Using Queries to Identify μ-formulas

Dana Angluin*, Yale University

YALE/DCS/RR-694
March 1989

*Supported by the National Science Foundation, IRI-8718975

Using queries to identify μ-formulas

Dana Angluin [*]
Yale University

March 1989

Abstract

We consider μ-formulas, that is, boolean formulas with at most one occurrence of each variable, and the types of queries needed to identify them exactly in polynomial time. We describe a polynomial time algorithm to learn μ-formulas in disjunctive or conjunctive normal form using equivalence and membership queries. We also describe a polynomial time algorithm to learn monotone μ-formulas using just membership queries. For general (non-monotone) μ-formulas, we show that membership and "relevant possibility" queries suffice for polynomial time identification. These results are compared with those of Valiant[5] on learning μ-formulas with queries. Finally, we show that no polynomial time algorithm can exactly identify μ-formulas using only equivalence queries, even if the target formula is known to be monotone and in disjunctive normal form.

1 Definitions

1.1 Boolean vectors

The domain of boolean values is $B_1 = \{0, 1\}$. Define $\overline{0} = 1$ and $\overline{1} = 0$. Also, $\wedge$, $\vee$, and $\oplus$ denote the binary operations "and", "or", and "exclusive or", respectively, on the domain B_1.

Let n be a positive integer. Let B_n denote the set of all boolean n-vectors, that is, n-tuples of 0's and 1's. Let x and y be elements of B_n. We denote by $\overline{x}$ the vector z such that $z_i = \overline{x_i}$ for $i = 1, \ldots, n$. Also, $x \wedge y$ denotes the vector z such that $z_i = x_i \wedge y_i$ for $i = 1, \ldots, n$. Similary, $x \vee y$ and $x \oplus y$ are defined componentwise. We denote by $d(x, y)$ the Hamming distance between x and y, that is, the number of 1's in the vector $x \oplus y$. We define $x \leq y$ if and only if $x_i \leq y_i$ for all $i = 1, \ldots, n$.

If L is any subset of $\{X_1, \ldots, X_n\}$, let π_L denote the boolean n-vector x such that $x_i = 1$ if and only if $X_i \in L$. Finally, for each $i = 1, \ldots, n$, we define π_i to be the boolean n-vector $\pi_{\{i\}}$, that is, the vector that is 1 in position i and 0 elsewhere.

1.2 AND/OR trees

A *tree* consists of a nonempty finite set of vertices V and a set $E \subseteq V \times V$ of directed edges, with the following properties. There is one vertex, the *root*, of in-degree zero; every other

[*]Supported by NSF grant IRI-8718975

node has in-degree one. There is a unique directed path from the root to each vertex of the graph. Vertices with out-degree zero are *leaves*; other vertices are *internal vertices*. If v is an internal vertex, then each vertex w such that $(v, w) \in E$ is called a *child* of v, and v is called the *parent* of w.

Let T be a tree with vertices V and edges E. A *subtree* of T is any tree whose vertices are a subset of V and whose edges are a subset of E. If $v \in V$, then *the subtree of T rooted at v* is the tree with vertices V' and edges E' such that $v' \in V'$ if and only if there is a path from v to v' in T and $E' = E \cap (V' \times V')$. Note that v is the root of the subtree of T rooted at v. If L is a nonempty set of leaves of a tree T with vertices V and edges E then the *subtree leaf-induced by L* is the tree with vertices V' and edges E' such that V' is the set of all vertices $v \in V$ such that v occurs on some path from the root to an element of L in T and $E' = E \cap (V' \times V')$.

An AND/OR tree is a tree such that every internal vertex is labelled either AND or OR, and no vertex has the same label as its parent. An *assignment* to an AND/OR tree T is a function a that maps the leaves of the tree to $\{0, 1\}$. An assignment a to T can be extended to map all the vertices of T to $\{0, 1\}$ by the usual inductive rule. That is, if all the children of an AND node are assigned 1, the AND node is assigned 1, otherwise it is assigned 0, and if all the children of an OR node are assigned 0 then the OR node is assigned 0, otherwise it is assigned 1. We denote by $a(T)$ the value assigned to the root of T by this extension of the assignment a.

Let T be an AND/OR tree. A *positive witness* for T is a nonempty set L of leaves of T such that the subtree T' of T leaf-induced by L has the following properties. If v is a vertex of T' labelled OR then at least one of the children of v in T is also in T'. If v is a vertex of T' labelled AND then all of the children of v in T are also in T'. A *minimal positive witness* for T is a positive witness L for T such that no proper subset of L is also a positive witness for T.

It is not difficult to see that L is a minimal positive witness for T provided L is a positive witness for T and the subtree T' leaf-induced by L is such that for every vertex v of T' labelled OR, exactly one of the children of v in T is also in T'.

We need also the dual concept of a *negative witness* for T, which is a nonempty set L of leaves of T such that the subtree T' of T leaf-induced by L has the following properties. If v is a vertex of T' labelled OR then all of the children of v in T are also in T'. If v is a vertex of T' labelled AND then at least one of the children of v in T is also in T'. L is a *minimal negative witness* for T provided L is a negative witness for T and no proper subset of L is also a negative witness for T. L is a minimal negative witness for T provided the subtree T' leaf-induced by L is such that for every vertex v of T' labelled AND, exactly one child of v in T is also in T'. The choice of terminology is justified by the following fact.

Lemma 1 *Let T be and AND/OR tree, L_1 any positive witness for T, and L_0 any negative witness for T. Let a be any assignment to the leaves of T. If $a(v) = 1$ for all $v \in L_1$ then $a(T) = 1$. If $a(v) = 0$ for all $v \in L_0$ then $a(T) = 0$.*

Proof. In the first case we inductively show that all the nodes in the subtree leaf-induced by L_1 are assigned 1, including the root. Similarly, in the second case we inductively show that all the nodes in the subtree leaf-induced by L_0 are assigned 0, including the root. Q.E.D.

1.3 Boolean functions and formulas

We consider boolean functions of n arguments, that is, functions from B_n to B_1. These will be represented by boolean formulas over the variables X_i for $i = 1, \ldots, n$, which for convenience we define in terms of AND/OR trees.

We use the symbols $\top$ and $\bot$ for the constant functions 1 and 0 respectively. A *literal* is a variable X_i or its negation, denoted $\neg X_i$. A *boolean formula* is $\top$ or $\bot$ or an AND/OR tree such that every internal node has at least two children, and the leaves are labelled by literals. The *size* of such a formula is 0 if it is $\top$ or $\bot$, and otherwise, the number of leaves in its AND/OR tree.

Each boolean formula represents a boolean function in the usual way. That is, given the boolean vector $y \in B_n$, we determine the value of f on y, denoted $f(y)$, as follows. If f is $\top$ then $f(y) = 1$ and if f is $\bot$ then $f(y) = 0$. Otherwise, define the assignment $a(v) = y_i$ if the label of v is X_i and $a(v) = \overline{y_i}$ if the label of v is $\neg X_i$. This assigns each leaf node a value 0 or 1, and the value of $f(v)$ is $a(T)$, where T is the AND/OR tree of f.

A μ-formula is a boolean formula with the additional restriction that no two leaves are labelled by literals of the same variable. Note that this means that the size of a μ-formula is at most n. For μ-formulas this representation is canonical, that is, μ-formulas representing the same boolean function must have isomorphic trees. Since the the leaves of a μ-formula are labelled by distinct literals, we shall not distinguish between the leaves and their labels.

A *monomial* is a μ-formula consisting of the constant symbol $\top$ or a single literal or a conjunction of literals. A monomial is *non-constant* if it is not equal to $\top$. (Note that it represents a non-constant boolean function.)

An *implicant* of a formula f is a set of literals L such that the conjunction of the literals in L logically implies f. A *minimal implicant* of a formula f is an implicant L such that no proper subset of L is an implicant of f.

