \in X$ such that $f(w) = 0$. Note that strings not in the domain of the concept are neither positive nor negative examples of it.

A *concept class* is a triple (R, Dom, μ), where R is a subset of Γ^*, *Dom* is a map from R to subsets of Σ^*, and for each $r \in R$, $\mu(r)$ is a function from $\mathit{Dom}(r)$ to $\{0, 1\}$. R is the set of legal representations of concepts. For each $r \in R$, the *concept represented by* r is $(\mathit{Dom}(r), \mu(r))$.

A concept (X, f) is *represented by* a concept class (R, Dom, μ) if and only if for some $r \in R$, (X, f) is the concept represented by r. The *size* of a concept (X, f) represented by (R, Dom, μ) is defined to be the length of the shortest string $r \in R$ such that r represents (X, f). The size of (X, f) is denoted by $|(X, f)|$; note that it depends on the concept class chosen.

The concept classes we consider in this paper are boolean formulas and syntactically restricted subclasses of them, boolean decision trees, and DFA’s. The representations are more or less standard, except each concept representation specifies the relevant domain. For DFA’s, the domain of every concept is the set Σ^* itself. For boolean formulas and decision trees, we assume that $\Sigma = \{0, 1\}$, and each concept representation specifies a domain of the form $\{0, 1\}^n$.

For each finite set S of strings from Σ^*, we define its *table-size*, denoted $||S||$, as the sum of the lengths of the strings in S and the number of strings in S. Note that $||S|| = 0$ if and

only if $S = \emptyset$. The table-size of a set of strings is related in a straightforward way to an encoding of a list of the strings; see Section 5.

2.2. Queries

For a learning problem we assume that there is an unknown target concept r drawn from a known concept class (R, Dom, μ). Information about the target concept is available to the learning algorithm as the answers to two types of queries: equivalence queries and membership queries.

In an equivalence query, the algorithm gives as input a concept $r' \in R$ with the same domain as the target, and the answer depends on whether $\mu(r) = \mu(r')$. If so, the answer is "yes", and the learning algorithm has succeeded in its goal of exact identification of the target concept. Otherwise, the answer is a *counterexample*, any string $w \in \mathit{Dom}(r)$ on which the functions $\mu(r)$ and $\mu(r')$ differ. We denote an equivalence query on a hypothesis h by $\mathrm{EQ}(h)$.

The *label* for a counterexample $v = \mathrm{EQ}(r')$ is the value of $\mu(r)$ on v, giving its classification by the target concept. Since the hypothesized concept r' and the target concept r differ on the classification of the counterexample v, the label of v is also the complement of the value of $\mu(r')$ on v. *Positive counterexamples* are those with label 1 and *negative counterexamples* are those with label 0.

In a membership query, the learning algorithm gives as input a string $w \in \mathit{Dom}(r)$, and the answer is either 0, 1, or $\perp$. If the answer is equal to the value of $\mu(r)$ on w, then the answer is *correct*. If the answer is equal to $\perp$, we say that the answer is an *omission* or a *"Don't know"*. If the answer is 0 or 1 but not equal to the value of $\mu(r)$ on w, then the answer is an *error* or a *lie*.

In a *standard membership query*, denoted MQ, all the answers are required to be correct. In a *limited membership query*, denoted LMQ, each answer is required to be correct or an omission. In a *malicious membership query*, denoted MMQ, each answer is required to be correct or an error (no omissions). Note that an answer of 0 or 1 to a limited membership query is always correct, but this is not true for answers to malicious membership queries.

The answers to malicious and limited membership queries are also restricted as follows.

1. They are *persistent*; that is, different membership queries with the same input string w receive the same answer. Note that non-persistent queries may reveal some information; in case two different queries to the same string receive different answers, the learning algorithm knows that there has been an error on this string, though this will not in general determine the correct classification of the string. Every algorithm designed to work with persistent queries can be made to work with non-persistent ones by caching the queries and always using the first answer for subsequent queries of the same string.

2. In addition, we bound the quantity of errors (or omissions) permitted in answers to malicious (resp., limited) membership queries. One natural quantity to bound would be the number of different strings whose membership queries can be answered incorrectly, and this works well in fixed-length domains. However, in variable-length domains, we wish to account for the lengths of the strings as well as their number.

Therefore, in general the algorithm is given a bound L on the table-size, $||S||$, of the set S of strings whose malicious (resp., limited) membership queries are answered erroneously (resp., with a $\perp$) during a single run. In the case of a fixed-length domain, $\{0,1\}^n$, we may instead give a bound ℓ on the number of different strings whose MMQ's (resp., LMQ's) are answered incorrectly (resp., with a $\perp$). Note that $L = \ell(n+1)$ is a bound on the table-size in this case.

Note that when $L = 0$ or $\ell = 0$ there can be no errors or omissions in the answers to MMQ's (or LMQ's) and we have the usual model of standard membership queries as a special case.

We assume that an on-line adversary controls the choice of counterexamples in answers to equivalence queries and the choice of which elements of the domain will be answered with errors (or $\perp$'s) in malicious (or limited) membership queries. When the learning algorithms we consider are deterministic, the adversary may be viewed as choosing in advance the set of strings for which it will give incorrect answers to membership queries, as well as all the counterexamples it will give to equivalence queries.

In this paper we consider models in which the learning algorithm has access to the following combinations of queries:

1. membership and equivalence queries,
2. limited membership queries alone,
3. limited membership queries and equivalence queries, and
4. malicious membership queries and equivalence queries.

Model (1) is just the usual model of a minimally adequate teacher (Angluin, 1987). In model (2), the learning algorithm need only achieve *nonstrict identification*. In other words, the concept output by the learning algorithm must agree with the target concept on all points not answered $\perp$ by the LMQ, but it may differ on points answered $\perp$. This corresponds to our view that $\perp$ points form the borderline of the concept and that the classification of them is irrelevant or meaningless.

For model (3) we consider both *nonstrict* and *strict* variants. In the nonstrict variant, equivalence queries are modified so that if the queried concept and the target concept differ only on points classified as $\perp$ by the LMQ, then the reply is "yes". Otherwise, a counterexample must be given from the set of points not classified as $\perp$ by the LMQ. In this case, as in model (2), only nonstrict identification is required. In the strict variant of model (3), as well as in model (4), equivalence queries are not modified, and the learning algorithm is required to achieve the usual kind of exact identification, that is, the output concept must agree with the target concept on every point in their common domain.

We extend the usual notion of polynomial-time learning to models (2-4) by allowing the polynomial bound on the running time to depend on three parameters, that is, $p(s, n, L)$. Here s is the usual parameter bounding the length of the representation of the target concept, n is the usual parameter bounding the length of the longest counterexample seen so far, and L is a new parameter, bounding the table-size of the set of strings on which LMQ answers $\perp$ or MMQ gives incorrect answers.

The definitions are extended in the usual way to cover randomized learning algorithms and their expected running times, and also extended equivalence queries, in which the inputs to equivalence queries and the final result of the algorithm are allowed to come from a concept class different from (usually larger than) the concept class from which the target is drawn.

It is straightforward to transform any algorithm that uses malicious membership queries into one that uses limited membership queries. Every $\perp$ answer can be replaced by a 0 or a 1 arbitrarily and given to the learner. Therefore learning with malicious membership queries is at least as hard as learning with limited membership queries in the strict model. The same applies to learning from equivalence and malicious membership queries and learning from equivalence and limited membership queries in the strict model. Furthermore, in Subsection 6.3 we show that the strict and the nonstrict models of learning from equivalence and limited membership queries are in fact polynomial-time equivalent. Note that the most general kind of membership queries is one in which both wrong and $\perp$ answers are possible, but such queries are not harder than the malicious ones and therefore we consider only the latter.

2.3. *Monotone DNF Formulas*

We assume a set of propositional variables V and denote its elements by $x_1, x_2, \ldots, x_n$, where n is the cardinality of V. A monotone DNF formula over V is a DNF formula over V where no literal is negated. The domain of such a formula is $\{0,1\}^n$. For example, for $n = 20$,

$$x_1x_4 \vee x_2x_{17}x_3 \vee x_9x_5x_{12}x_3 \vee x_8$$

is a monotone DNF formula (with domain $\{0,1\}^{20}$). We assume that the target formula h^* has been minimized, that is, it contains no redundant terms. (Incidentally, there is an efficient algorithm to minimize the number of terms of a monotone DNF formula.) For a monotone DNF formula f, let $\#(f)$ denote the number of terms in f. In the above example, $\#(f) = 4$.

We view the domain $\{0,1\}^n$ of monotone DNF formulas (with or without exceptions) as a lattice, with componentwise "or" and "and" as the lattice operations. The top element is the vector of all 1's, and the bottom element is the vector of all 0's. The elements are partially ordered by $\leq$, where $v \leq w$ if and only if $v[i] \leq w[i]$ for all $1 \leq i \leq n$. Often we refer to the examples as points of the hypercube $\{0,1\}^n$. For a point v, all points w such that $w \leq v$ are called the *descendants* of v. Those descendants that can be obtained by changing exactly one coordinate of v from a 1 to a 0 are called the *children* of v. The *ancestors* and the *parents* are defined similarly. Note that v is both a descendant and ancestor of itself.

For convenience, we use a representation of monotone DNF formulas in which each term is represented by the minimum vector, in the ordering $\leq$, that satisfies the term. Thus, vector 10011 (where $n = 5$) denotes the term $x_1x_4x_5$. In this representation, if h is a monotone DNF formula and v is a vector in the domain, v satisfies h if and only if for some term t of h, $t \leq v$. That is, a monotone DNF formula is satisfied only by the ancestors of its terms. In the other direction, we say that term t *covers* point v if and only if v satisfies t. For the sake of simplicity we often use in our algorithms something called "the empty

DNF formula". This is the formula with no terms, which is not satisfied by any point, and is therefore the identically false formula.

For any n-argument boolean function f, we call point x a *local minimum point* of f if $f(x) = 1$ but for every child y of x in the lattice, $f(y) = 0$. The local minimum points of a minimized DNF formula represent its terms in our representation.

For two n-argument boolean functions f_1 and f_2 we define the set $Err(f_1, f_2)$ to be the set of points where they differ. I.e., $Err(f_1, f_2) = \{x \mid f_1(x) \neq f_2(x)\}$. The cardinality of $Err(f_1, f_2)$ is called the *distance* between f_1 and f_2 and is denoted by $d(f_1, f_2)$.

3. Limited Membership Queries

In this section, we present results concerning learning with the help of limited membership queries. We start with the description of a subroutine that is repeatedly used in all algorithms for this model.

3.1. Delimiting A Term

Algorithm DELIMIT takes a positive point and finds a set of candidates for a term in the target concept covering the point. This algorithm plays the role of the algorithms called "Reduce" in other works on learning monotone DNF (Angluin & Slonim, 1994). We choose a different name because those algorithms output a single monomial, whereas Algorithm DELIMIT finds a set of points that must include a correct term.

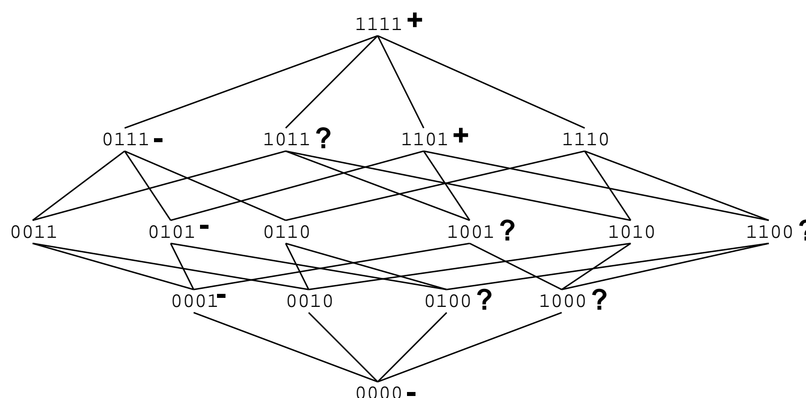

Figure 1. Example run of Algorithm DELIMIT for target concept x_1x_2. Boldface $+$'s, $-$'s and ?s indicate responses 1, 0, and $\perp$, respectively, of the LMQ oracle.

Consider, for example, the situation in Figure 1 where the target concept is x_1x_2, and we start with the known positive point $p = 1111$. With complete information, we would begin by querying each child of p, updating p to be the first positive child found. This process would be repeated until eventually we had $p = 1100$. After determining by membership queries that every child of p is negative, we could stop.

```
DELIMIT(p)
{
    offbits = 00···0   (n bits)
    ⟨root, DK⟩ = DOWN(p, offbits)
    ⟨P, DK⟩ = UP(DK)
    P = P ∪ {root}
    T = DK ∪ {root}
    Return ⟨T, P⟩
}
```

Figure 2. Algorithm DELIMIT. *offbits* is a bit array used in recursive subroutine DOWN to improve efficiency; *root* is a special positive point (with everything beneath it being negative or belonging to DK); DK is a certain set of points with LMQ $\perp$, created by DOWN and further thinned by UP; P is a certain set of positive points above DK, a useful byproduct of UP; T is the set of points among which the correct term of the target formula must lie.

Because Algorithm DELIMIT can make only limited membership queries, it may encounter the difficulty that some positive point p has children that are all "Don't know," or a mix of "No" and "Don't know." For instance, in the example in Figure 1, all queries of children of 1101 return 0 or $\perp$. In this case, Algorithm DELIMIT continues by querying all the children of *all* the "Don't know" points. Should it ever get another 1 response to a limited membership query, it replaces p by that point. What we have just described is the subroutine DOWN of DELIMIT, which invokes itself recursively upon finding a new positive point.

Detailed code of algorithm DELIMIT and its subroutines is given in Figures 2, 3 and 5. In our C-like pseudocode we often use a **For** loop over a changing set of points; for example, "**For** (each $b \in A$)". By this we mean that in each iteration of the loop the current point (i.e., b in this example) is marked, and that only unmarked points can be considered in the next iterations. Furthermore, the loop condition is checked every time, that is, we check for an existence of some unmarked point in the possibly changing set (A in this example). If one exists, we mark it and do the body of the loop which may add new (unmarked) points to the set or delete some existing points (either marked or unmarked). Points that are deleted before marking are not processed. Furthermore, to ensure that the algorithms are deterministic, we need that the current point is chosen according to some unambiguous rule, i.e., there must not be any choice as to which unmarked point of the set will be marked and processed next. Thus, we assume that there is some total order on all the elements in the sample space and use this to unambiguously choose the current point. When the points of the sample space correspond to assignments of 0 and 1 to n variables, the easiest total order is created by treating each point as an n-bit number. In all our algorithms that learn boolean formulas, we assume that this total order is used and always pick the point with the *lowest number*.

Another thing worth explanation about our pseudocode is the use of $\langle \cdot, \cdot, \ldots, \cdot \rangle$ on the left side of assignments and as return values for subroutines. We do this to avoid confusing global variables or things passed by reference. Thus, everything is passed by value (by making a copy in the called subroutine) and everything that the calling routine needs is returned to it explicitly. If many things need to be returned, then we put them in a tuple,

denoted by $\langle \cdot, \cdot, \ldots, \cdot \rangle$, and return the tuple. The tuple is basically just a convenient notation for a structure.

```
DOWN(p, offbits)
{
    DK = ∅
    C = { c : c is a child of p  &&  c ≥ offbits }
    While (C ≠ ∅)
    {
        a = maximal element of C with the lowest number
        Delete a from C
        If (LMQ(a) == 1)
            Return DOWN(a, offbits)
        Else If (LMQ(a) == ⊥)
        {
            DK = DK ∪ {a}
            C = C ∪ { c : c is a child of a  &&  c ≥ offbits }
        }
        Else If (a is a child of p)
        {
            i = bit where a and p differ
            offbits[i] = 1
        }
    }
    Return ⟨p, DK⟩
}
```

Figure 3. Subroutine DOWN. p is the point it is called on; *offbits* is a bit array used to improve efficiency; DK is a set of points with LMQ $\perp$ and with all descendants $\perp$ or negative; C is a set of points that have to be processed in the main loop.

The subroutine DOWN uses a variable *offbits* to improve efficiency. If a query to a child a of a known positive point p gives a 0 response, then we know that in any descendant of p, switching off the bit position that distinguishes a from p will lead to a negative point, because this point will be a descendant of a. Therefore, variable *offbits* keeps track of those bit positions that must be 1, allowing subroutine DOWN to save some queries.

Eventually DOWN is called on some positive point p and finds a (possibly empty) set DK of descendants of p such that the limited membership oracle responded $\perp$ for every point in DK, and all other proper descendants of p are known to be negative. Then DOWN returns and point p from then on is called the *root*. Since *root* is positive, and we know that every descendant of *root* not in set DK is negative, a term of the monotone DNF must lie somewhere in the set $DK \cup \{root\}$. This set is outlined in Figure 4.

The next major part of DELIMIT is the subroutine UP. Any point corresponding to a term of the target concept must have only positive ancestors. Subroutine UP ensures that

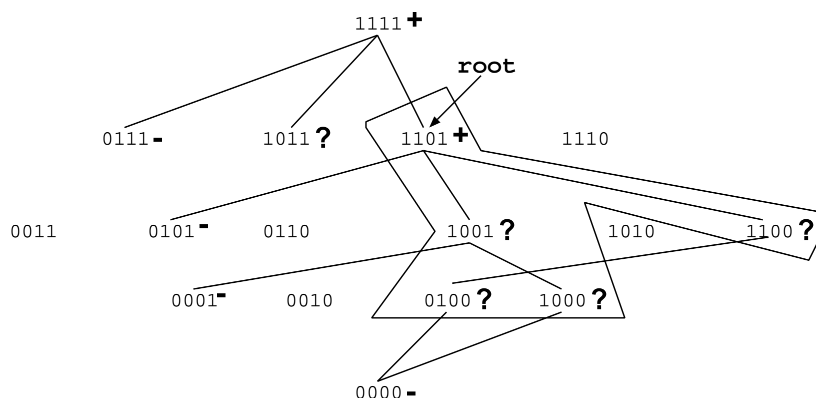

Figure 4. The set of candidate terms obtained by subroutine DOWN.

```
UP(DK)
{
    P = ∅
    For (each a ∈ DK)
    {
        A = {parents of a}
        For (each b ∈ A)
            If (LMQ(b) == 0)
            {
                Delete all descendants of b from DK
                Break                /* Out of the For (each b ∈ A) loop */
            }
            Else If (LMQ(b) == 1)
                P = P ∪ {b}
            Else
                A = A ∪ {parents of b}
    }
    Return ⟨P, DK⟩
}
```

Figure 5. Subroutine UP. DK is a set of points with LMQ $\perp$ that has to be thinned; P is a set of minimal ancestors with LMQ 1 of points in DK; A is a set of certain ancestors of the current point a.

no point in DK has any ancestor with LMQ 0. Thus, DELIMIT gets a thinned set of possible terms T. In the example of Figures 1 and 4, 0100 is deleted from DK because $\mathrm{LMQ}(0101) = 0$. If $\mathrm{LMQ}(1010) = 0$, then 1000 will also be deleted.

As a useful byproduct of subroutine UP, we get a set P, which consists of the minimal ancestors with LMQ 1 of the points in DK (i.e., the minimal points in the lattice ordering of the set of those ancestors of elements of DK that have LMQ 1). If there are no points in DK, P contains the *root* only.

We summarize our discussion in the following lemma. Recall that since DELIMIT is deterministic, the adversary may be thought of as choosing in advance the answers to all possible limited membership queries. We denote by $\mathrm{LMQ}(v)$ the value of this function on the point v.

LEMMA 1 *Let T and P be the outputs of running Algorithm* DELIMIT *on a positive point p when the target concept is a monotone DNF. Then*

1. *T contains a term of the target concept covering p.*
2. *For every $t \in T$, every point covered by t has* LMQ *1 or $\perp$.*
3. *For any v such that $\mathrm{LMQ}(v) = 1$, and for any $t \in T$ that covers v, t covers some point in P that is a descendant of v.*

Proof: Part 1 follows from the discussion above. Part 2 holds since every ancestor of the *root* is positive (since *root* itself is), and since the code in UP ensures this condition for every other point in T. To see Part 3, suppose v has $\mathrm{LMQ}(v) = 1$ and t is some point of T that covers v. If t is *root*, then *root* itself is a point in P that is a descendant of v. If t is some other element of T, then UP has made an upward search from t to find all the minimal ancestors of t with LMQ 1 and placed them in P. ■

We are now ready to analyze the running time of DELIMIT.

LEMMA 2 *For any monotone DNF target concept, Algorithm* DELIMIT *makes at most $n\ell + n$ limited membership queries, where ℓ is the number of $\perp$ responses received. In general, any sequence of s calls to Algorithm* DELIMIT *for the same target concept makes at most $n(\ell + s)$ limited membership queries.*

Proof: Every point queried by DELIMIT(p) is either a child of p, or a child of a previously queried point with LMQ 1, or else is within Hamming distance 1 of a $\perp$ point.

For the last case, each time we receive a $\perp$ response on a point v, we could in principle need to query all the children of v in DOWN and then all the parents of v in UP, except that there is a parent or a child of v that we must have queried before querying v. This leads to at most $n - 1$ queries for each $\perp$ received, plus 1 for the $\perp$ query itself. As long as the algorithm remembers the answers to all queries, this holds for any number of calls to DELIMIT on the same concept.

For the first two cases, let us first note that they only happen in subroutine DOWN. Let us call the point that DOWN was called on the *current point*, which can be either p or some previously queried point a with $\mathrm{LMQ}(a) = 1$. When DOWN makes a query on a child of the current point and receives a response of $\perp$, the query is already accounted for. If it receives a response of 0, then a bit is turned on in *offbits*, decreasing the Hamming distance between the current point and *offbits*. If it receives a response of 1, then a recursive call to DOWN is made, and the Hamming distance between the (new) current point and *offbits* is also one less. Since the maximum Hamming distance between two points is n, there can be at most n queries of this kind in every call to DELIMIT. ■

The running time of DELIMIT is polynomial in the number of queries it makes.

3.2. Learning Monotone Monomials from Limited Membership Queries Alone

We begin with a very simple application of Algorithm DELIMIT to learn monotone monomials from limited membership queries in the nonstrict model. This should elucidate the basic ideas at the heart of the more complicated algorithm for learning arbitrary monotone k-term DNF from limited membership queries that follows.

THEOREM 1 *Monotone monomials can be learned from no more than $n\ell + n + 1$ limited membership queries in the nonstrict model, where ℓ is the number of $\perp$ responses received.*

Proof: The method is to run Algorithm DELIMIT starting with the all 1's vector, and then output any term $t \in T$ that covers every point in P. (If either $\mathrm{LMQ}(11 \cdots 1) = 0$, or $\mathrm{LMQ}(11 \cdots 1) = \perp$ and the "down" phase of Algorithm DELIMIT finds no positive points, then we will output the empty DNF formula.)

First, observe that such a term t must exist in T, since the true target monomial covers every point in P and lies in T by point 1 of Lemma 1. The output term t covers every point with LMQ 1 by point 3 of Lemma 1, since it covers every point in P. And by point 2 of Lemma 1, it cannot cover any points with LMQ 0, since it is in T. The bound on the number of queries follows from Lemma 2, which counts all of them except for the first query to $11 \cdots 1$. ∎

The running time again is polynomial in the number of queries made.

Another computationally somewhat more efficient way of learning monotone monomials would be to run DELIMIT on the all 1's vector, and then output the monomial m that is the intersection of all the points in P. That is, the learner's output would have a variable if and only if that variable is in every term in P.

In this version, monomial m covers every point in P, so by point 3 of Lemma 1, it must cover all points v such that $\mathrm{LMQ}(v) = 1$. On the other hand, since m is the intersection of positive points, it must either be the correct monotone monomial or an ancestor of the correct monotone monomial, so m cannot cover any negative points.

Now we present a lower bound for this problem.

THEOREM 2 *For any integer c, if the limited membership oracle gives $\sum_{k=0}^{c} \binom{n}{k}$ responses of $\perp$, then any learner can be forced to use at least $\binom{n}{c+1} - 1$ limited membership queries* not counting those answered $\perp$ *to learn monotone monomials.*

Proof: Let the target concept be defined by a monomial of length $n - (c+1)$. It covers no point with more than $(c+1)$ 0-bits, and exactly one point with $(c+1)$ 0-bits. Now let us assume that all queries of a learning algorithm on points containing c or fewer 0-bits are answered by $\perp$, and its first $\binom{n}{c+1} - 2$ queries of points with at least $(c+1)$ 0-bits are answered by 0. After that there are at least two unqueried points with $(c+1)$ 0-bits. The corresponding two monomials are both consistent with the answers given so far and they differ on points that did not receive a $\perp$ response. Hence the learning algorithm needs at least one more query. ∎

COROLLARY 1 *For any fixed constant c, if the limited membership oracle gives $O(n^c)$ responses of $\perp$, then the learner can be forced to use $\Omega(n^{c+1})$ limited membership queries to learn monotone monomials.*

3.3. Learning Monotone k-term DNF from Limited Membership Queries Alone

We are now ready to present the algorithm ONLYLMQ, that learns monotone k-term DNF in the nonstrict model from limited membership queries alone. The detailed code is given in Figures 6 and 7.

```
FINDPOS(h)
{
    S = { maximal elements v ∈ {0,1}^n s.t. h(v) == 0 }
    For (each a ∈ S)
        If (LMQ(a) == 1)
            Return a
        Else If (LMQ(a) == ⊥)
            S = S ∪ {children of a}
    Return "no"
}
```

Figure 6. Subroutine FINDPOS. S is the set of maximal points in the sample space that are not covered by the current hypothesis h.

Algorithm ONLYLMQ initializes its hypothesis h to be the empty formula, and repeats the following.

By searching down from each maximal point not covered by hypothesis h, a point q with $\mathrm{LMQ}(q) = 1$ that is not covered by h is found. This is done by the subroutine FINDPOS of Algorithm ONLYLMQ. If no such point is found, then h is correct and the algorithm outputs it and halts.

Now the point q is given to Algorithm DELIMIT, which returns the sets T and P. If there is a single term in T that covers all the positive points in P not already covered by h, then this term is added to h and we repeat the main **For** loop by looking for another point with LMQ 1 that is not covered by the current hypothesis.