A *clause* is a μ-formula consisting of the constant symbol $\bot$ or a single literal or a disjunction of literals. A clause is *non-constant* if it is not equal to $\bot$.

A *consequent* of a formula f is a set of literals L such that f logically implies the disjunction of the literals in L. A *minimal consequent* of a formula f is a consequent L such that no proper subset of L is a consequent of f.

A μ-DNF formula is a μ-formula that is a single monomial or a disjunction of two or more non-constant monomials. A μ-CNF formula is a μ-formula that is a single clause or a conjunction of two or more non-constant clauses.

A μ-formula is *monotone* if it contains no negative literals. Note that if f is a monotone μ-formula, then if $v, w \in B_n$ and $v \leq w$ then $f(v) \leq f(w)$.

1.4 Exact identification using queries

The paper [2] describes a general setting for learning using different types of queries. The following definitions are sufficient for the purposes of this paper.

We consider a setting in which there is a known set of variables $X_1, \ldots, X_n$, and an unknown μ-formula f over these variables. A learning algorithm has access to information about f in the form of specified types of queries, and is required to halt and output a μ-formula logically equivalent to f. This criterion of identification is called *exact identification.*

We shall initially consider two types of queries, though others will be described in Section 6. For a *membership query*, the learning algorithm provides an element $x \in B_n$ and the reply is "yes" if $f(x) = 1$, and "no" if $f(x) = 0$.

For an *equivalence query*, the learning algorithm provides a μ-formula g. The reply is "yes" if g is logically equivalent to f, otherwise the reply is "no". If the reply is "no", a *counterexample* is also returned, that is, an element $x \in B_n$ such that $f(x) \neq g(x)$. The choice of counterexample is assumed to be arbitrary – the learning algorithm must work as advertised no matter what legal counterexample is returned.

Since in the case of μ-formulas both the examples and formulas have size bounded by n, a learning algorithm will be said to run in polynomial time if and only if its running time is bounded by a polynomial in n.

2 Previous results on the learnability of μ-formulas

Valiant[5] has shown that general μ-formulas can be exactly identified in polynomial time using oracles (or queries) called "necessity", "relevant possibility", and "relevant accompaniment". The results in the present paper improve upon and refine this positive result of Valiant's, which is further discussed in Section 6.

Pitt and Valiant [4] consider the *probably approximately correct* identification of μ-formulas and show that general μ-formulas are not *pac*-identifiable in polynomial time if RP $\neq$ NP. This holds even if the target formula is known to be monotone and in disjunctive normal form. They further show that if RP $\neq$ NP then no randomized polynomial time algorithm can find with high probability a μ-formula that covers even an exponentially small fraction of the positive examples while avoiding all the negative examples.

Two additional results concerning μ-formulas appear in the paper by Kearns et al[3]. The first is a reduction, showing if μC is *pac*-identifiable in polynomial time, then C is *pac*-identifiable in polynomial time, where C is any class of boolean formulas and μC is the set of formulas obtained from C by substituting a distinct variable for each occurrence of a variable in a formula from C.

The second result shows that μ-DNF formulas can be approximately identified with high probability in polynomial time assuming uniform distributions on the positive and negative instances. Unfortunately, the uniform distribution is not preserved by the reduction in the preceding result, and the corresponding question for general DNF is open.

To summarize broadly – for the strict context of *pac*-identification, the restriction that each variable appears at most once does not increase learnability of classes of boolean formulas, but under stronger assumptions (restricted classes of distributions, or additional types of oracles) it may.

3 Learning μ-DNF formulas

We describe a polynomial time algorithm that exactly identifies an arbitrary μ-DNF formula f using equivalence and membership queries. For each $x \in B_n$ we define the *sensitive set* of x, denoted $S(x)$, to be the set of those indices i such that complementing x_i changes the value of f on x. More formally,

$$S(x) = \{i : f(x) \neq f(x \oplus \pi_i)\}.$$

If $x \in B_n$, define $Smon(x)$ to be the unique monomial that consists of a conjunction of literals, one for each index $i \in S(x)$, namely the literal X_i if $x_i = 1$ or the literal $\neg X_i$ if $x_i = 0$. The central fact that our algorithm uses is the following.

Lemma 2 *Suppose f is a non-constant μ-DNF formula and x and y are elements of B_n such that $f(x) = 1$ and $f(y) = 0$ and $d(x, y) = 1$. Then $Smon(x)$ is a monomial of f.*

Proof. Suppose x and y differ in position i. Then the variable X_i must appear in some monomial t of f. Since f is a μ-formula, X_i appears in no other monomial of f. Because y agrees with x on all variables other than X_i, it must be the case that t is the only monomial of f such that $t(x) = 1$.

Now we argue that every literal in t is in $Smon(x)$. Suppose X_j is a literal in t. Then $X_j(x) = 1$, that is, $x_j = 1$. So $x \oplus \pi_j$ has a 0 in the j^{th} position, and $X_j(x \oplus \pi_j) = 0$. Since the monomial t contains the literal X_j, $t(x \oplus \pi_j) = 0$. Consider any monomial $t' \neq t$ of f. The variable X_j cannot occur in t', and we know $t'(x) = 0$. Since $x \oplus \pi_j$ agrees with x in all but position j, we must have $t'(x \oplus \pi_j) = 0$. Thus, $f(x \oplus \pi_j) = 0 \neq f(x)$, so $j \in S(v)$. Since $x_j = 1$, this means that the literal $X_j \in Smon(v)$. The argument in case $\neg X_j$ is a literal in t is similar.

We now show that every literal in $Smon(x)$ is in t. Suppose X_j is a literal in $Smon(x)$. Then we know $x_j = 1$ and $f(x \oplus \pi_j) = 0$. Since $t(x) = 1$, if t does not contain the literal X_j, then $t(x \oplus \pi_j) = 1$ and $f(x \oplus \pi_j) = 1$, a contradiction. Hence the literal X_j must be in the monomial t. The argument in case $\neg X_j$ is a literal in $Smon(x)$ is similar. Thus, $Smon(x)$ and t contain precisely the same set of literals. Q.E.D.

The μ-DNF algorithm

1. Make an equivalence query with $\top$. If the reply is "yes", output $\top$ and halt. Otherwise, the reply is a counterexample $z \in B_n$ such that $f(z) = 0$.

2. Initialize $h = \bot$. At each stage h consists of a disjunction of a subset of the monomials of f.

3. Make an equivalence query with h. If the reply is "yes", output h and halt. Otherwise, let $x \in B_n$ be the counterexample. We have $f(x) = 1$ and $h(x) = 0$.

4. Construct a new vector $y \in B_n$ such that $y_i = z_i$ if the variable X_i appears in h, otherwise $y_i = x_i$. For this we also have $f(y) = 1$ and $h(y) = 0$.

5. Find the least positive integer i such that $y_i \neq z_i$. Make a membership query to determine the value of $f(y \oplus \pi_i)$. If $f(y \oplus \pi_i) = 1$, then set $y = y \oplus \pi_i$ and repeat step 5.

6. Otherwise, we have $f(y) = 1$ and $f(y \oplus \pi_i) = 0$. Use membership queries to determine the values of $f(y \oplus \pi_j)$ for all $j \neq i$, and compute the sensitive set of y. Set h to the disjunction of h and $Smon(y)$ and go to step 3.

Theorem 3 *The above algorithm runs in polynomial time and exactly identifies μ-DNF formulas using equivalence and membership queries.*

Proof. Let f be an arbitrary μ-DNF formula over $X_1, \ldots, X_n$. Clearly, if $f = \top$ or $f = \bot$, then the algorithm correctly identifies f in step (1) or the first time step (3) is executed. Suppose f is a non-constant μ-DNF formula. Then in step (1), the value of z is set to some element of B_n such that $f(z) = 0$ and does not subsequently change.