Otherwise, we have to add more than one term to h to cover all the points in P, and we begin a search for a set of terms to add. This search may disclose more positive points, so this process itself may have to be iterated. We begin by initializing *Terms* to be T and *Pos* to be those points in P that are not covered by h. We record the fact that point q has already been delimited by placing it in the set *Delimited*. We then call Algorithm DELIMIT on every point in *Pos* and for each call gather together the points from P in *NewPos* as well as add the terms from T to *Terms*. When all points in *Pos* have been delimited, we add the points gathered in *NewPos* to *Pos* and try to cover the newly enlarged set *Pos* with any $j = 2$ terms from *Terms*. If we succeed, we put these terms in h; if we fail, we keep enlarging *Terms*, gathering points in *NewPos*, then enlarging *Pos*, incrementing j, and trying again in the same manner to cover all points in *Pos* with j terms from *Terms*.

```
ONLYLMQ()
{
    h = "the empty DNF formula"
    For (ever)
    {
        q = FINDPOS(h)
        If (q == "no")
        {
            Output h
            Return
        }
        ⟨T, P⟩ = DELIMIT(q)
        Delimited = {q}
        Terms = T
        Pos = { r ∈ P : h(r) == 0 }
        For (j = 1;    ; j++)
            If (∃t_1, t_2, ..., t_j ∈ Terms s.t. ∀p ∈ Pos ((t_1 ∨ ... ∨ t_j)(p) == 1))
            {
                Add the terms t_1, ..., t_j to h
                Break            /* Out of the For (j = 1;    ; j++) loop */
            }
            Else
            {
                NewPos = ∅
                For (each p ∈ Pos s.t. p ∉ Delimited)
                {
                    ⟨T, P⟩ = DELIMIT(p)
                    Delimited = Delimited ∪ {p}
                    NewPos = NewPos ∪ { r ∈ P : h(r) == 0 }
                    Terms = Terms ∪ T
                }
                Pos = Pos ∪ NewPos
            }
    }
}
```

Figure 7. Algorithm ONLYLMQ, which learns monotone k-term DNF from LMQ's. *Delimited* is the set of points that have been delimited in the current iteration of the main loop; *Terms* is the set of candidate terms; *Pos* is a set of known positive instances that need to be covered by a certain number of points in *Terms*.

THEOREM 3 *Algorithm* ONLYLMQ *learns monotone k-term DNF from $O(kn^k + n^2\ell)$ limited membership queries in the nonstrict model, where ℓ is the number of $\perp$ responses it receives.*

Proof: We prove the theorem in three stages. First we argue that the algorithm eventually terminates with a DNF hypothesis h that correctly classifies all non-$\perp$ points. Next, in Lemmas 3 through 6, we argue that h includes at most k terms. In particular, Lemma 6 says that at all times through the run of the algorithm, h is at least as efficient—in terms of positive instances covered per DNF term—as the target concept. Finally, we argue that the query bound is correct.

* Algorithm DELIMIT terminates with a correct hypothesis. All the terms added to the hypothesis are at some point in a set T output by Algorithm DELIMIT. Therefore, by point 2 of Lemma 1, no term that we ever add to our hypothesis can err by classifying a point with LMQ 0 as positive.

Furthermore, since point 1 of Lemma 1 guarantees that each time we call the subroutine DELIMIT from a point we get at least one term in T covering that point, the algorithm must eventually succeed in covering the set *Pos* with some number of terms and break out of the "**For** ($j = 1$; ; j++)" loop. Since there are only a finite number of positive points, this means that the algorithm eventually terminates with a hypothesis that correctly classifies all non-$\perp$ points.

* Final hypothesis contains at most k terms. The following lemmas provide the argument that each hypothesis of Algorithm ONLYLMQ contains at most k terms.

LEMMA 3 *No term in the set Terms is ever implied by the current hypothesis h inside of the* "**For** ($j = 1$; ; j++)" *loop.*

Proof: The hypothesis h is constant during the execution of the loop. Every element of *Terms* entered *Terms* from the set T generated by calling Algorithm DELIMIT with this hypothesis h. The first call to DELIMIT (the one made immediately before entering the "**For** ($j = 1$; ; j++)" loop) is made on some point q such that $h(q) = 0$, and therefore no element of T can be implied by h. All subsequent calls to DELIMIT are made on some point $r \in$ *Pos*. The set *Pos* is constructed so that $h(r) = 0$, so again no points in the set T returned by DELIMIT can be implied by h, because T contains only descendants of r.

■

LEMMA 4 *Whenever Algorithm* ONLYLMQ *attempts to cover the set of positive points Pos with a disjunction of j terms from Terms (line* "**If** $(\exists t_1, t_2, \ldots, t_j \in$ *Terms s.t.* $\forall p \in$ *Pos* $((t_1 \vee \ldots \vee t_j)(p) == 1))$" *in the code), the set Terms actually contains at least j distinct terms of the target concept.*

Proof: The proof is by induction on j. For the base case, $j = 1$, the set *Terms* $= T$. Thus the base case is provided by point 1 of Lemma 1, which says that there must must be a term of the target concept in T at the end of the first call to Algorithm DELIMIT.

For the inductive step, suppose that we are trying to cover *Pos* with $j + 1$ terms. Then we know that we tried and failed to cover the previous version of *Pos* with j terms. By the inductive hypothesis, this implies that the previous version of *Terms* contained at least j

distinct terms of the target concept. If the previous version of *Terms* in fact contained more than j distinct terms of the target concept, then we are done.

Otherwise, the previous version of *Terms* contained exactly j distinct terms of the target concept, but nevertheless failed to cover all points in *Pos*. That is, there was some point p in the previous version of *Pos* not covered by any of those particular j terms. This point p cannot have been delimited before the attempt to cover *Pos* with j terms, because if it had, the previous version of *Terms* would have contained an element covering p. Therefore, after attempting and failing to cover *Pos* with j terms p is delimited. Now by point 1 of Lemma 1, a term of the target concept covering p is in the set T output by this call to DELIMIT, and this point is added to *Terms*. Thus *Terms* now contains at least $j+1$ distinct terms of the target concept. ■

LEMMA 5 *Let h' be the hypothesis that Algorithm* ONLYLMQ *obtains when it updates its hypothesis h. Then h' covers all points with* LMQ *1 that are covered by any term in Terms.*

Proof: Assume the contrary holds for h, *Terms*, and h'. Let $t \in Terms$ be a term and x be a point with LMQ 1 such that $h'(x) = 0$ and $t(x) = 1$.

Term t was added to *Terms* by some call to subroutine DELIMIT. At that call, according to point 3 of Lemma 1, the set P must have contained a point p such that t covered p and p is a descendant of x. Since $h'(x) = 0$, we know $h(x) = 0$, so $h(p) = 0$. Therefore, after the call to Algorithm DELIMIT when the outputs had $t \in T$ and $p \in P$, point p was added to *Pos*. Thus since h' is satisfied by all points in *Pos*, it must be that $h'(p) = 1$ and thus $h'(x) = 1$, a contradiction. ■

LEMMA 6 *Consider the hypothesis h of Algorithm* ONLYLMQ *at any point in the run of it for a monotone DNF target concept c. There is a monotone DNF d consisting of at least $\#(h)$ distinct terms from c such that $\{v : d(v) = \mathrm{LMQ}(v) = 1\} \subseteq \{v : h(v) = 1\}$.*

Proof: The proof is by induction on the number of times that h is enlarged by adding new terms. The base case with an empty h holds trivially. Let us now assume that the lemma holds up to the k-th time we add new terms to h, and that it does not hold the $k+1$-st time. Let us denote h after these additions of terms by h_k and h_{k+1}, respectively. That is:

(1) there are $\#(h_k)$ terms $t_1, t_2, \ldots, t_{\#(h_k)}$ in c such that they cover a subset of those points with LMQ 1 that are covered by h_k,

(2) for any way we take $\#(h_{k+1})$ terms from c, there always is some point w with LMQ 1 that is covered by these terms but not by h_{k+1}.

Furthermore, let us assume that exactly j terms were added to h_k when making h_{k+1}, that is, $\#(h_{k+1}) - \#(h_k) = j$. At the moment these j terms were added, Lemma 4 guaranteed the existence of j terms $t'_1, t'_2, \ldots, t'_j$ from c in *Terms*. By Lemma 3 none of $t'_1, t'_2, \ldots, t'_j$ was implied by h_k. Now suppose we take terms $t_1, t_2, \ldots, t_{\#(h_k)}, t'_1, t'_2, \ldots, t'_j$, which are all in c, for a total of $\#(h_{k+1})$ terms. By our assumption there is a point w, such that $\mathrm{LMQ}(w) = 1$, that is covered by these terms but not by h_{k+1}. By the assumption, w cannot be covered by any of the $t_1, t_2, \ldots, t_{\#(h_k)}$, or it would be covered by h_k and hence

h_{k+1}. Therefore, it must be the case that w is covered by some of $t'_1, t'_2, \ldots, t'_j$. But then, by Lemma 5, h_{k+1} must cover w, a contradiction. ■

Lemma 6 implies that the hypothesis h of Algorithm ONLYLMQ never contains more than k terms, which is what we needed to show. What remains for the proof of Theorem 3 is to show that the number of limited membership queries is $O(kn^k + n^2\ell)$.

* Number of queries made. A monotone k-term DNF formula can have at most n^k maximal negative points. This follows from the following observations. If v is a negative point and t is a term, then there must be a variable x_i in t such that $v[i] = 0$, or else v would satisfy t. Thus, by setting to 0 at least one variable from each term, we can obtain exactly the negative points. Maximal negative points are a subset of the points that can be obtained by setting exactly one variable from each term to 0, which can be done in at most n^k ways. Thus, each time S is initialized in subroutine FINDPOS, it can have at most n^k elements.

FINDPOS is called at most $k + 1$ times, since each time it is called (except the last), it returns a new positive point q which causes at least one term to be added to h.

Each call to FINDPOS performs at most one LMQ for each of at most n^k elements initially in S. After that, each element queried is a child of a point with LMQ $\perp$, so the number of such queries over all calls to FINDPOS is at most $n\ell$. (We assume FINDPOS caches answers.) Thus, the number of LMQ's made by FINDPOS is at most $(k+1)n^k + n\ell$.

The only other LMQ's are made in calls to DELIMIT, and by Lemma 2 the total number of queries is bounded by $n(\ell + s)$, where s is the number of calls to DELIMIT. DELIMIT is called at most k times on points q returned by FINDPOS, and is called at most once for each element that is ever added to *Pos*. The elements added to *Pos* are all returned in P by UP, which means that they are all parents of points with LMQ $\perp$. Hence, at most $n\ell$ elements are ever added to *Pos*, and the total number s of calls to DELIMIT is bounded by $k + n\ell$. This gives a bound of $n(\ell + k + n\ell)$ for the number of LMQ's made in the calls to DELIMIT. By adding the bound for the number on LMQ's made in calls to FINDPOS, we can easily get the desired bound of $O(kn^k + n^2\ell)$ on the total number of LMQ's. ■

The running time of the algorithm ONLYLMQ is clearly polynomial in the number of queries it makes.

3.4. Learning Monotone DNF from Equivalence and Limited Membership Queries

In this subsection, we give a very simple algorithm that learns monotone DNF with an unbounded number of terms from equivalence and limited membership queries.

THEOREM 4 *Monotone DNF formulas can be learned from $n(m+\ell)$ limited membership queries together with $m + 1$ equivalence queries in the nonstrict model, or together with $m + \ell + 1$ equivalence queries in the strict model, where ℓ is the number of $\perp$ responses received, and m is the number of terms in the target monotone DNF.*

Proof: We modify Angluin's algorithm for learning monotone DNF from ordinary membership and equivalence queries (Angluin, 1988), by using Algorithm DELIMIT. The com-

```
EQToo()
{
      h = "the empty DNF formula"
      While ((v = EQ(h)) ≠ "yes")
            If (h(v) == 0)
            {
                  ⟨T, P⟩ = Delimit(v)
                  Add terms of T to h
            }
            Else
                  For (each term t of h)
                        If (t(v) == 1)
                              Delete term t from h
      Output h
}
```

Figure 8. Algorithm EQToo for learning monotone DNF from EQ's and LMQ's. v is the current counterexample.

plete code of the algorithm is given in Figure 8. Note that the same algorithm works for both the nonstrict and strict models.

The Up subroutine of Algorithm Delimit guarantees that all terms we put in h are either correct, or err only by classifying negative points with LMQ $\perp$ as positive. Lemma 1 guarantees that each time we call Algorithm Delimit we get a new term of the target monotone DNF.

In the nonstrict model, terms that cover negative points with LMQ $\perp$ do not matter, so there is at most one call to Delimit and one equivalence query per term. In the strict model we may add terms with LMQ $\perp$ that cover negative points, and such terms must be subsequently deleted in response to equivalence queries. There is at most one equivalence query for each such term, provided that we modify the algorithm to remember deleted terms and never add them again to the hypothesis. This implies the bounds for the number of equivalence queries. The bound for the number of LMQ's then follows from Lemma 2. ■

The running time of the algorithm is polynomial in the number of queries it makes.