We show that each time step (3) is executed, h consists of a disjunction of a subset of the monomials contained in f. Moreover, we show that each time the loop consisting of steps (3) – (6) is executed, a monomial in f but not in h is added to h. Clearly, the first time step (3) is executed, $h = \bot$, which is the disjunction of the null subset of the monomials of f.

If the algorithm does not halt in step (3), then h is a proper subset of the monomials of f, and the counterexample $x \in B_n$ will be such that $f(x) = 1$ and $h(x) = 0$. Thus, there is at least one monomial t in f but not in h such that $t(x) = 1$.

Consider the vector y constructed in step (4). For each i such that X_i appears in h, $y_i = z_i$. Thus, y and z agree on all the variables that appear in the monomials of h, and since these are a subset of the monomials of f and $f(z) = 0$, we must have $h(y) = 0$. The variables that appear in the monomial t do not appear in h (since f is a μ-formula), so, since y and x agree on the variables that do not appear in h, $t(y) = t(x) = 1$. Therefore, $f(y) = 1$. Hence, we have that $f(y) = 1$ and $h(y) = 0$.

Consider now step (5). Since $f(z) = 0$ and $f(y) = 1$, $z \neq y$, so there exists some least i such that $y_i \neq z_i$. If $f(y \oplus \pi_i) = 1$, then we set $y = y \oplus \pi_i$ and repeat step (5). Note that the initial conditions $f(y) = 1$ and $f(z) = 0$ are preserved. Since the number of bits in which y disagrees with z is monotonically decreasing in this loop, and since $f(z) = 0$, we must after at most n iterations find an i such that $f(y) = 1$ and $f(y \oplus \pi_i) = 0$, and exit the loop to step (6).

In step (6) we determine $Smon(y)$ and set h to the disjunction of h and $Smon(y)$. Lemma 2 guarantees that $Smon(y)$ is a monomial of f, since $d(y, y \oplus \pi_i) = 1$, $f(y) = 1$, and $f(y \oplus \pi_i) = 0$. Let t denote the monomial $Smon(y)$ added to h. To see that t does not already appear in h, note that $t(y) = 1$. However, y initially agrees with z on all the variables in h, and can only be modified in step (5) to agree with z on additional variables, so we have $h(y) = 0$, so the monomial t does not appear in h. Thus, step (6) adds a new monomial of f to h.

The number of times step (3) can be executed is bounded by one more than the number of monomials of f, so it is clear that the algorithm terminates correctly in time bounded by a polynomial in n. Q.E.D.

By duality, an analogous polynomial time algorithm exactly identifies μ-CNF formulas using equivalence and membership queries. It builds the formula a clause at a time, using an analog of Lemma 2 for clauses.

4 Minimal implicants and consequents

In this section we establish a couple of facts about minimal implicants and consequents of general μ-formulas that are of use in subsequent sections. We first characteize minimal implicants and consequents of a μ-formula in terms of minimal positive and negative witnesses of its AND/OR tree. The following lemma is intuitively obvious.

Lemma 4 *Let f be a non-constant μ-formula with AND/OR tree T. If L is an implicant of f then L contains a positive witness for T. If L is a consequent of f then L contains a negative witness for T.*

Proof. For variety, we'll do the dual case. Suppose L is a consequent of f. Consider any vector $x \in B_n$ such that $Y(x) = 0$ for all the literals $Y \in L$, and $Y(x) = 1$ for all other literals Y occurring in f. Since L is a consequent of f, we must have $f(x) = 0$.

Let a be the assignment to the leaves of T corresponding to the vector x. Since $a(T) = f(x) = 0$, the root node of T is labelled with 0. We recursively construct a subtree T' of T consisting exclusively of vertices assigned 0 as follows.

First include the root of T. Then, for each vertex included, if it is labelled with OR all its children must be assigned 0, so include them all. If the vertex included is labelled AND, then at least one of its children must be assigned 0, so choose one such child and include it. The vertices included, together with the induced edges, form a subtree T' of T with the property that all its vertices are assigned 0, and for each vertex v of T', if v is labelled OR, then all the children of v in T are in T', and if v is labelled AND, exactly one of the children of v in T is in T'.

Since leaves of T not in L are assigned 1, the leaves L' of T' are a subset of L, and L' is a (minimal) negative witness for T. Q.E.D.

Thus, we have the following.

Lemma 5 *Let f be a non-constant μ-formula and T its AND/OR tree. Let L be a set of literals. Then L is a minimal implicant of f if and only if L is a minimal positive witness for T. Dually, L is a minimal consequent of f if and only if L is a minimal negative witness for T.*

Proof. We'll prove the first assertion; the second is dual. Recall that Lemma 1 shows that if L is a positive witness for T then L is an implicant of f.

Suppose L is a minimal positive witness for T. Then L is an implicant of f. If L is not a minimal implicant, then there is a proper subset L' of L that is an implicant of f. By Lemma 4, L' contains a positive witness for T, contradicting the assumption that L is minimal positive witness. Hence L is a minimal implicant of f.

Suppose L is a minimal implicant of f. Then by Lemma 4, L contains a positive witness for T. Let L' be a minimal positive witness contained in L. By Lemma 1, L' is an implicant of f, so if L' is a proper subset of L, this would contradict the assumption that L is a minimal implicant. Hence, $L = L'$, and L is a minimal positive witness for T. Q.E.D.

The next result is an easy combinatorial result on the sizes of minimal implicants and minimal consequents of a μ-formula that is used in Section 7.

Lemma 6 *Let T be any AND/OR tree. Let $P(T)$ denote the minimum cardinality of any positive witness for T, and let $N(T)$ denote the minimum cardinality of any negative witness for T, and let $l(T)$ denote the number of leaves of T. Then $P(T)N(T) \leq l(T)$.*

Proof. This is proved by induction on the number of vertices of the tree T. The base case is a tree consisting of a single leaf vertex v. For this tree, $P(T) = N(T) = l(T) = 1$, and the result holds.

Assume that the result holds for any tree with at most $n \geq 1$ vertices, and let T be a tree with $n+1$ vertices. Then the root of T is an internal vertex v_0, which must be labelled AND or OR. Let T_i for $i = 1, \ldots, k$ be the subtrees of T rooted at the children of v_0.

Suppose the label of v_0 is AND. Then a positive witness for T may be obtained by taking the union of any set of positive witnesses for the subtrees T_i, so

$$P(T) \leq P(T_1) + \ldots + P(T_k).$$

A negative witness for T may be obtained by taking a negative witness for any one of the subtrees T_i, so

$$N(T) \leq \min\{N(T_1), \ldots, N(T_k)\}.$$

Since for each $i = 1, \ldots, k$,

$$\min\{N(T_1), \ldots, N(T_k)\} \leq N(T_i),$$

we have

$$P(T)N(T) \leq P(T_1)N(T_1) + \ldots + P(T_k)N(T_k).$$

By the inductive assumption, $P(T_i)N(T_i) \leq l(T_i)$ for each $i = 1, \ldots, k$, so

$$P(T)N(T) \leq l(T_1) + \ldots + l(T_k) = l(T).$$

The case of the label of v_0 being OR is dual. Q.E.D.

Corollary 7 *If f is any non-constant μ-formula of size n, then there is a minimal implicant or a minimal consequent of f of cardinality at most $\sqrt{n}$.*

Proof. By Lemma 5, for a non-constant μ-formula f with AND/OR tree T, each minimal positive witness of T is a minimal implicant of f and each minimal negative witness of T is a minimal consequent of f. Hence, by Lemma 6, the product of the minimum cardinality of a minimal implicant and the minimum cardinality of a minimal consequent is at most the number of leaves of T, that is, at most n. Hence at least one of the minimum cardinality of a minimal implicant of f or the minimum cardinality of a minimal consequent of f must be bounded by $\sqrt{n}$. Q.E.D.