Note that polynomial learnability of monotone DNF from equivalence and limited membership queries is implied by the stronger result of Section 4. It is at least as difficult to learn from a malicious membership oracle as it is from a limited membership oracle, as pointed out in Subsection 2.2, so that algorithm for monotone DNF could be used here. The direct algorithm does, however, give better bounds on the number of queries required.

4. Malicious Membership Queries

In this section, we present and analyze an algorithm that uses equivalence and malicious membership queries to learn monotone DNF formulas. The key idea is to depend on equivalence queries as much as possible, since they are correct.

4.1. The Algorithm

The algorithm keeps track of all the counterexamples and their labels received through equivalence queries and consults them first, before asking a membership query. The pairs of counterexamples and their labels are kept in a set named *CounterExamples*. Obviously, for a positive counterexample v, if $x \geq v$ then it is not worth making a membership query about x; it must be a positive point. Similarly, for a negative counterexample v, if $x \leq v$ then x has to be a negative point of the target formula. For this reason we define a subroutine CHECKEDMQ and use it instead of a membership query. The subroutine is given in Figure 9.

```
CHECKEDMQ(x, CounterExamples)
{
    If (∃⟨v,1⟩ ∈ CounterExamples s.t. x ≥ v)
        Return 1
    If (∃⟨v,0⟩ ∈ CounterExamples s.t. x ≤ v)
        Return 0
    Return MMQ(x)
}
```

Figure 9. Subroutine CHECKEDMQ.

As in (Angluin, 1988) and (Angluin & Slonim, 1994), our algorithm also uses a subroutine REDUCE in order to move down in the lattice from a positive counterexample. All the membership queries are done using the subroutine CHECKEDMQ, which possibly lets the algorithm avoid some incorrect answers. The subroutine REDUCE is given in Figure 10.

```
REDUCE(v, CounterExamples)
{
    For (each child w of v)
        If (CHECKEDMQ(w, CounterExamples) == 1)
            Return REDUCE(w, CounterExamples)
    Return v
}
```

Figure 10. Subroutine REDUCE.

The algorithm for exactly identifying monotone DNF formulas using equivalence queries and malicious membership queries is given in Figure 11.

```
LearnMonDNF()
{
    CounterExamples = ∅
    h = "the empty DNF formula"
    While ((v = EQ(h)) ≠ "yes")
    {
        Add ⟨v, (1 − h(v))⟩ to CounterExamples
        If (h(v) == 0)
        {
            w = Reduce(v, CounterExamples)
            Add term w to h
        }
        Else
            For (each term t of h)
                If (t(v) == 1)
                    Delete term t from h
    }
    Output h
}
```

Figure 11. The algorithm for learning monotone DNF from EQ's and MMQ's.

The algorithm is based on a few simple ideas. A positive counterexample is reduced to a point that is added as a term to the existing hypothesis h, which is a monotone DNF. That is, the new hypothesis classifies the latest counterexample and possibly some other points as positive.

Negative counterexamples are used to detect inconsistencies between membership and equivalence queries. They show that there have been errors in membership queries that have caused wrong terms to be added to the hypothesis. The algorithm reacts by removing all the terms that are inconsistent with the latest counterexample. These are the terms that have the negative counterexample above them. A term is removed only when there is a negative counterexample above it.

4.2. *Analysis of* LearnMonDNF

Theorem 5 LearnMonDNF *learns the class of monotone DNF formulas in polynomial time using equivalence and malicious membership queries.*

We need a definition and a simple lemma before proving the theorem.

Let h^* be a monotone boolean function on $\{0,1\}^n$, and let h' be an arbitrary boolean function on $\{0,1\}^n$. Let C be any subset of $\{0,1\}^n$. The *monotone correction of* h' *with* h^* *on* C, denoted $mc(h', h^*, C)$, is the boolean function h'' defined for each string $x \in \{0,1\}^n$ as follows.

$$h''(x) \stackrel{\text{def}}{=} \begin{cases} 1 & \text{if there exists } y \in C \text{ such that } y \leq x \text{ and } h^*(y) = 1, \\ 0 & \text{if there exists } y \in C \text{ such that } x \leq y \text{ and } h^*(y) = 0, \\ h'(x) & \text{otherwise.} \end{cases}$$

Note that since h^* is monotone, the first two cases above cannot hold simultaneously. It is also clear that if the value of $h''(x)$ is determined by one of the first two cases, $h''(x) = h^*(x)$. We prove a simple monotonicity property of the monotone correction operation.

LEMMA 7 *Suppose h^* is a monotone boolean function and h' is an arbitrary boolean function on $\{0,1\}^n$. Let $C_1 \subseteq C_2$ be two subsets of $\{0,1\}^n$. Let $h_1 = mc(h', h^*, C_1)$ and $h_2 = mc(h', h^*, C_2)$. Then the set of points on which h_2 and h^* differ is contained in the set of points on which h_1 and h^* differ. That is, $Err(h_2, h^*) \subseteq Err(h_1, h^*)$.*

Proof: Let x be an arbitrary point on which $h_2(x) \neq h^*(x)$. Then it must be that $h_2(x) = h'(x)$ and there does not exist any point $y \in C_2$ such that $x \leq y$ and $h^*(y) = 0$ or $y \leq x$ and $h^*(y) = 1$. Since C_1 is contained in C_2, there is no point $y \in C_1$ such that $x \leq y$ and $h^*(y) = 0$ or such that $y \leq x$ and $h^*(y) = 1$. Thus, $h_1(x) = h'(x)$ and $h_1(x) \neq h^*(x)$. Consequently, $Err(h_2, h^*) \subseteq Err(h_1, h^*)$. ■

Now we start the proof of Theorem 5.

Proof: Let h^* denote the target concept, an arbitrary monotone DNF formula over $\{0,1\}^n$ with m terms. Let ℓ be a bound on the number of strings whose MMQ's are answered incorrectly. Because equivalence queries are answered correctly, if the algorithm ever halts, the hypothesis output is correct, so we may focus on proving a polynomial bound on the running time.

Since LEARNMONDNF is deterministic and the target concept h^* is fixed, we may assume that the adversary chooses in advance how to answer all the queries, that is, chooses a sequence $y_1, y_2, \ldots$ of counterexamples to equivalence queries and a set S of strings on which to answer MMQ's incorrectly. Note that $|S| \leq \ell$.

In turn, these choices determine a particular computation of LEARNMONDNF which we now focus on. It suffices to bound the length of this computation. In this computation the answers to MMQ's agree with the boolean function h_0 defined as follows. $h_0(x) = h^*(x)$ for all strings $x \notin S$ and $h_0(x) = 1 - h^*(x)$ for all strings $x \in S$. Also, if CHECKEDMQ is called with string x and set $C =$ *CounterExamples*, the answer agrees with the boolean function $mc(h_0, h^*, C)$.

The set *CounterExamples* only changes when a new counterexample is received. Therefore, the successive distinct sets of counterexamples in this computation can be denoted by $C_0, C_1, \ldots$, where $C_0 = \varnothing$ and $C_i = C_{i-1} \cup \{y_i\}$, for $i = 1, 2, \ldots$. If we also define

$$h_i = mc(h_0, h^*, C_i)$$

for $i = 1, 2, \ldots$, then CHECKEDMQ answers according to h_0 until the first counterexample is received, then according to h_1 until the second counterexample is received, and so on.

Clearly, since h_0 disagrees with h^* on at most ℓ strings, $d(h_0, h^*) \leq \ell$. Since the sets $C_0, C_1, \ldots$ are monotonically nondecreasing, Lemma 7 shows that $Err(h_i, h^*) \subseteq Err(h_{i-1}, h^*)$ for $i = 1, 2, \ldots$.

We say that a counterexample y_i *corrects a positive error* at point x if $h_{i-1}(x) = 1$ but $h_i(x) = h^*(x) = 0$. We say that a counterexample y_i *corrects a negative error* at point x if $h_{i-1}(x) = 0$ but $h_i(x) = h^*(x) = 1$. Note that from the construction of CHECKEDMQ it follows that positive errors can be corrected only by negative counterexamples and negative errors can be corrected only by positive counterexamples. Let there be ℓ_p positive and ℓ_n negative errors corrected in the whole computation. Of course, $\ell_p + \ell_n \leq \ell$.

CLAIM 1 *If* REDUCE *is called after counterexample y_i and before counterexample y_{i+1}, it returns a local minimum point of h_i.*

Proof: After y_i is added to *CounterExamples*, CHECKEDMQ answers according to h_i. The claim follows from the construction of REDUCE. ■

CLAIM 2 *The following condition is preserved. At the $(i+1)$th equivalence query* $\mathrm{EQ}(h)$*, each term of h is a positive point of h_i.*

Proof: We prove the claim by induction.

Basis: The first EQ is made on an empty formula. Thus, the claim is vacuously true.

Induction step: Suppose the claim is true up to the ith EQ. Let h' be the hypothesis h at the ith EQ and h'' be the hypothesis h at the $(i+1)$th EQ. There are two cases to consider.

Case 1: y_i is a positive counterexample. Then $h_i(x) = 1$ if and only if $h_{i-1}(x) = 1$ or $x \geq y_i$. Let t be the term returned by REDUCE with parameters y_i and *CounterExamples*. Then $h'' = h' \vee t$. Let t'' be a term in h''. Then either t'' is a term of h' or $t'' = t$. If t'' is a term of h' then $h_{i-1}(t'') = 1$ by the inductive assumption and therefore $h_i(t'') = 1$. If $t'' = t$ then $h_i(t'') = 1$ since t was returned by REDUCE$(y_i$, *CounterExamples*$)$ which used CHECKEDMQ, which answered according to h_i.

Case 2: y_i is a negative counterexample. Then $h_i(x) = 1$ if and only if $h_{i-1}(x) = 1$ and $x \not\leq y_i$. Let t'' be a term in h'', which consists of all those terms t' of h' such that $t' \not\leq y_i$. Therefore, $t'' \not\leq y_i$ and by the inductive assumption $h_{i-1}(t'') = 1$. It follows that $h_i(t'') = 1$. ■

CLAIM 3 *Once a term x is deleted from hypothesis h, it can never reappear in it.*

Proof: Since x was deleted, there must have been a negative counterexample y_i such that $y_i \geq x$. But then $(y_i, 0)$ belongs to *CounterExamples* and the call CHECKEDMQ$(x$, *CounterExamples*$)$ can never return 1 again, which is necessary for x to be added to h. ■

We divide the run of the algorithm into non-overlapping *stages*. A new stage begins either at the beginning of the run or with a new negative counterexample. Thus with each new stage *CounterExamples* contains one more negative counterexample and some (possibly none) new positive counterexamples. The following claim establishes that the distance $d(h_i, h^*)$ decreases with every new stage.

CLAIM 4 *Every negative counterexample corrects at least one error. More formally, if y_i is a negative counterexample, then there exists $x \in \{0,1\}^n$ such that $h_{i-1}(x) = 1$ and $h_i(x) = h^*(x) = 0$.*

Proof: Let y_i be a negative counterexample returned by $\mathrm{EQ}(h)$. Hence $h(y_i) = 1$, and there is some term $x \leq y_i$ in h. By Claim 2, $h_{i-1}(x) = 1$.

Since $h^*(y_i) = 0$ and $y_i \geq x$ it follows that $h^*(x) = 0$. By the definition of h_i it follows that $h_i(x) = 0$. ■

From Claim 4 it follows that there are at most ℓ_p negative counterexamples. Hence there are at most $\ell_p + 1$ stages in the run of the algorithm.

We divide each stage of the algorithm into non-overlapping *substages*. A substage begins either at the beginning of a stage or with a new positive counterexample that corrects an error. Obviously there can be no more than ℓ_n positive counterexamples that correct errors and hence no more than $\ell_p + \ell_n + 1$ substages in the whole run of the algorithm. The distance $d(h_i, h^*)$ decreases with every new substage. If, however, functions h_i and h_j belong to the same substage, they are equivalent and their local minima are the same. This allows us to bound the total number of positive counterexamples.

CLAIM 5 *Every new positive counterexample is reduced to a local minimum point of $h_0, h_1, \ldots$ that has not been found earlier.*

Proof: Let v be a positive counterexample that REDUCE is started with. Let t be the point REDUCE(v, *CounterExamples*) returns. Assume, by way of contradiction, that t has already been found before. From Claim 3 it follows that t is a term in h. Since $v \geq t$, it follows that $h(v) = 1$. This is a contradiction to the assumption that v is a positive counterexample. ■

We denote the set of local minimum points of a boolean function f by *Lmp*(f). We bound the total number of different local minima of the functions $h_0, h_1, \ldots$.

LEMMA 8 *Let f and f' be n-argument boolean functions such that $Err(f, f') = \{x\}$. Then*

(a) If $f'(x) = 1$ then $|Lmp(f') - Lmp(f)| \leq 1$.

(b) If $f'(x) = 0$ then $|Lmp(f') - Lmp(f)| \leq n$.

Proof:

(a) The only point that can be a local minimum of f' and is not a local minimum of f, is x itself. The claim follows immediately.

(b) Any point which is a local minimum of f' but not of f is a parent of x. Since x has at most n parents, the claim follows. ■

COROLLARY 2 *Let f and f' be n-argument boolean functions such that $Err(f, f')$ contains d_p positive points of f' and d_n negative points of f'. Then*

$$|\mathit{Lmp}(f') - \mathit{Lmp}(f)| \leq nd_n + d_p.$$

COROLLARY 3 *Let $g_0, g_1, \ldots, g_r$ be the subsequence of $h_0, h_1, \ldots$, such that each g_i is the first of all the h_j's in its substage. Let $Err(h^*, g_{i-1}) - Err(h^*, g_i)$ contain $\ell_{p,i-1}$ positive and $\ell_{n,i-1}$ negative points of h^* for all $i = 1, 2, \ldots, r$. Let $Err(h^*, g_r)$ contain $\ell_{p,r}$ positive and $\ell_{n,r}$ negative points of h^*. Then the total number of different local minima of functions $g_0, g_1, \ldots, g_r, h^*$ is bounded above by $m + n\sum_{i=0}^{r} \ell_{n,i} + \sum_{i=0}^{r} \ell_{p,i}$.*

Proof: Note that $g_0, g_1, \ldots, g_r$ are the different functions in $h_0, h_1, \ldots$, and that subroutine CHECKEDMQ first answers according to g_0, then according to g_1 and so on. Obviously, $Err(h^*, g_i) \subseteq Err(h^*, g_{i-1})$ and $Err(g_{i-1}, g_i) = Err(h^*, g_{i-1}) - Err(h^*, g_i)$ for all $i = 1, 2, \ldots, r$. Also note that for each $i = 0, 1, \ldots, r-1$, one of $\ell_{p,i}$ and $\ell_{n,i}$ is 0, but $\ell_{p,r}$ and $\ell_{n,r}$ may both be positive.

We want to find $\left|\bigcup_{i=0}^{r} \mathit{Lmp}(g_i) \cup \mathit{Lmp}(h^*)\right|$, knowing that $\left|\mathit{Lmp}(h^*)\right| = m$. Since

$$\bigcup_{i=0}^{r} \mathit{Lmp}(g_i) \cup \mathit{Lmp}(h^*)$$
$$\subseteq \mathit{Lmp}(h^*) \cup \big(\mathit{Lmp}(g_r) - \mathit{Lmp}(h^*)\big) \cup \bigcup_{i=0}^{r-1} \big(\mathit{Lmp}(g_i) - \mathit{Lmp}(g_{i+1})\big),$$

from Corollary 2 it follows that

$$\left|\bigcup_{i=0}^{r} \mathit{Lmp}(g_i) \cup \mathit{Lmp}(h^*)\right| \leq \left|\mathit{Lmp}(h^*)\right| + \big(n\ell_{n,r} + \ell_{p,r}\big) + \sum_{i=0}^{r-1} \big(n\ell_{n,i} + \ell_{p,i}\big)$$

and the bound follows. ■

Since each error can be corrected at most once, it follows that $\sum_{i=0}^{r} \ell_{n,i} \leq \ell_n$ and $\sum_{i=0}^{r} \ell_{p,i} \leq \ell_p$. Hence the total number of the local minima and the total number of positive counterexamples that can be found in a computation is bounded by $m + n\ell_n + \ell_p$. The number of negative counterexamples in a complete run is bounded by the number of positive errors. The total number of counterexamples is therefore bounded by $m + \ell_n n + \ell_p + \ell_p \leq m + \ell(n+1) = O(m + \ell n)$.

We now count the number of membership queries in a complete run of the algorithm. Each positive counterexample v may cause at most $n(n+1)/2$ membership queries, before REDUCE(v, *CounterExamples*) returns. Therefore there can be at most $O(mn^2 + \ell n^3)$ membership queries in a complete run of the algorithm.

It is also clear that the running time of the algorithm is polynomial in m, n and ℓ. This concludes the proof of Theorem 5. ■

Comparing LEARNMONDNF with the algorithm EQTOO of Theorem 4, we see that LEARNMONDNF is able to cope with MMQ's instead of the more benign LMQ's, but

at a cost of making more queries overall. In particular, it uses $O(m+\ell n)$ equivalence queries, versus $m+\ell+1$ for EQTOO, and $O(mn^2+\ell n^3)$ membership queries, versus $mn+\ell n$ for EQTOO. It is open whether an algorithm to learn monotone DNF formulas using EQ's and MMQ's can attain query complexity closer to that of EQTOO.

5. Finite Exceptions

5.1. Exceptions

For a concept (X, f) and a finite set $S \subseteq X$, we define *the concept* (X, f) *with exceptions* S, denoted *xcpt*$((X, f), S)$, as the concept (X, f') where $f'(w) = f(w)$ for strings in $X - S$, and $f'(w) = 1 - f(w)$ for strings in S. (Thus f and f' have the same domain, and are equal except on the set of strings S, which is a subset of their common domain.) It is useful to note that S is partitioned by (X, f) into into the set of *positive exceptions* S_+ that are classified as negative by f, and the set of *negative exceptions* S_- that are classified as positive by f. When the domain X of a function f is clearly understood and we do not wish to mention it explicitly, we often just call this function itself a concept and we also use a shorthand notation for *xcpt*$((X, f), S)$, namely, we just write *xcpt*(f, S).

A concept class (R, Dom, μ) is *closed under finite exceptions* provided that for every concept (X, f) represented by (R, Dom, μ) and every finite set $S \subseteq X$, the concept *xcpt*$((X, f), S)$ is also represented by (R, Dom, μ). If, in addition, there is a fixed polynomial of two arguments such that the concept *xcpt*$((X, f), S)$ is of size bounded by this polynomial in the size of (X, f) and $||S||$, we say that (R, Dom, μ) is *polynomially closed under finite exceptions*.

This definition differs from a similar earlier definition (Board & Pitt, 1992) in that we do not require the existence of a polynomial-time algorithm that produces the new concept given the old concept and a list of exceptions. However, for the classes that we consider there are such algorithms.

We define a natural operation of adding finite exception tables to a class of concepts to produce another class of concepts that “embeds” the first and is polynomially closed under finite exceptions.

We assume $\Sigma \subseteq \Gamma$ and $|\Gamma| \geq 2$. We define a simple encoding e that takes a string r from Γ^* and a finite set of strings $S \subseteq \Sigma^*$ and produces a string r' in Γ^* from which r and the elements of S can easily be recovered, and is such that $|r'| = 2(1+|r|+||S||)$. The details of the encoding are as follows.

Assume that 0 and 1 are distinct symbols in Γ. We define

$$e_b(b_1 b_2 \ldots b_j) \stackrel{\text{def}}{=} bbbb_1bb_2 \ldots bb_j,$$

for $b \in \{0,1\}$ and $b_1, b_2, \ldots, b_j \in \Gamma$. Note that $|e_b(w)| = 2(1+|w|)$ for every string $w \in \Gamma^*$. We then define the encoding of r and S as

$$r' = e(r, S) \stackrel{\text{def}}{=} e_0(r)e_1(s_1)e_0(s_2)\ldots e_{k \bmod 2}(s_k),$$

where $s_1, s_2, \ldots, s_k$ are the strings in S.

Given a concept class (R, Dom, μ), we define the *class obtained from it by adding exception tables* to be (R', Dom', μ'), where R' is the set of all strings of the form $e(r, S)$ such that $r \in R$ and S is a finite subset of $Dom(r)$, and for each $r' \in R'$, the concept represented by $r' = e(r, S)$ is the concept represented by r with exceptions S, that is, $(Dom'(r'), \mu'(r')) = xcpt((Dom(r), \mu(r)), S)$.

For example, adding exception tables to the monotone DNF formulas produces a concept class which we term *monotone DNF formulas with finite exceptions*. More detailed discussion of classes obtained by adding exception tables and of polynomial closure under finite exceptions can be found in Subsection 5.2.

5.2. Examples and Lemmas

EXAMPLE: The class of regular languages represented by DFA's is polynomially closed under finite exceptions. Board and Pitt give an algorithm that takes as input a DFA M and an exception set S, and produces a new DFA for $xcpt(M, S)$ (Board & Pitt, 1992). The DFA's size is polynomial in the size of M and S. □

EXAMPLE: Another example of a class that is polynomially closed under finite exceptions is the class of boolean decision trees. This result is taken from (Board & Pitt, 1992) but since the construction is not given there, we sketch it here. □

LEMMA 9 *The class of boolean decision trees is polynomially closed under finite exceptions.*

Proof: Let T be a decision tree on n variables. Let S be the exception set for T. We construct the decision tree for $xcpt(T, S)$ as follows. We treat each exception point $x \in S$ individually. First we walk down from the root of the original tree T to see where x is located in it. If this leads us to a leaf with depth n (i.e., if all variables are tested on this path), then we just reverse the value of the leaf, because this path is for x only. However, if we find ourselves at a leaf with depth less than n, we have to add new internal nodes to the tree. Denote the value of this leaf by b. We then continue the path that led us to this leaf with a path in which all the remaining variables are tested. We end the path by a leaf with value $1 - b$. For each new internal node on the path, we make the other child (the one not on the path) a leaf, and give it the original value b. Thus, each counterexample adds at most n new internal nodes to the tree. The size of the new tree, measured as the number of internal nodes, is bounded by $|T| + n \times |S| = |T| + ||S||$. ■

EXAMPLE: One more interesting example is the class of DNF formulas. □

LEMMA 10 *The class of DNF formulas is polynomially closed under finite exceptions.*

Proof: Let f be an m-term DNF formula over n variables and S be an exception set for it. Let S be partitioned into the sets of positive and negative exceptions (S_+ and S_-, respectively), as described in Section 5.1. We construct a DNF formula for $xcpt(f, S)$ from the formula $(f \wedge f_-) \vee f_+$, where f_- is a DNF formula which is true on all the points in

its domain except the ones in S_-, and f_+ is a DNF formula which is true exactly on the points in S_+. The domain for all these formulas is $\{0,1\}^n$.

Obtaining f_+ is easy—straightforward disjunction of all the *terms* in S_+, where we make terms from points by substituting the respective variable for a 1 value of a coordinate and its negation for a 0 value. Obtaining f_- is harder. First we make a decision tree corresponding to f_-. We put each point from S_- individually in the tree as a 0-valued leaf at the end of a path of length n. All the remaining leaves get value 1. Then for each leaf with value 1 we make a term that will go into f_- by following the path from this leaf to the root. Obviously f_- has at most $n \times |S_-|$ terms. Thus, after "multiplying" the terms out, the formula $(f \wedge f_-) \vee f_+$ will have at most $mn \times |S_-| + |S_+| \leq (mn+1) \times |S|$ terms. ■

EXAMPLE: By duality it follows that the class of CNF formulas is polynomially closed under finite exceptions. □

Note that stronger bounds on the size of the new formula can be obtained by using the result in (Zhuravlev & Kogan, 1985). We, however, chose to present a simpler argument. Also note that the size bound is insufficient for *strong polynomial closure under exception lists* as defined in (Board & Pitt, 1992).

EXAMPLE: As our final example we show that any class that is obtained by adding exception tables to another class is polynomially closed under finite exceptions. □

LEMMA 11 *Let (R, Dom, μ) be any class of concepts. Then the concept class obtained from it by adding exception tables is polynomially closed under finite exceptions.*

Proof: Let (R', Dom', μ') be the class obtained from (R, Dom, μ) by adding exception tables, as defined in Section 5.1. Let (X', f') be any concept from (R', Dom', μ') and let $r' \in R'$ be a shortest representation of (X', f'). Then there exists a concept $r \in R$ and a finite set $S \subseteq Dom(r)$, such that $(Dom'(r'), \mu'(r')) = xcpt((Dom(r), \mu(r)), S)$ and $|r'| = 2(1 + |r| + ||S||)$. Let $S' \subseteq Dom'(r') = Dom(r)$ be any finite set. Let concept h'' be defined as $h'' \stackrel{\text{def}}{=} xcpt((Dom'(r'), \mu'(r')), S')$. It is easy to see that $h'' = xcpt((Dom(r), \mu(r)), S \bigtriangleup S')$ and thus h'' is represented by some $r'' \in R'$ with size $2(1 + |r| + ||S \bigtriangleup S'||) \leq 2(1 + |r| + ||S|| + ||S'||) = |r'| + 2||S'||$. ■

COROLLARY 4 *The class of monotone DNF formulas with finite exceptions is polynomially closed under finite exceptions.*

5.3. Learning Monotone DNF Formulas With Finite Exceptions

In this section, we present an algorithm that learns the class of monotone DNF formulas with finite exceptions. The target concept is a boolean function on n variables $h^* \stackrel{\text{def}}{=} xcpt(h^*_M, S^*)$, where h^*_M is some monotone DNF formula and S^* is a set of exceptions for it. The domain of the target concept is $\{0,1\}^n$.

We assume that we have an upper bound on the cardinality of S^* and denote it by l (i.e., $|S^*| \leq l$). If this bound is not known, we can start out by assuming it to be any positive

integer and doubling it whenever convergence is not achieved within the proper time bound, which will be given later. We assume that h^*_M is minimized and has m terms.

Like LEARNMONDNF, our current algorithm also has a set *CounterExamples* that stores all labeled counterexamples received from equivalence queries. The purpose of it is slightly different: it lets the algorithm conclude that some points cannot be classified by h^*_M alone, and, therefore, have to be included in the exception set.