5 Monotone μ-formulas

Now we pass to the case of general μ-formulas which are restricted to be monctone, that is, every literal in the formula is positive. We describe a polynomial time algorithm to learn monotone μ-formulas using only membership queries. The basic idea is to construct the formula recursively from the top down, by gathering enough information to determine whether the root is labelled AND or OR and to simulate membership oracles for the subformulas corresponding to the children of the root.

5.1 Checking for a constant formula

Let f be any monotone boolean formula. If $f(x) = 1$ for the vector x of all 0's, $f(y) = 1$ for all vectors $y \in B_n$, so $f \equiv \top$. Similarly, if $f(x) = 0$ for the vector x of all 1's, then $f \equiv \bot$. Thus, using two membership queries we can determine whether an arbitrary monotone boolean formula f is equivalent to $\top$ or $\bot$ or neither.

5.2 Finding implicants and consequents of monotone formulas

Lemma 8 *Let f be a monotone boolean formula and let L be a set of variables. Then L is an implicant of f if and only if $f(\pi_L) = 1$ and L is a consequent of f if and only if $f(\overline{\pi_L}) = 0$*

Proof. By the definition of an implicant, if L is an implicant of f then $f(\pi_L) = 1$. Conversely, if $f(\pi_L) = 1$, then for any vector y such that $y_i = 1$ for every i such that $X_i \in L$, we have $\pi_L \leq y$ and therefore $f(y) = 1$, because f is monotone. The other case is dual. Q.E.D.

Suppose we have a membership oracle for a non-constant monotone formula f. Then we can determine a minimal implicant of f as follows. Let z denote the vector of all 0's. Thus $\overline{z}$ is the vector of all 1's. Since f is non-constant and monotone, we must have $f(z) = 0$ and $f(\overline{z}) = 1$. Then we change bits of $\overline{z}$ from 1 to 0 using a greedy algorithm, preserving the condition that f is still 1 on the resulting vector.

More precisely, the algorithm is as follows.

1. Let $x = \overline{z}$.

2. For $i = 1, \ldots, n$, if $f(x \oplus \pi_i) = 1$ set $x = x \oplus \pi_i$.

Note that this procedure uses at n queries to the membership oracle for f in step (2). Concerning the resulting vector x, we have the following.

Lemma 9 *Let L denote the set of variables X_i such that $x_i = 1$ at the end of this procedure. Then L is a minimal implicant of f.*

Proof. Clearly, at the end of the procedure we have $f(x) = 1$, and therefore L is an implicant of f, since $x = \pi_L$. Suppose L is not a minimal implicant. Then there is some proper subset L' of L that is also an implicant of f. Let $y \in B_n$ be the vector such that $y_i = 1$ if and only if $X_i \in L'$. Note that $f(y) = 1$.

Consider the least integer i such that X_i is in L but not in L'. Then, when the index i was considered in step (2) of the procedure, $y \leq (x \oplus \pi_i)$, so since $f(y) = 1$ and f is monotone, $f(x \oplus \pi_i) = 1$, so bit i must be 0 in the final value of x, a contradiction. Hence L is a minimal implicant. Q.E.D.

The dual case, of finding minimal consequents, simply starts with $x = z$ instead of $x = \overline{z}$, and preserves the condition $f(x) = 0$. The analog of the lemma above is that for the resulting vector x, the set L of variables X_i such that $x_i = 0$ is a minimal consequent of f. Thus, given a membership oracle for a non-constant monotone formula f, we can determine a minimal implicant and a minimal consequent of f in polynomial time.

5.3 Checking for equivalence to X_i

Lemma 10 *Let f be any monotone boolean formula. Then $f \equiv X_i$ if and only if $f(\pi_i) = 1$ and $f(\overline{\pi_i}) = 0$.*

Proof. If $f \equiv X_i$ then clearly $f(\pi_i) = 1$ and $f(\overline{\pi_i}) = 0$. Conversely, if $f(\pi_i) = 1$ then $\{X_i\}$ is an implicant of f and if $f(\overline{\pi_i}) = 0$ then $\{X_i\}$ is a consequent of f, and therefore, $f \equiv X_i$. Q.E.D.

Hence, with at most $2n$ membership queries we can determine whether $f \equiv X_i$ for some $i = 1, \dots, n$.

5.4 The more complex case

Suppose f is a monotone μ-formula that is not equivalent to $\top$, $\bot$, or X_i for any $i = 1, \dots, n$. Assume we have a procedure that, given a membership oracle for f, will return one of the following two possible results.

1. The value "AND" and two nonempty disjoint sets L_1 and L_2 of variables such that there are two non-constant monotone μ-formulas g and h containing disjoint sets of variables such that $f \equiv (g \wedge h)$, and L_1 is a minimal implicant for g, and L_2 is a minimal implicant for h.

2. The value "OR" and two nonempty disjoint sets L_1 and L_2 of variables such that there are two non-constant monotone μ-formulas g and h containing disjoint sets of variables such that $f \equiv (g \vee h)$, and L_1 is a minimal consequent for g, and L_2 is a minimal consequent for h.

Then we could simulate membership oracles for the formulas g and h as follows. In case (1), given the vector $x \in B_n$, in order to determine $g(x)$, we construct the vector y such that $y_i = 1$ if $X_i \in L_2$, and $y_i = x_i$ otherwise. Since y assigns 1 to every variable in L_2 and L_2 is an implicant of h, $h(y) = 1$. Since no variable in L_2 appears in g, $g(y) = g(x)$. Hence,

$$f(y) = (g(y) \wedge h(y)) = (g(y) \wedge 1) = g(y) = g(x).$$

Thus, by querying the membership oracle for f on the vector y, we determine the value of $g(x)$. Determining $h(x)$ in case (1) is symmetric.

In case (2), given $x \in B_n$, to determine $g(x)$ we construct the vector y such that $y_i = 0$ for all i such that $X_i \in L_2$, and $y_i = x_i$ otherwise. Then

$$f(y) = (g(y) \vee h(y)) = (g(y) \vee 0) = g(y) = g(x).$$

Determining $h(x)$ in case (2) is symmetric.

Since we could simulate membership oracles for the formulas g and h in these two cases, we could continue by determining whether they are equivalent to $\top$, $\bot$, or X_i for some i, and, if not, recursively using the procedure assumed above to determine the root label and a nontrivial division into other formulas. Below we make this argument more precise and show how to implement the required subprocedure.

5.5 Determining a non-trivial division

Suppose that f is a monotone μ-formula that is not equivalent to $\top$, $\bot$, or X_i for any $i = 1, \ldots, n$. Let T be the AND/OR tree of f. Let T_j denote the subtrees of T rooted at the children of the root, for $j = 1, \ldots, k$, and let f_j be the monotone μ-formula represented by T_j. Note that $k \geq 2$.

Let P be any minimal implicant of f. There is a partition P_j for $j = 1, \ldots, k$ of P such that P_j is precisely the elements of P that appear in the subtree T_j. If the root label of T is AND, then for each j, P_j is a minimal implicant for f_j. If the root label of T is OR, then for exactly one j, P_j is a minimal implicant for f_j. For each variable $X_i \in P$, let $[X_i]$ denote the partition block P_j containing X_i.

We choose any variable $X_i \in P$, and our goal is to determine the elements of $[X_i]$. Then if the root label of T is AND, our non-trivial division of P will be into the sets $[X_i]$ and $P - [X_i]$. If the root label of T is OR, then $[X_i] = P$, and we continue with the dual computation to find a minimal consequent of f and one of its partition elements.

5.6 Compensators

Let f be a non-constant monotone μ-formula and P a minimal implicant of f. Let T be the AND/OR tree of f and let T_j for $j = 1, \ldots, k$ be the subtrees of T rooted at the children of the root of T.