The algorithm tries to find a suitable monotone DNF formula, which, coupled with a proper exception set, would give the target concept. The equivalence queries are made on a pair $\langle h, S\rangle$ of a monotone DNF formula h and a set of exceptions S. The algorithm focuses only on building h, and sets S to be those elements of the set *CounterExamples* that are currently misclassified by h. It uses a simple subroutine GETEXCEPTIONS for building S. The subroutine is given in Figure 12.

```
GetExceptions(h, CounterExamples)
{
    S = ∅
    For (each ⟨x, b⟩ ∈ CounterExamples)
        If (h(x) ≠ b)
            Add x to S
    Return S
}
```

Figure 12. Subroutine GETEXCEPTIONS. h is the monotone DNF part of the current hypothesis; *CounterExamples* is the set of pairs of counterexamples and their labels seen so far; S is the set of those counterexamples that are misclassified by h.

In order to classify the counterexamples received, the algorithm needs to evaluate the current function *xcpt*(h, S). This is done by another very simple subroutine THEFUNCTION, given in Figure 13.

```
TheFunction(h, S, x)
{
    If (x ∈ S)
        Return 1 − h(x)
    Else
        Return h(x)
}
```

Figure 13. Subroutine THEFUNCTION. h is the monotone DNF part of the current hypothesis; S is the set of exception points for it; x is the point that the current hypothesis is evaluated on.

As in (Angluin, 1988), (Angluin & Slonim, 1994), and Section 4, our algorithm also uses a subroutine REDUCE to move down in the lattice from a positive counterexample. Its goal is to reduce the positive counterexample to some point that can be added as a term to the formula h. Then the new hypothesis would classify the counterexample and possibly some other points as positive. However, this may not always be possible. There can be

overwhelming evidence that the candidate point is just a positive exception and thus should not be added to h. More precisely, if there are more than l negative counterexamples above a term of h, then they all have to be in the exception set, which is then too big. Therefore the current subroutine REDUCE is somewhat more complex and checks whether a point has enough evidence to be an undoubted exception point or not. The subroutine is given in Figure 14.

```
REDUCE(v, CounterExamples)
{
    For (each child w of v)
        If ((MQ(w) == 1)  &&  (|{ y ≥ w : ⟨y,0⟩ ∈ CounterExamples}| ≤ l))
            Return REDUCE(w, CounterExamples)
    Return v
}
```

Figure 14. Subroutine REDUCE. *CounterExamples* is the set of counterexamples and their labels seen so far; l is the bound on the number of exception points, a globally known constant.

The algorithm for learning monotone DNF formulas with at most l exceptions using equivalence queries and membership queries is given in Figure 15.

The algorithm is based on the following ideas. Each positive counterexample is reduced if possible to a new term to be added to the formula, as was explained above. In case this is not possible, the algorithm benefits anyway by storing it in the set *CounterExamples*.

Negative counterexamples imply that there are not as many positive points in the target concept as we thought. Sometimes more exception points are necessary for the hypothesis to be correct. Other times some terms have to be removed from the formula. Deleting a term happens only when there is enough evidence that a term is wrong, namely, when there are more than l negative counterexamples above it.

5.4. *Correctness and Complexity of the Algorithm*

THEOREM 6 LEARNMONDNFWITHFX *learns the class of monotone DNF formulas with exceptions in polynomial time using equivalence and standard membership queries.*

Proof: We begin the analysis with this simple claim.

CLAIM 6 *Once a term t is deleted from hypothesis h, it can never reappear in it.*

Proof: A term t can be deleted only if there are more than l negative counterexamples above it. To reappear, t must be returned by REDUCE. But every point returned by REDUCE must have at most l negative counterexamples above it at the time it is returned, so REDUCE cannot return t again. ■

The following lemma shows what points REDUCE can return.

LEMMA 12 REDUCE *always returns either a local minimum of h^* or a parent of a positive exception in S^*.*

```
LearnMonDNFwithFX()
{
    S = CounterExamples = ∅
    h = "the empty DNF formula"
    While ((v = EQ(⟨h, S⟩)) ≠ "yes")
    {
        Add ⟨v, (1 − TheFunction(h, S, v))⟩ to CounterExamples
        If (TheFunction(h, S, v) == 1)
            For (each term t of h)
                If (|{ w ≥ t : (w, 0) ∈ CounterExamples }| > l)
                    Delete term t from h
        For (each ⟨x, 1⟩ ∈ CounterExamples)
            If ((h(x) == 0)  &&  (|{ y ≥ x : ⟨y, 0⟩ ∈ CounterExamples }| ≤ l))
            {
                w = Reduce(x, CounterExamples)
                Add term w to h
            }
        S = GetExceptions(h, CounterExamples)
    }
    Output ⟨h, S⟩
}
```

Figure 15. The algorithm for learning monotone DNF formulas with finite exceptions. *CounterExamples* is the set of counterexamples and their labels seen so far; l is the bound on the number of exception points, a globally known constant; h is the monotone DNF part of the current hypothesis; S is the set of points in *CounterExamples* misclassified by h.

Proof: First note that REDUCE can only be called on points x such that $h^*(x) = 1$ and can only return points w such that $h^*(w) = 1$. Let w be a point returned by REDUCE. Assume w is not a local minimum point of h^*. Then there is some child y of w such that $h^*(y) = 1$, and the number of negative counterexamples above y must exceed l (or else REDUCE would have been called recursively on y). Hence, y cannot be above any term t of h^*_M, since each term t can have at most l negative counterexamples above it. Therefore, y is a positive exception in S^*. ■

Now we are ready to bound the number of different points that can be returned by the subroutine REDUCE.

CLAIM 7 *The number of different points that* REDUCE *can return is at most* $m + (n+1)l$.

Proof: By Lemma 12, the number of different points that can be returned by REDUCE is at most the number of points that are local minima of h^* or parents of positive exceptions in S^*. Let S^* contain l_p positive exceptions and l_n negative exceptions, where $l_p + l_n \leq l$. The formula h^*_M has m terms and therefore m local minima. By Lemma 8, the number of local minima of h^* is at most $m + l_p + nl_n$. Each positive exception has at most n parents, so the number of parents of positive exceptions is bounded by nl_p. Thus, the number of

different points REDUCE can return, and the number of calls to REDUCE, is bounded by $m + (n+1)l_p + nl_n \leq m + (n+1)l$. ■

All equivalence queries are asked about the current hypothesis *xcpt*(h, S). Since S is computed right before each equivalence query, the argument of an equivalence query is always consistent with all the counterexamples seen to that point. Let h_i and h_j denote the function *xcpt*(h, S) at the time when ith and jth equivalence query is asked, respectively, and let $i < j$. Let v_i be the counterexample returned by the ith equivalence query. Clearly, the values of $h_i(v_i)$ and $h_j(v_i)$ must be different. Thus, the function *xcpt*(h, S) is different for each equivalence query. This allows us to bound the total number of equivalence queries.

CLAIM 8 *The number of equivalence queries before success is bounded by* $O(m^2n^2l^3)$.

Proof: We examine how *xcpt*(h, S) changes. Either h itself changes, or h remains the same and S changes; namely, it contains exactly one more point, the most recent counterexample.

By Claim 6, each term of h can appear in h or disappear from it only once. Thus each possible term can induce at most two changes in formula h—first by appearing in it and then by disappearing. Thus, h can only change twice as many times as the number of terms that REDUCE can return. Therefore, by Claim 7, there can be at most $2(m + (n+1)l) + 1$ different functions h in a complete run of the algorithm.

We now count the number of times S can change while h remains the same. Set S grows larger by one with each new counterexample. It contains some (possibly none) points x such that $h(x) = 1$ and some (possibly none) points x such that $h(x) = 0$. We bound the number of each of these separately.

Each point $x \in S$ such that $h(x) = 1$ is above some term of h. No term can have more than l negative counterexamples above it. Therefore, the number of points $x \in S$ such that $h(x) = 1$ can be bounded by l times the bound $m + (n+1)l$ on the number of different terms of h, that is, by $ml + (n+1)l^2$.

Each point $x \in S$ such that $h(x) = 0$ is a positive counterexample, and thus is not above any term in h. Such an x must have more than l negative counterexamples above it. Otherwise, the algorithm would have called REDUCE on x and added a new term $t \leq x$ to h. If x has more than l negative counterexamples above it, then it cannot be above a term in h^*_M and thus has to be a positive exception in S^*. Hence we have a bound of l_p on the number of points $x \in S$ such that $h(x) = 0$.

Altogether, we can bound the cardinality of S by $|S| \leq ml + (n+1)l^2 + l_p \leq (m + 1)l + (n+1)l^2$. While h stays the same, the number of possible different sets S is at most $(m+1)l + (n+1)l^2 + 1$.

Hence, the total number of equivalence queries in a complete run of the algorithm is bounded by $(2(m + (n+1)l) + 1) \times ((m+1)l + (n+1)l^2 + 1) = O(m^2n^2l^3)$. ■

We now count the total number of membership queries. Membership queries are made only in REDUCE, at most $n(n+1)/2$ per call to REDUCE. Claim 7 bounds the number of different points that REDUCE can return by $m + (n+1)l$. By Claim 6, the number of calls to REDUCE is bounded by the number of different points that it can return. Therefore, the total number of membership queries is bounded by $O(mn^2 + n^3l)$.

It is not difficult to see that the total running time of the algorithm is polynomial in n, m and l. This concludes the proof of Theorem 6. ■

6. Comparison of the Models

In this section, we compare the models of learning discussed earlier, and give a relation between learning concepts with exceptions and learning with malicious membership queries.

6.1. Exceptions and Lies

In this subsection, we give a generic algorithm transformation. This transformation shows that any class of concepts that is polynomially closed under finite exceptions and learnable in polynomial time with equivalence and standard membership queries is also learnable in polynomial time using equivalence and malicious membership queries.

THEOREM 7 *Let H be a class of concepts that is polynomially closed under finite exceptions and learnable in polynomial time with equivalence and standard membership queries. Then H is learnable in polynomial time with equivalence and malicious membership queries.*

Proof: Let $H = (R, \mathit{Dom}, \mu)$ be a target class of concepts that is polynomially closed under finite exceptions. We assume that LEARN is an algorithm to learn H using equivalence (EQ) and standard membership queries (MQ) in time $p_A(s, n)$, for some polynomial p_A. Without loss of generality, p_A is non-decreasing in both arguments. We transform this algorithm into algorithm LEARNANYWAY, which learns any concept $h^* \in H$ using equivalence and malicious membership queries in time polynomial in $|h^*|$, n and the table-size L of the set of strings on which MMQ may lie.

As in Sections 4 and 5.3 the main idea is to keep track of all the counterexamples seen and to use them to avoid unnecessary membership queries. For this purpose we use a set *CounterExamples* again. As before it stores pairs of counterexamples and their labels. Now, before asking a membership query about string x, we scan *CounterExamples* to see whether it already contains x and a label for it. If x and the label are found, the algorithm knows the answer and does not make the query. (For some concept classes, such as monotone DNF formulas, it might be possible to infer the classification of x according to the target concept h^* even though x and its label are not contained in *CounterExamples*. However, this simple checking suffices for our algorithm and, what is more important, works in the general case.)

Another idea is to keep track of the answers received from membership queries, and to use them to conclude that MMQ has lied. For this purpose LEARNANYWAY has a set *MembershipAnswers*. This set stores pairs $\langle x, b\rangle$ for which MMQ was called on string x and returned answer b. After receiving a new counterexample from EQ, the algorithm stores it in *CounterExamples* and checks whether this counterexample is already contained in *MembershipAnswers*. If it is present in *MembershipAnswers* with the wrong label, the algorithm discards everything except the set *CounterExamples* and starts from scratch. If this is not the case, the algorithm continues the simulation of LEARN, which we now describe in detail.

```
NewMQ(x, CounterExamples, MembershipAnswers)
{
    If (⟨x, b⟩ ∈ CounterExamples)
        Return b
    b = MMQ(x)
    Add ⟨x, b⟩ to MembershipAnswers
    Return b
}
```

Figure 16. Subroutine NEWMQ. *CounterExamples* is the set of counterexamples and their labels seen so far; *MembershipAnswers* is the set of points queried using MMQ and the corresponding answers.

The new algorithm simulates LEARN on the target concept, but modifies LEARN's queries as follows:

- Each membership query $\mathrm{MQ}(x)$ of algorithm LEARN is replaced by a subroutine call $\mathrm{NEWMQ}(x, \textit{CounterExamples}, \textit{MembershipAnswers})$. The subroutine is given in Figure 16.
- Each equivalence query of LEARN, $x = \mathrm{EQ}(h)$, as well as the output statement, **Output** h, is replaced by the block of code given in Figure 17.

```
{
    x = EQ(h)
    If (x == "yes")
    {
        Output h
        Return
    }
    Add ⟨x, (1 − h(x))⟩ to CounterExamples
    If (⟨x, h(x)⟩ ∈ MembershipAnswers)
    {
        MembershipAnswers = ∅
        Restart Simulation, retaining CounterExamples
    }
}
```

Figure 17. The block of code replacing "$x = \mathrm{EQ}(h)$" or "**Output** h". h is the current hypothesis; x is the current counterexample; *CounterExamples* is the set of counterexamples and their labels seen so far; *MembershipAnswers* is the set of points queried using MMQ and the corresponding answers.

Note that when the simulation is restarted, only the set *CounterExamples* reflects any work done so far. We now show that LEARNANYWAY is correct and runs in time polynomial in $|h^*|$, n, and L. We partition the run of the algorithm into *stages*, where a stage begins with a new simulation of LEARN. First we show that a stage cannot last forever.

CLAIM 9 *Every stage ends in time polynomial in* $|h^*|$, n, *and* L.

Proof: Note that H is polynomially closed under finite exceptions, which means that there is a polynomial $p(\cdot,\cdot)$ such that for every concept $h \in H$ and every finite set $S \subseteq \mathit{Dom}(h)$ there exists a concept $h' \in H$ equal to $\mathit{xcpt}(h, S)$ such that size $|h'| \leq p(|h|, ||S||)$. Without loss of generality we can assume that p is non-decreasing in both arguments. We now prove that each stage ends in time bounded by $p_A(p(|h^*|, L), n)$, where we count only the time spent on LEARN operations (i.e., we do not count the simulation and bookkeeping overhead).

We prove this by contradiction. Assume that stage i goes over the limit. Let us look at the situation right after the number of simulated steps of LEARN exceeds our stated time bound. Let S_i denote the set of strings the MMQ has lied about during this stage, up to the time bound. Let n denote the length of the longest counterexample received during this stage, up to the time bound.

None of the strings in S_i can belong to *CounterExamples*. Assume by way of contradiction otherwise. Let $x \in S_i$ be a string contained in *CounterExamples* with some label. Set S_i contains exactly the strings that the MMQ lied on in this stage and time bound, so there was a query $\mathrm{MMQ}(x)$. It must have happened before x was added to *CounterExamples*. But then at the moment x was added to *CounterExamples* it already belonged to *MembershipAnswers* and an inconsistency had to be found. The stage had to end.

Therefore, considering S_i as an exception set, all the information received by LEARN in this stage and within the given time bound is consistent with the concept $h' = \mathit{xcpt}(h^*, S_i) \in H$. LEARN either has to output h' in time bounded by

$$p_L\big(p(|h^*|, ||S_i||), n\big) \leq p_L\big(p(|h^*|, L), n\big),$$

or it has to receive a counterexample $x \in S_i$. In the former case, LEARNANYWAY makes an equivalence query $\mathrm{EQ}(h')$ and receives a counterexample $x \in S_i$, since only counterexamples from S_i are possible at that point. In either case, an element of S_i is added to *CounterExamples* by the above time bound, which we showed above was impossible. This is a contradiction to the assumption that stage i goes over this bound. ■

What remains is to show that there can be only a small number of stages. That is, we do not restart the simulation too many times.

CLAIM 10 *There are at most* $L + 1$ *stages in the run of the algorithm* LEARNANYWAY.

Proof: At the beginning of each stage (except the first one) the algorithm discovers a new string where the MMQ lies and from then on MMQ can never lie on this string again, because it is added to *CounterExamples*. To be more precise, MMQ does not get a chance to lie on this string because it is never asked about it again. Let S be the set of the strings that MMQ lies on. Since $|S| \leq ||S|| \leq L$, in stage $L + 1$ the MMQ can lie on no strings (i.e., it is not asked queries about any of the strings where it may lie). Therefore LEARN has to converge to the target concept h^*. ■

The time spent on simulation and bookkeeping is clearly polynomial in $|h^*|$, n, and L. Thus, LEARNANYWAY is a polynomial-time algorithm that uses equivalence and malicious

membership queries to learn the class of concepts $H = (R, Dom, \mu)$. This concludes the proof of Theorem 7. ■

As corollaries of Theorem 7 we have the following.

COROLLARY 5 *The class of regular languages, represented by DFA's, is learnable in polynomial time with equivalence and malicious membership queries.*

Proof: In (Board & Pitt, 1992) it was shown that this class of concepts is polynomially closed under finite exceptions. In (Angluin, 1987) it was shown that it is learnable in polynomial time using membership and equivalence queries. ■

COROLLARY 6 *The class of boolean decision trees is learnable in polynomial time with extended equivalence and malicious membership queries.*

Proof: Lemma 9 shows that the class of boolean decision trees is polynomially closed under finite exceptions. In (Bshouty, 1993) it was shown that it is learnable in polynomial time using membership and extended equivalence queries. ■

COROLLARY 7 *The class of monotone DNF formulas with finite exceptions is learnable in polynomial time with equivalence and malicious membership queries.*

Proof: Corollary 4 shows that the class of monotone DNF formulas with exceptions is polynomially closed under finite exceptions. In Section 5.3 we gave an algorithm that learns this class in polynomial time with membership and equivalence queries. ■

Note that we can also learn the class of monotone DNF formulas without any exceptions with this generic algorithm, using extended equivalence and malicious membership queries, since it is just a subclass of the class that allows exceptions. However, the algorithm is much less efficient than the one described in Section 4.

6.2. Learning with and without "Don't knows"

In this subsection, we digress from exceptions and malicious membership queries, and focus again on limited membership queries and standard membership queries. We present a lower bound result, the proof of which has ideas useful in further subsections.

We start by briefly describing a method for converting any algorithm for exact identification from membership and equivalence queries to one that works for limited membership queries. We can also show that in some cases an exponential blowup in the number of queries is necessary.

THEOREM 8 *Every concept class that is learnable with m equivalence and membership queries is learnable with $2^{\ell}(m-\ell+1)+\ell-1$ equivalence and limited membership queries, where ℓ is the number of $\perp$ responses received.*

Proof: Let Algorithm A exactly identify concept class $\mathcal{C}$ from at most m equivalence and membership queries. We construct a learning algorithm A' for equivalence and limited

membership queries as follows: For each instance x such that $\mathrm{LMQ}(x) = \perp$, start running two copies of A in parallel, one assuming x is positive and the other assuming that x is negative. Furthermore, store all the queries and their answers in a global table, and do not repeat a query that has already been made (possibly by another copy of A). For each copy of A, if the answers that it has seen (including the guesses for the $\perp$ answers) are consistent with some concept from $\mathcal{C}$, then it must output a final hypothesis after at most m queries (including the $\perp$ ones). Those copies that have answers inconsistent with any concept from $\mathcal{C}$ may be stopped; this will take at most m queries. For those copies that do obtain the final hypothesis after at most m queries (except for one, possibly), we may have to ask the final equivalence query to see which one of them has the correct answer. But, obviously, some copy of A will make the correct guesses for the $\perp$ answers and therefore it will have a correct final hypothesis after at most m queries.

The exact bound can be proven by induction on the number ℓ of $\perp$ answers out of the total m of EQ's and LMQ's. The proof is easier if we think about the computation of A' as a tree. Every query that A' makes is a node in the tree. Each node is labeled by the query made. Every $\perp$ answer is a *branching node* (i.e., such nodes have two children, one that assumes the answer is 0 and the other that assumes it is 1). There is no branching on equivalence queries or LMQ's that return a 0 or a 1. All paths from the root to the leaves have at most $m + 1$ nodes on them (for the sake of simplicity, we will assume that a final EQ is made, and allow for this in the formula). Of course, on each path there are at most ℓ branching nodes. Furthermore, on every root-to-leaf path, the labels of the branching nodes (i.e., queries made) are all distinct.

We need to bound the total number of nodes in the tree, but for the branching nodes we need count only how many different labels they have (since no query is repeated). That is, we basically have to count the non-branching nodes and add ℓ.

For convenience let us name these trees. If such a tree as described above has ℓ branching nodes and at most $m + 1$ nodes on each path from the root to the leaves, we call it an *ℓ-m-branching-tree*. (Of course, $\ell \leq m$ for every valid ℓ-m-branching-tree.) We call ℓ plus the number of the tree's non-branching nodes the *labeled-node count*, since if all the non-branching nodes had different labels (which they may, if the tree corresponds to a computation of A' and the labels are given with respect to the LMQ's or EQ's being done in A'), then this would really be just the count of the labels. We now begin the inductive proof that the labeled-node count of any ℓ-m-branching-tree does not exceed $2^{\ell}(m - \ell + 1) + \ell$.

Base case, $\ell = 0$, $m \geq 0$: If there are no branching nodes then there is only one path from the root to the leaf in the tree, and since its length is bounded by $m + 1$ the bound holds.

Inductive assumption: Assume that for all $\ell' \leq \ell$ and for all $m \geq \ell'$ we have proved the bound. That is, the labeled-node count of every ℓ'-m-branching-tree (where $\ell' \leq \ell$) is bounded by $2^{\ell'}(m - \ell' + 1) + \ell'$.

Induction step: Now we prove the bound for $\ell + 1$ and every $m \geq \ell + 1$. That is, we take an arbitrary $(\ell+1)$-m-branching-tree. We start from the root of the tree and follow down the only path until we reach the first branching node b. Let the number of nodes from the root to b, inclusively, be m^*. The left subtree of b is an ℓ_0-$(m-m^*)$-branching-tree, for some $\ell_0 \leq \ell$. The right subtree of b is an ℓ_1-$(m - m^*)$-branching-tree, for

some $\ell_1 \leq \ell$. We need to further elaborate on the labels of the branching nodes in these subtrees. None of these labels are the same as the label for b, since on every path all labels have to be different. Let ℓ_0^* be the number of branching nodes that have labels not occurring in the right subtree and let ℓ_1^* be the number of branching nodes that have labels not occurring in the left subtree. Let ℓ^* be the labels that exist in both subtrees of b. Obviously, we know that $\ell_0 = \ell_0^* + \ell^*$ and that $\ell_1 = \ell_1^* + \ell^*$. We also know that $\ell_0^* + \ell_1^* + \ell^* = \ell$.

The labeled-node count of the original $(\ell+1)$-m-branching-tree can be expressed as the labeled-node count of the left subtree of b, plus the labeled-node count of the right subtree of b, plus m^*, and minus ℓ^*, the number of branching nodes that have been counted in the labeled-node count of both subtrees.

If we use the inductive assumption for the labeled-node counts of the left and right subtrees, we just have to verify that

$$\begin{aligned} &2^{\ell_0^*+\ell^*}\big((m-m^*)-(\ell_0^*+\ell^*)+1\big)+(\ell_0^*+\ell^*) \\ &+2^{\ell_1^*+\ell^*}\big((m-m^*)-(\ell_1^*+\ell^*)+1\big)+(\ell_1^*+\ell^*)+m^*-\ell^* \\ &\qquad\qquad\leq 2^{\ell+1}\big(m-(\ell+1)+1\big)+(\ell+1). \end{aligned}$$

Some simplifications on the left side of this inequality, lead us to

$$\begin{aligned} &2^{\ell-\ell_1^*}\big(m-(\ell-\ell_1^*)+1\big) \;+\; 2^{\ell-\ell_0^*}(m-(\ell-\ell_0^*)+1)+\ell \\ &\qquad -\; m^*(2^{\ell-\ell_1^*}+2^{\ell-\ell_0^*}-1), \end{aligned}$$

and we have to verify that it does not exceed $2^{\ell+1}(m-\ell)+\ell+1$. We can increase the left side by taking m^* as small as possible, namely 1. Therefore, we now only need to show that

$$2^{\ell-\ell_1^*}\big(m-(\ell-\ell_1^*)\big)+2^{\ell-\ell_0^*}\big(m-(\ell-\ell_0^*)\big)\leq 2^{\ell+1}(m-\ell).$$

It is easy to prove that $2^{\ell-k}\big(m-(\ell-k)\big) \leq 2^{\ell}(m-\ell)$, if $k \leq \ell \leq m$. Since $\ell_1^* \leq \ell \leq m$ and $\ell_0^* \leq \ell \leq m$, we can apply the above formula to both terms of the left side of the inequality that we are trying to prove. This completes the inductive proof.

The only difference between the number of queries A' makes and the labeled-node count of a ℓ-m-branching-tree that corresponds to its computation is that we can save the last equivalence query for one of the copies of A. This concludes the proof. ■

The next theorem shows that in some cases such an exponential blowup in the number of queries is in fact necessary.

THEOREM 9 *There is a concept class learnable with m equivalence and membership queries that requires $2^{\ell}(m-\ell+1)-1$ equivalence and limited membership queries, where ℓ is the number of $\perp$ responses received.*

Proof: We construct a concept class $\mathcal{C}$ that is a variant of ADDRESSING (Maass & Turán, 1992). Let the instance space be $X = \bigcup_{i=0}^{2^{\ell}} X_i$, where the X_i's are disjoint, $X_0 =$

$\{1, \ldots, \ell\}$, and $|X_i| = m - \ell + 1$ for $1 \leq i \leq 2^\ell$. Since $|X_0| = \ell$, each of its subsets can be viewed as an ℓ-bit number. A set $c \subseteq X$ is in $\mathcal{C}$ if and only if it has the following form. It contains exactly one element x that is not in X_0, and if i denotes the number represented by $c \cap X_0$, then that x is in X_i.

Concept class $\mathcal{C}$ can be learned by ℓ membership queries for the elements in X_0 followed by $m - \ell$ membership queries for the elements of X_i, where i is the number represented by the responses obtained in the first phase.