Let Q be a set of variables not containing X_i such that $(P - \{X_i\}) \cup Q$ is an implicant of f. Then Q is a *positive compensator* for X_i in P. If Q is a positive compensator for X_i in P such that no proper subset of Q is a positive compensator for X_i in P, then Q is a *minimal positive compensator* for X_i in P. We may characterize the minimal positive compensators for X_i in P as follows.

Lemma 11 *The set Q is a minimal positive compensator for X_i in P if and only if there exists a vertex v labelled OR on the path from X_i to the root of T and Q is a minimal implicant of a subtree not containing X_i and rooted at one of the children of v.*

Proof. Suppose Q is a minimal positive compensator for X_i in P. Let $S = (P - \{X_i\}) \cup Q$. Let j be such that the subtree T_j contains X_i. Consider the assignment of 0 or 1 to the vertices of T induced by the assignment corresponding to the vector π_S. We have $f(\pi_S) = 1$, so the root of T must be assigned 1, and the leaf X_i is assigned 0. Let w be the highest vertex on the path from X_i to the root of T that is assigned 0, and let v be the parent of w.

Since v is assigned 1 and one of its children is assigned 0, v must be labelled OR. Moreover, one of the other children, say w', of v must be assigned 1. Let T' be the subtree of T rooted at w'. Let Q' consist of those elements of Q that are in T'.

Then it is clear that Q' is an implicant of the subtree T', so let $Q'' \subseteq Q'$ be a minimal implicant of T'. Then $(P - \{X_i\}) \cup Q''$ is an implicant of f. Since Q was chosen to be minimal with this property, and $Q'' \subseteq Q$, we must have $Q = Q''$. Thus, Q is a minimal implicant of T', which is the subtree of T rooted at w', which is a child of the node v, which is labelled OR and is on the path from X_i to the root of T. Moreover, T' does not contain X_i.

Conversely, suppose Q is a minimal implicant of a subtree T' rooted at a vertex w' that is a child of a vertex v labelled OR on the path from X_i to the root of T, and that X_i is not in T. Then $(P - \{X_i\}) \cup Q$ is an implicant of f. Moreover, suppose a proper subset Q' of Q has the property that $(P - \{X_i\}) \cup Q'$ is an implicant of f. Then it is easy to show that Q' must be an implicant of T', contradicting the choice of Q as a minimal implicant of T'. Q.E.D.

The following lemma characterizes the partition element $[X_i]$ in terms of minimal positive compensators of X_i in P.

Lemma 12 *Let T_j be the subtree containing X_i that is rooted at a child of the root of T, and let $[X_i]$ be the set of all leaves of T_j that are in P. Then, for all $X_m \in (P - \{X_i\})$, $X_m \in [X_i]$ if and only if there is a minimal positive compensator Q for X_i in P such that $(P - \{X_m\}) \cup Q$ is an implicant of f.*

Proof. Suppose $X_m \in (P - \{X_i\})$ is an element of $[X_i]$. Thus, X_m is a leaf of T_j. Let w be the lowest common ancestor of X_m and X_i in T_j. Since P is a minimal implicant, the label of w must be AND. Let v be the parent of w in T. (Note that w is in T_j and is therefore not the root of T.) The label of v must be OR.

There exists a vertex w' that is a child of v and not equal to w. Let T' be the subtree of T rooted at w' and let Q be a minimal implicant of T'. Note that X_m is not in T'. By the preceding lemma, Q is a minimal positive compensator for X_i in P. Moreover, $(P - \{X_m\}) \cup Q$ is an implicant of f.

Conversely, let $X_m \in (P - \{X_i\})$ be such that there is a minimal positive compensator Q for X_i in P such that $(P - \{X_m\}) \cup Q$ is an implicant of f. Suppose X_m is in the subtree T_r for some $r \neq j$. Then the lowest common ancestor of X_i and X_m is the root of T, which must therefore be labelled AND.

By the preceding lemma, there is a vertex v on the path from X_i to the root of T such that v is labelled OR and there is a child w' of v not on the path from X_i to the root of T such that Q is a minimal implicant of the subtree T' rooted at w'. Note that v and T' must be in the subtree T_j, since v is labelled OR and the root of T is labelled AND.

Then $(P - \{X_m\})$ fails to contain an implicant for T_s, and since all of the elements of Q are in T_j, $(P - \{X_m\}) \cup Q$ also fails to contain an implicant for T_s. Hence $(P - \{X_m\}) \cup Q$ does not contain an implicant for f, contradiction the assumptions of this case. Hence $X_m \in [X_i]$. Q.E.D.

We summarize the facts for the dual case, which hold by analogous reasoning. Let f be a non-constant monotone μ-formula with AND/OR tree T. Let C be any minimal consequent of f and let $X_i \in (C - \{X_i\})$. A set Q is a *negative compensator* for X_i in C if and only if Q is a set of variables not containing X_i such that $(C - \{X_i\}) \cup Q$ is a consequent of f. If Q is a negative compensator for X_i in C and no proper subset of Q is a negative compensator for X_i in C, then Q is a *minimal negative compensator* for X_i in C.

Lemma 13 *The set Q is a minimal negative compensator for X_i in C if and only if there exists a vertex v labelled AND on the path from X_i to the root of T and Q is a minimal consequent of a subtree not containing X_i and rooted at one of the children of v.*

Lemma 14 *Let T_j be the subtree of T containing X_i that is rooted at a child of the root of T, and let $[X_i]$ be the set of all leaves of T_j that are in C. Then, for all $X_m \in (C - \{X_i\})$, $X_m \in [X_i]$ if and only if there is a minimal negative compensator Q for X_i in C such that $(C - \{X_m\}) \cup Q$ is a consequent of f.*

5.7 Procedure for finding root label and non-trivial division

We are now ready to specify the algorithm to determine the root label and a non-trivial division of a minimal implicant or consequent. Suppose we are given a membership oracle for a non-constant monotone μ-formula f.

Divide

1. Find a minimal implicant P and a minimal consequent C of f. If $P = C = \{X_i\}$ then return the label "variable" and the variable X_i.
2. Select any $X_i \in P$. Let $D = \{X_i\}$. Let $R = \{X_1, \ldots, X_n\} - P$.
3. Find a minimal set $Q \subseteq R$ of variables such that $Q \cup (P - \{X_i\})$ is an implicant of f.
4. If such a Q is found, add all those variables $X_j \in P$ such that $Q \cup (P - \{X_j\})$ is an implicant of f to D. Then set $R = R - Q$, and go to step 3.
5. If there is no such Q and $D \neq P$ then return the label "AND" and the sets D and $P - D$. Otherwise, continue with the following step.
6. Select any $X_i \in C$. Let $D = \{X_i\}$. Let $R = \{X_1, \ldots, X_n\} - C$.
7. Find a minimal set $Q \subseteq R$ of variables such that $Q \cup (P - \{X_i\})$ is a consequent of f.
8. If such a Q is found, add all those variables $X_j \in C$ such that $Q \cup (C - \{X_j\})$ is a consequent of f to D. Then set $R = R - Q$, and go to step 7.
9. If there is no such Q then return the label "OR" and the sets D and $C - D$.

Concerning this procedure, we prove the following.

Lemma 15 *Let f be any non-constant monotone μ-formula, and let T be the AND/OR tree of f. Suppose the procedure above is run with a membership oracle for f. If $f = X_i$ for some i, then the procedure returns the label "variable"and the variable X_i. Otherwise, let l be the label of the root of T and let $T_1, \ldots, T_k$ be the subtrees rooted at the children of the root of T. Then $k \geq 2$. If l is AND then the procedure returns the label "AND" and a pair of nonempty disjoint sets D and E such that D is a minimal implicant one of the T_j's, and E is a minimal implicant of the conjunction of the rest of the T_j's. If l is OR then the procedure returns the label "OR" and a pair of nonempty disjoint sets D and E such that D is a minimal consequent of one of the T_j's, and E is a minimal consequent of the disjunction of the rest of the T_j's.*

Proof. Clearly, if $f = X_i$ then the minimal implicant P and the minimal consequent C must both be equal to $\{X_i\}$, so the procedure returns the label "variable" and the variable X_i.