On the other hand, the following adversary strategy shows that at least $2^\ell(m - \ell + 1) - 1$ equivalence and limited membership queries are required to learn $\mathcal{C}$. The limited membership oracle responds $\perp$ to all instances in X_0. Membership queries for other elements are answered by "No." Equivalence queries are answered by providing a negative counterexample outside X_0. ∎

Taking $\ell = m$ in the proof of Theorem 9 gives us the original concept class ADDRESSING, and an example where m $\perp$ responses increase the number of queries required from m for ordinary membership queries to $2^m - 1$ for limited membership and equivalence queries.

Note that ADDRESSING also causes the incomplete membership query model (Angluin & Slonim, 1994) to have an expected exponential blowup over ordinary membership queries when the probability of a $\perp$ response is a constant. For constant probability p of $\perp$, the expected number of instances in X_0 answered $\perp$ is pm. This will increase the number of queries required from m for ordinary membership queries to 2^{pm} for incomplete membership and equivalence queries.

If, instead of allowing equivalence queries only from the concept class, one allows extended equivalence queries with *any* set, then such a blowup cannot occur. This follows from a result of Auer and Long (1994) showing that in this model membership queries can speed up learning by only a constant factor.

6.3. *Strict versus Nonstrict*

Recall that in the nonstrict model the final hypothesis need only agree with the target concept on points x such that $\mathrm{LMQ}(x) \neq \perp$, while in the strict model, they must be exactly equal.

Every learning algorithm that works in the strict model can be run in the nonstrict model without increasing its complexity. A relationship in the other direction can be established by a method similar to the one used in the proof of Theorem 7 in subsection 6.1.

Every learning algorithm that works in the nonstrict model can be adapted to work in the strict model as follows. First note that the problem that may occur when running a nonstrict algorithm in the strict model is that it may receive as a counterexample to an equivalence query a point that was previously classified as a "Don't know" in a membership query. In this case, the execution of the algorithm is interrupted. The algorithm is then restarted, remembering the point and its classification for possible later use in answering a membership query. Since each interruption corresponds to a new "Don't know," this simulation essentially adds a multiplicative factor of ℓ to the complexity of the learning algorithm.

Hence, from the point of view of polynomial learnability, the strict and nonstrict models using EQ's and LMQ's are equivalent.

6.4. *Lies versus Omissions*

As noted in Subsections 2.2 and 3.4, learning with MMQ's and EQ's is at least as difficult as with LMQ's and EQ's. To show that learning with MMQ's and EQ's is in fact more difficult, we construct yet another variant of ADDRESSING parameterized by m and ℓ as follows. The universe consists of a set X_0 of m elements, and a disjoint set X_1 of $\binom{m}{\ell}$ elements. We choose some fixed one-to-one correspondence between the elements in X_1 and subsets of X_0 of cardinality ℓ. The desired class contains concepts Y consisting of a subset of X_0 of cardinality ℓ together with the corresponding element of X_1. This concept class can be learned using m LMQ's followed by at most 2^ℓ EQ's. On the other hand, consider an adversary that answers MMQ's with 0, and EQ's with the element of the queried concept from X_1 as the counterexample. Then MMQ's convey no information, and EQ's eliminate concepts one at a time, so at least $\binom{m}{\ell} - 1$ MMQ's and EQ's are required to learn this concept class.

This example, with $\ell = \log m$, shows that the number of MMQ's and EQ's necessary to learn a concept class cannot be bounded by any polynomial in the number of LMQ's and EQ's. On the other hand, since in this example the number of LMQ's and EQ's is exponential in ℓ, it does not answer the question "Are there any concept classes polynomially learnable from EQ's and LMQ's that are not polynomially learnable from EQ's and MMQ's?"

7. Summary

Most of the results proven in this paper are summarized in Table 1. The remaining ones are given below.

1. Strict and nonstrict models of learning from equivalence and limited membership queries are polynomial-time equivalent. (Subsection 6.3.)

2. Polynomial-time learnability from equivalence and malicious membership queries implies polynomial-time learnability from equivalence and limited membership queries. (Subsection 2.2.)

3. Learning monotone monomials in the nonstrict model from limited membership queries alone may require $\Omega(n^{c+1})$ queries when $O(n^c)$ omissions are given. (Corollary 1.)

4. Any class of concepts that is polynomially closed under finite exceptions and is learnable in polynomial time from equivalence and standard membership queries is also learnable in polynomial time from equivalence and malicious membership queries. (Theorem 7.)

5. Every concept class that is learnable from m equivalence and standard membership queries is learnable in the strict model from $2^\ell(m - \ell + 1) + \ell - 1$ equivalence and limited membership queries. (Theorem 8.)

Table 1. Summary of the results for various boolean formulas: n denotes the number of variables; m denotes the number of terms in a formula; s denotes the size of a concept; ℓ denotes the number of lies or omissions; l denotes the number of exceptions.

Class of Concepts	Type of MQ's	Number of EQ's	Number of MQ's
Monotone Monomials (Theorem 1)	LMQ's		$n\ell + n + 1$
Monotone k-term DNF (Theorem 3)	LMQ's		$O(kn^k + n^2\ell)$
Monotone DNF, Nonstrict (Theorem 4)	LMQ's	$m + 1$	$n(m + \ell)$
Monotone DNF, Strict (Theorem 4)	LMQ's	$m + \ell + 1$	$n(m + \ell)$
Monotone DNF (Theorem 5)	MMQ's	$O(m + n\ell)$	$O(mn^2 + \ell n^3)$
Monotone DNF with Finite Exceptions (Theorem 6)	MQ's	$O(m^2n^2l^3)$	$O(mn^2 + ln^3)$
DFA's (Corollary 5)	MMQ's	$poly(s, n, \ell)$	$poly(s, n, \ell)$
Decision Trees (with Extended EQ's, Corollary 6)	MMQ's	$poly(n, \ell)$	$poly(n, \ell)$
Monotone DNF with Finite Exceptions (Corollary 7)	MMQ's	$poly(n, m, l, \ell)$	$poly(n, m, l, \ell)$

6. There exists a concept class learnable with m equivalence and standard membership queries that requires $2^\ell(m - \ell + 1) - 1$ equivalence and limited membership queries to be learned in the strict model. (Theorem 9.)

7. There exists a concept class learnable in the strict model with $m + 2^\ell$ equivalence and limited membership queries that requires $\binom{m}{\ell} - 1$ equivalence and malicious membership queries to be learned. (Subsection 6.4.)

8. Discussion and Open Problems

As noted in the introduction, there are many classes of concepts that are efficiently learnable with membership and equivalence queries. For some of them we now have learning algorithms that use equivalence and limited or malicious membership queries. Many other problems still remain unexplored. For example, there is not yet any algorithm for learning read-once formulas from equivalence and limited or malicious membership queries, even though there is an algorithm for learning read-once formulas from equivalence and standard membership queries. A start in this direction is made in (Angluin, 1994), which gives a randomized polynomial-time algorithm to learn μ-DNF formulas with equivalence and malicious membership queries. In the model of PAC learning with membership queries, it would be interesting to see whether Baum's algorithm (Baum, 1991) can be modified to tolerate "I don't know" answers.

Another type of open problem is finding lower bounds for any of the classes of concepts for which we give learning algorithms using equivalence and limited or malicious membership queries. So far, we have lower bounds for only a specially constructed class and for monotone monomials in the model that uses only limited membership queries. For other

classes, the following question is still relevant: can we prove something stronger than the trivial bound that there must be more membership queries than lies or omissions?

Moving on to the comparison of models, we have two very intriguing questions. The first one is, are there any classes of concepts that are polynomial-time learnable from equivalence and limited membership queries, but not polynomial-time learnable from equivalence and malicious membership queries? The second question is, are learning with exceptions and learning with lies equally difficult for classes that are polynomially closed under finite exceptions, or is learning with exceptions more difficult for these classes? That is, is there a class that is polynomially closed under finite exceptions, is learnable with malicious membership queries in polynomial time, and is not polynomial-time learnable with exceptions? We also do not know how the difficulty of learning with exceptions classes that are not polynomially closed under finite exceptions relates to learning such classes with malicious membership queries.

A less important extension of this work would be to improve the time bound for the algorithm that learns monotone DNF formulas with exceptions and possibly for the one that learns monotone DNF from equivalence and malicious membership queries.

Acknowledgments

This research was funded in part by the National Science Foundation, under grants CCR-9213881, CCR-9108753, CCR-9314258 and CCR-9208170, and by Esprit BRA ILP, Project 6020, OTKA grant T-014228, and OTKA grant T-016349.

We thank the referees for their careful reading and helpful suggestions. This work appeared as two separate papers in the Proceedings of the 7th Annual ACM Conference on Computational Learning Theory (Angluin & Kriķis, 1994b), (Sloan & Turán, 1994). Part of it is also available as a technical report (Angluin & Kriķis, 1994a).

References

Anderson, J. R. (1980). *Cognitive Psychology and Its Implications*. W. H. Freeman and Company.

Angluin, D. (1987). Learning regular sets from queries and counterexamples. *Inform. Comput.*, 75(2):87–106.

Angluin, D. (1988). Queries and concept learning. *Machine Learning*, 2(4):319–342.

Angluin, D. (1994). Exact learning of μ-DNF formulas with malicious membership queries. Technical Report YALEU/DCS/TR-1020, Yale University Department of Computer Science.

Angluin, D., & Kriķis, M. (1994a). Malicious membership queries and exceptions. Technical Report YALEU/-DCS/TR-1019, Yale University Department of Computer Science.

Angluin, D., & Kriķis, M. (1994b). Learning with malicious membership queries and exceptions. In *Proc. 7th Annu. ACM Workshop on Comput. Learning Theory*, pages 57–66. ACM Press, New York, NY.

Angluin, D., & Slonim, D. K. (1994). Randomly fallible teachers: learning monotone DNF with an incomplete membership oracle. *Machine Learning*, 14(1):7–26.

Auer, P., & Long, P. M. (1994). Simulating access to hidden information while learning. In *Proc. of the 26th Annual ACM Symposium on Theory of Computing*, pages 263–272. ACM Press, New York, NY.

Baum, E. (1991). Neural net algorithms that learn in polynomial time from examples and queries. *IEEE Transactions on Neural Networks*, 2:5–19.

Board, R., & Pitt, L. (1992). On the necessity of Occam algorithms. *Theoret. Comput. Sci.*, 100:157–184.

Bshouty, N. H., (1993). Exact learning via the monotone theory. In *Proceedings of the 34th Annual Symposium on Foundations of Computer Science*, pages 302–311. IEEE Computer Society Press, Los Alamitos, CA.

Bultman, W. J. (1991). *Topics in the Theory of Machine Learning and Neural Computing*. PhD thesis, University of Illinois at Chicago Mathematics Department.

Dean, T., Angluin, D., Basye, K., Engelson, S., Kaelbling, L., Kokkevis, E., & Maron, O. (1995). Learning finite automata with stochastic output functions and an application to map learning. *Machine Learning*, 18(1):81–108.

Frazier, M., Goldman, S., Mishra, N., & Pitt, L. (1994). Learning from a consistently ignorant teacher. In *Proc. 7th Annu. ACM Workshop on Comput. Learning Theory*, pages 328–339. ACM Press, New York, NY.

Goldman, S. A., Kearns, M. J., & Schapire, R. E. (1993). Exact identification of read-once formulas using fixed points of amplification functions. *SIAM J. Comput.*, 22(4):705–726.

Goldman, S. A., & Mathias, H. D. (1992). Learning k-term DNF formulas with an incomplete membership oracle. In *Proc. 5th Annu. Workshop on Comput. Learning Theory*, pages 77–84. ACM Press, New York, NY.

Kearns, M. (1993). Efficient noise-tolerant learning from statistical queries. In *Proc. 25th Annu. ACM Sympos. Theory Comput.*, pages 392–401. ACM Press, New York, NY.

Kushilevitz, E., & Mansour, Y. (1993). Learning decision trees using the Fourier spectrum. *SIAM J. Comput.*, 22(6):1331–1348. Earlier version appeared in STOC 1991.

Lang, K. J., & Baum, E. B. (1992). Query learning can work poorly when a human oracle is used. In *International Joint Conference on Neural Networks*, Beijing.

Maass, W., & Turán, G. (1992). Lower bound methods and separation results for on-line learning models. *Machine Learning*, 9:107–145.

Ron, D., & Rubinfeld, R. (1995). Learning fallible deterministic finite automata. *Machine Learning*, 18(2/3):149–185.

Sakakibara, K. (1991). On learning from queries and counterexamples in the presence of noise. *Inform. Proc. Lett.*, 37(5):279–284.

Sloan, R. H., & Turán G. (1994). Learning with queries but incomplete information. In *Proc. 7th Annu. ACM Workshop on Comput. Learning Theory*, pages 237–245. ACM Press, New York, NY.

Valiant, L. G. (1985). Learning disjunctions of conjunctions. In *Proceedings of the 9th International Joint Conference on Artificial Intelligence, vol. 1*, pages 560–566, Los Angeles, California. International Joint Committee for Artificial Intelligence.

Zhuravlev, Y., & Kogan, Y. (1985). Realization of boolean functions with a small number of zeros by disjunctive normal forms, and related problems. *Soviet Math. Doklady*, 32:771–775.

Received May 9, 1995
Accepted November 20, 1995
Final Manuscript April 23, 1997