Otherwise, the tree T must have a root vertex labelled AND or OR and there must be $k \geq 2$ children of the root. Let T_j for $j = 1, \ldots, k$ denote the subtrees rooted at the children of the root of T. Let l denote the label of the root of T.

If l is AND, then each minimal implicant of f must contain at least two variables, and if l is OR, then each minimal consequent of f must contain at least two variables. Hence, in either case, we cannot have $P = C = \{X_i\}$ for any i, and the procedure reaches step (2).

When step (3) is executed for the first time, R consists of all variables except those in P, and at subsequent executions R is strictly decreasing in the subset ordering. Thus, there can be no more than n iterations of the loop consisting of steps (3) and (4). Also, if a set $Q \subseteq R$ is found such that $Q \cup (P - \{X_i\})$ is an implicant of f and this does not hold of any subset of Q, then by definition, Q is a minimal positive compensator for X_i in P.

By Lemma 12, if Q is a minimal positive compensator for X_i in P and X_j is such that $Q \cup (P - \{X_j\})$ is an implicant of f, then $X_j \in [X_i]$. Hence, in step (4), every variable X_j added to D is an element of $[X_i]$. Since the first time step (4) is executed, $D = \{X_i\}$, D is a subset of $[X_i]$ every time step (4) is compeleted.

We need to show that when step (5) is executed, $D = [X_i]$. Suppose that there is some element $X_m \in ([X_i] - D)$ when step (5) is executed. By Lemma 12, there is a minimal positive compensator Q_m for X_i in P such that $Q_m \cup (P - \{X_m\})$ is an implicant of f. Clearly, Q_m is disjoint from P, so Q_m is a subset of R the first time step (3) is executed. By Lemma 11, Q_m is a minimal implicant of a subtree T' rooted at a vertex w' that is a child of a vertex v such that v is labelled OR and is on the path from X_i to the root of T. Moreover, X_i is not in T'.

Whenever a set Q is found in step (3), it is a minimal implicant of a subtree of T that is rooted at a vertex that is a child of some vertex on the path from X_i to the root of T. Hence, every such Q is either a minimal implicant of T' or is disjoint from T'. If Q is a minimal implicant of T', then X_m must be added to D, but we are assuming $X_m \in ([X_i] - D)$ when step (5) is executed, so every Q found in step (3) and removed from R in step (4) must be disjoint from T'. But this means that when step (5) is executed, the set Q_m is still a subset of R, contradicting the entry condition for step (5). This contradiction shows that when step (5) is executed, $D = [X_i]$.

If the root label l of T is AND, then the partition element $[X_i]$ is not equal to P, and in step (5) the procedure will return the label "AND" and the non-trivial partition $[X_i]$ and $P - [X_i]$. The set $[X_i]$ is a minimal implicant of the subtree T_j containing X_i, and $P - [X_i]$ is a minimal implicant of the conjunction of the rest of the subtrees. Thus, if l is AND, correct values are returned in step (5).

If l is OR, then P is a prime implicant of just one subtree T_j, so the partition element $[X_i]$ will contain all the variables in P. Hence, in this case, execution will proceed with step (6). The proof that this portion of the procedure will correctly return a non-trivial partition of the minimal consequent C is analogous. Q.E.D.

5.8 The main procedure

Suppose we are given a membership oracle for a monotone μ-formula f. We first describe a *modified membership oracle* for f. This is initialized with a pair of disjoint sets S_0 and S_1. It takes as input a boolean vector $x \in B_n$. It computes the vector y such that $y_i = 0$ if $X_i \in S_0$, $y_i = 1$ if $X_i \in S_1$, and $y_i = x_i$ otherwise. It then queries the true membership oracle for f on the vector y and returns the reply as its reply. Note that when S_0 and S_1 are each the empty set, this is equivalent to the membership oracle for f.

The central procedure called *MM*. It takes as a pair of disjoint sets S_0 and S_1 of variables, calls the procedure *Divide* to establish the label of the root and a non-trivial division of a minimal implicant or consequent. Then it calls itself recursively to determine formulas for the separate portions of the tree, and combines these formulas using conjunction or disjunction.

$$MM(S_0, S_1)$$

1. Call the procedure *Divide* with the modified membership oracle for f initialized with the sets S_0 and S_1.
2. If *Divide* returns "variable" and the variable X_i, return the formula X_i.
3. If *Divide* returns "AND" and the sets D and E then recursively call $MM(S_0, S_1 \cup E)$, which returns formula g, and recursively call $MM(S_0, S_1 \cup D)$, which returns formula h. Return the formula that represents the conjunction of g and h.
4. If *Divide* returns "OR" and the sets D and E then recursively call $MM(S_0 \cup E, S_1)$, which returns formula g, and recursively call $MM(S_0 \cup D, S_1)$, which returns formula h. Return the formula that represents the disjunction of g and h.

Lemma 16 *Let f and f' be non-constant monotone μ-formulas. Suppose the procedure MM is called with the membership oracle for f and a pair of disjoint sets S_0 and S_1 such that the modified membership oracle for f initialized with S_0 and S_1 is equivalent to the membership oracle for f'. Then MM runs in polynomial time and returns a monotone μ-formula equivalent to f'.*

Proof. The proof is by induction on the size of the formula f'. Clearly, if f' is X_i for some i, then by the fact that the modified membership oracle is equivalent to the true membership oracle for f', and the correctness of *Divide*, the value returned is the formula X_i.

Assume that *MM* works correctly on all non-constant monotone μ-formulas f' of size at most s, for some $s \geq 1$. Let f' be a non-constant monotone μ-formula with size $s+1$. Let T be the AND/OR tree of f'. Then T must have a root label of AND or OR, and the root has $k \geq 2$ children. Let T_j for $j = 1, \ldots, k$ be the subtrees of T rooted at the children of the root of T.

Suppose the monotone μ-formula f and the disjoint sets of variables S_0 and S_1 are such that the modified membership oracle for f initialized with S_0 and S_1 is equivalent to the membership oracle for f'. Then, by the correctness of *Divide*, the values returned will be the correct root label and a correct partition of a minimal implicant or consequent of f'.

Suppose the root label of f' is AND. Then, in particular, *Divide* will return "AND" and two sets D and E such that D is a minimal implicant of some T_j and E is a minimal implicant of the conjunction of T_r such that $r \neq j$. Let g' be the monotone μ-formula represented by T_j, and let h' be the monotone μ formula obtained as follows. If $k = 2$, then let h' be the unique T_r such that $r \neq j$. If $k > 2$, then let h' be the tree T with the subtree T_j removed. Then f' is equivalent to $g' \wedge h'$. Note that the sizes of g' and h' will be at most s.

In step (3), the first recursive call of *MM* will be with the sets S_0 and $S_1 \cup E$. It is not difficult to show that E is disjoint from S_0 and S_1. We need to show that the modified membership oracle for f initialized with the sets S_0 and $S_1 \cup E$ is the membership oracle for the formula g'.

Let $x \in B_n$. Let x' be defined by $x'_i = 1$ if $X_i \in E$ and $x'_i = x_i$ otherwise. Since E is an implicant for h' and contains no variables of g', we have

$$f'(x') = g'(x') \wedge h'(x') = g'(x) \wedge 1 = g'(x).$$

Let y be defined by $y_i = 0$ if $X_i \in S_0$, $y_i = 1$ if $X_i \in S_1 \cup E$, and $y_i = x_i$ otherwise. When we call the modified membership oracle for f initialized with S_0 and $S_1 \cup E$ on the input x, it returns "yes" if and only if $f(y) = 1$. However, $y_i = 0$ if $X_i \in S_0$, $y_i = 1$ if $X_i \in S_1$, and otherwise $y_i = x'_i$. Hence, by the assumption that the modified membership oracle for f initialized with S_0 and S_1 is a correct membership oracle for f', we must have $f'(x') = f(y)$.

Since $f'(x') = g'(x)$, a call with input x to the modified membership oracle for f initialized with S_0 and $S_1 \cup E$ returns "yes" if and only if $g'(x) = 1$. That is, it is equivalent to a membership oracle for g'. Thus, the first recursive call in step (3) returns a monotone μ-formula g equivalent to g', by the inductive assumption, because the size of g' is at most s.

Similarly, we argue that the second recursive call returns a monotone μ-formula h equivalent to h'. The procedure then constructs a monotone μ-formula that represents the conjunction of g and h. That is, if the root label of h is AND, the tree for g is made a child of the root of h. If the root label of h is not AND, g and h are made children of a new root labelled AND. This formula is a monotone μ-formula equivalent to f'. The case for the root label of f' being OR is dual.

To see that the procedure *MM* runs in polynomial time, note that each vertex of f' can be charged with one call to *MM*, namely, the call that establishes its label. Each call to *MM* consists of a call to *Divide* and some other processing that is clearly bounded by a polynomial in n, the total number of variables. Q.E.D.

We have as a corollary the main result of this section.

Theorem 17 *There is a polynomial time algorithm that exactly identifies monotone μ-formulas using membership queries.*

Proof. Given a membership oracle for a monotone μ-formula f, we first query f on the vector of all 0's. If the reply is "yes", we output $\top$ and halt. Otherwise, we query f on the vector of all 1's. If the reply is "no", we output $\perp$ and halt. Otherwise f is non-constant, and we make a top-level call to the procedure *MM* with the sets S_0 and S_1 both equal to the empty set. In this case the modified membership oracle for f initialized with S_0 and S_1 is the correct membership oracle for f, so by Lemma 16, *MM* returns in polynomial time a monotone μ-formula equivalent to f. We output the formula returned and halt. Q.E.D.

6 General (non-monotone) μ-formulas

In the non-monotone case, the literals that appear in a μ-formula may be positive or negative. We have the following easy result.

Lemma 18 *No polynomial time algorithm can exactly identify all the (non-monotone) μ-formulas using only membership queries.*

Proof. Suppose A is an algorithm that correctly identifies every μ-formula over the variables $\{X_1, \ldots, X_n\}$ and runs in time $p(n)$ for some polynomial p. Choose n sufficiently large that $2^n > p(n) + 1$.

Consider the set S_n of monomials that contain exactly n literals over the set of variables $\{X_1, \ldots, X_n\}$. Recall that no monomial contains both a literal and its negation, by definition. $|S_n| = 2^n$.

Run the algorithm A, answering "no" to each of its membership queries until it halts or until it has run for $p(n)$ steps, whichever comes first. Because $2^n > p(n) + 1$, there are at least two monomials in S_n that are consistent with all the answers to queries so far, so A must either give an incorrect output for one of them or it must run for more than $p(n)$ steps, in either case contradicting the assumptions on A. Q.E.D.

Thus, if a polynomial time algorithm is to identify all (non-monotone) μ-formulas, it will require something other than just membership queries. The information that we are lacking to reduce the non-monotone case to the monotone case is the signs of the literals that occur in f. Valiant's algorithm [5] to exactly identify an arbitrary μ-formula f makes use of several kinds of queries concerning f, which we now briefly describe.

The input to the *necessity oracle* is a set L of literals, and the result is 1 if and only if L is an implicant of f. The *relevant possibility oracle* takes as input a set L of literals and returns 1 if and only if L is a subset of some minimal implicant of f. The *relevant accompaniment oracle* takes as input two sets V and W of variables and the result is 1 if and only if every minimal implicant of f that contains a variable from V also contains a variable from W.

The necessity oracle gives, in the terminology of [2], a subset query for f in which the input is restricted to be a monomial. That is, if the input to the necessity oracle is L and m is the conjunction of the literals in L, the result of the query is 1 if and only if m logically implies f, that is, if and only if all the assignments that satisfy m also satisfy f.

The relevant possibility oracle can be used to determine the signs of the literals occurring in f, as follows.

Lemma 19 *The literal Y occurs in f if and only if the relevant possibility oracle with input $\{Y\}$ returns 1.*

Proof. It is clear from Lemma 5 that if the literal Y occurs in a minimal implicant then it must be a leaf of the AND/OR tree of f, that is, it must occur in f.

Conversely, suppose that Y is a literal labelling a leaf of the AND/OR tree T of f. Then we may construct a minimal implicant of f containing Y as follows. Initially let $L = \{Y\}$. Consider the path from Y to the root of T. At each vertex v labelled AND, let $w_1, \ldots, w_k$ be the children of v that are not on the path from Y to the root of T. For each w_j, add a minimal implicant of the subtree of T rooted at w_j to L. It clear that the final set L is a minimal implicant of f that contains Y. Q.E.D.

Corollary 20 *There is a polynomial time algorithm that exactly identifies every μ-formula using membership and relevant possibility queries.*

Proof. Let f be the μ-formula to be identified. Begin by querying the relevant possibility oracle with $\{Y\}$ for each of the $2n$ possible literals Y. This determines exactly which literals Y occur in f.

Now we run the procedure for monotone μ-formulas, except that each query to the membership oracle is modified as follows. If the element being queried is $x \in B_n$, then query instead the vector y such that $y_i = \overline{x_i}$ if $\neg X_i$ occurs in f, and $y_i = x_i$ otherwise.

The procedure to identify monotone formulas will eventually return a monotone μ-formula g. In this formula, replace X_i by $\neg X_i$ if $\neg X_i$ occurs in f, for each $i = 1, \ldots, n$. Then the resulting formula f' will be equivalent to f. Q.E.D.

This algorithm uses membership queries, which are weaker than necessity queries, and relevant possibility queries (just to determine the proper signs of the literals.) It does not need relevant accompaniment queries.

7 Equivalence queries do not suffice

In this section we show that no polynomial algorithm can exactly identify all the μ-formulas using only equivalence queries, even if the target formula is known to be monotone and in disjunctive normal form. A related result in [1] shows that no polynomial time algorithm can exactly identify all disjunctive normal form formulas using only equivalence queries.

The proof in that paper makes use of the fact that any particular DNF formula is satisfied by some assignment with "few" 1's or falsified by some assignment with "few" 0's. Moreover, the behavior of the formula on this assignment is shared by "few" other DNF formulas. Thus, when the learning algorithm makes a query with a DNF formula, one of these distinguishing assignments can be returned as the counterexample, which eliminates the proposed formula and very "few" others. This shows that "many" equivalence queries will be necessary to narrow down the consistent hypotheses to just one.

The result for DNF formulas does not immediately imply the result for μ-formulas, since it does not allow for the input to the equivalence query to be a general μ-formula (not necessarily in disjunctive normal form) and the target class of formulas is not a class of μ-formulas. However, a very similar proof works for μ-formulas, since Corollary 7 shows that every μ-formula is satisfied by an assignment with "few" 1's or falsified by and assignment with "few" 0's. What remains to be done is to define a target class of μ-formulas for which using these distinguishing assignments as counterexamples eliminates "very few" hypotheses.

7.1 The target class

Let $m \geq 1$ and let $n = m^2$. Let H_n be the class of all monotone μ-DNF formulas with exactly m literals in each monomial. There is a one-to-one correspondence of elements of H_n with permutations of the n variables. That is, each element of H_n can be imagined as the result of taking a permutation of the variables and grouping them, in order, into m monomials of size m each. Thus, $|H_n| = n!$.

Now each element of H_n is logically equivalent to those elements that can be obtained by permuting the monomials and by permuting the variables within a monomial. Thus, each element of H_n is logically equivalent to $(m!)^2$ others. Thus, the elements of H_n represent $n!/(m!)^2$ logically distinct functions.

Suppose $x \in B_n$ is a boolean vector such that $x_i = 1$ for exactly m values of i. Let L_x be the set of variables X_i such that $x_i = 1$. How many elements $f \in H_n$ have $f(x) = 1$? In order to have $f(x) = 1$, one of the monomials of f must contain exactly the variables in L_x. There are m choices for which monomial this is, and $m!$ permutations of L_x to constitute it. For the remaining $n - m$ variables, there are $(n - m)!$ permutations giving the order in which they are placed in the remaining $n - m$ positions. Thus, there are

$$m \cdot m! \cdot (n - m)!$$

elements $f \in H_n$ such that $f(x) = 1$.

Suppose that $x \in B_n$ is a boolean vector such that $x_i = 0$ for exactly m values of i. Let L_x be the set of variables X_i such that $x_i = 0$. How many elements $f \in H_n$ are such that $f(x) = 0$? In order to have $f(x) = 0$, each monomial of f must contain at least one element of L_x. However, since there are m elements of L_x and m monomials that contain disjoint sets of variables, this means that each monomial of f must contain exactly one element of L_x. There are m^m choices of one place in each monomial for the elements of L_x, and $m!$ permutations of L_x by which they might be filled. For the remaining $n - m$ elements, there are $(n - m)!$ orders in which to put them into the remaining $n - m$ places. Thus, there are

$$m^m \cdot m! \cdot (n - m)!$$

elements $f \in H_n$ such that $f(x) = 0$.

Lemma 21 *For any constant $C > 1$ and for all sufficiently large integers m, if $n = m^2$ then*

$$(m^m \cdot m! \cdot (n - m)!)/n! \leq C\sqrt{2\pi(m-1)}e^{-(m-1)}.$$

Proof. Let $C > 1$ be given. Choose $\epsilon > 0$ sufficiently small that

$$(1 + \epsilon)^2/(1 - \epsilon) \leq C.$$

By Stirling's approximation to the factorial, we may choose $m > 1$ sufficiently large that for all $k \geq m$,

$$(1 - \epsilon)\sqrt{2\pi k}(k^k/e^k) \leq k! \leq (1 + \epsilon)\sqrt{2\pi k}(k^k/e^k).$$

Let $n = m^2$. Then, since $n \geq (n - m) \geq m$, we have after some simplification,

$$(m^m \cdot m! \cdot (n - m)!)/n! \leq C\sqrt{2\pi(m-1)}(1 - m/n)^{n-m}.$$

Using the fact that $(1-x) \leq e^{-x}$, we have

$$(m^m \cdot m! \cdot (n-m)!)/n! \leq C\sqrt{2\pi(m-1)}e^{(m-1)}.$$

Q.E.D.

The main result now follows easily.

Theorem 22 *There is no polynomial time algorithm that exactly identifies all μ-formulas using equivalence queries, even if the target class is known to be monotone and in disjunctive normal form.*

Proof. Suppose to the contrary that A is a polynomial time algorithm that exactly identifies all μ-formulas using equivalence queries. Let $p(n)$ be a polynomial bounding the running time of A. Let $C > 1$, and let $B(m)$ be the function defined by

$$B(m) = C\sqrt{2\pi(m-1)}e^{-(m-1)}.$$

Choose m sufficiently large that for $n = m^2$ we have

$$(m^m \cdot m! \cdot (n-m)!)/n! \leq B(m),$$

(by Lemma 21) and also

$$n!(1 - B(m)p(n)) > (m!)^2.$$

This latter is possible since for $n = m^2$, $B(m)p(n) \to 0$ as $m \to \infty$. Let $n = m^2$.

Now consider the following adversary strategy. Run algorithm A with variable set $\{X_1, \ldots, X_n\}$ until it makes an equivalence query or has run for $p(n)$ steps, whichever comes first. If it makes an equivalence query with the μ-formula g, then we answer as follows.

If g is $\top$, the reply is "no" and the counterexample is the vector of all 0's. If g is $\bot$, the reply is "no" and the counterexample is the vector of all 1's. Otherwise, g is a non-constant μ-formula of size at most n. By Corollary 7, g must have a minimal implicant or a minimal consequent of size at most $\sqrt{n} = m$.

Suppose g has a minimal implicant L of size at most m. Thus, any vector $x \in B_n$ that assigns 1 to all the literals in L must also assign 1 to g. Let r denote the number of positive literals in L. Let S be a set of $m - r$ variables that are distinct from all the variables occurring in literals of L.

Define a vector x as follows: $x_i = 1$ if the variable X_i occurs as a positive literal in L or if $X_i \in S$. Otherwise, $x_i = 0$. Then x is a vector that contains exactly m 1's and assigns the value 1 to all the literals in L, so $g(x) = 1$. The reply to the equivalence query in this case is "no" and the counterexample is the vector x.

Dually, suppose g has a minimal consequent L of size at most m. As above, we can find a vector $x \in B_n$ such that x contains exactly m 0's and $g(x) = 0$. In this case the reply to the equivalence query is "no" and the counterexample is x.

Consider now the target set H_n. We argue that after $p(n)$ steps of A, at least two logically inequivalent members of H_n are consistent with all the replies given to equivalence queries. To see this, note that each element of H_n is a non-constant μ-formula, so the counterexamples given in response to $\top$ or $\perp$ do not eliminate any elements of H_n.

In case the counterexample is a vector x with m 1's such that $g(x) = 1$, then we have argued above that there are at most

$$m \cdot m! \cdot (n - m)!$$

elements $f \in H_n$ such that $f(x) = 1$. By our choice of m, since $m \leq m^m$, this means that no more than $B(m)n!$ elements are eliminated from H_n by this counterexample.

In the case that the counterexample is a vector x with m 0's such that $g(x) = 0$, we have argued above that there are at most

$$m^m \cdot m! \cdot (n - m)!$$

elements $f \in H_n$ such that $f(x) = 0$. By our choice of m, this means that at most $B(m)n!$ elements are eliminated from H_n by this counterexample.

Thus, each counterexample eliminates at most $B(m)n!$ elements from H_n. Since H_n initially contains $n!$ elements, this means that when A has run for no more than $p(n)$ steps, there are at least

$$n! - p(n)B(m)n! = n!(1 - p(n)B(m) > (m!)^2$$

elements remaining in H_n that are consistent with all the counterexamples returned to this point. Since each element of H_n is logically equivalent to $(m!)^2$ elements of H_n, this means that at least two logically inequivalent elements of H_n are consistent with all the replies to this point. Hence, A must either run for more than $p(n)$ steps, or it must fail to identify correctly at least one of the elements of H_n. This contradiction shows that no such A can exist. Note that the set H_n contains only μ-formulas that are monotone and in disjunctive normal form. Q.E.D.

8 Comments

Several open problems remain, of which the most interesting is whether we can reduce the strength of the queries needed to get polynomial time identification of general (non-monotone) μ-formulas. In particular, do equivalence and membership queries suffice? Do subset and superset queries suffice?

9 Acknowledgements

This research was funded by the National Science Foundation, under grant number IRI-8718975.

References

[1] D. Angluin. *Equivalence queries and DNF formulas.* Technical Report, Yale University, YALEU/DCS/RR-659, 1988.

[2] D. Angluin. Queries and concept learning. *Machine Learning*, 2:319–342, 1987. Preliminary version appeared as YALEU/DCS/RR-479.

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

[4] L. Pitt and L. Valiant. Computational limitations on learning from examples. *J. ACM*, 35:965–984, 1988.

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