An Algorithm to Learn Read-Once Threshold Formulas, and some Generic Transformations between Learning Models

(Revised Version)

Nader H. Bshouty[1] Thomas R. Hancock[2]
Lisa Hellerstein[3] Marek Karpinski[4]

TR-93-037

July 1993

Abstract

We present a membership query (i.e. black box interpolation) algorithm for exactly identifying the class of read-once formulas over the basis of boolean threshold functions. We also present a catalogue of generic transformations that can be used to convert an algorithm in one learning model into an algorithm in a different model.

[1] Department of Computer Science, The University of Calgary, 2500 University Drive N.W., Calgary, Alberta, Canada T2N 1N4, bshouty@cpsc.ucalgary.ca, Research supported in part by the NSERC of Canada

[2] Siemens Corporate Research Inc., 755 College Road East, Princeton, NJ 08540, hancock@learning.scr.siemens.com, Research supported by ONR grant N00014-85-K-0445 and NSF grant NSF-CCR-89-02500. Research was done while the author was at Aiken Computation Laboratory, Harvard University

[3] Department of EECS, Northwestern University, 2145 Sheridan Road, Evanston, IL 60208-3118, hstein@eecs.nwu.edu, Research supported in part by a grant from the Siemens Corporation and ONR grant N00014-86-K-0454. Portions of this research were done while the author was visiting the Dept. of Computer Science, University of Bonn, the University of California at Santa Cruz, and the Massachusetts Institute of Technology

[4] Dept. of Computer Science, University of Bonn, 53117 Bonn, and International Computer Science Institute, Berkeley, California, marek@cs.bonn.edu, Research supported in part by the DFG Grant KA 673/4-1, by the ESPRIT BR Grant 7097, and by the Volkswagenstiftung Grant I/68055.

1 Introduction

In one of the simplest models of learning, the learner must exactly identify an unknown target function by asking membership queries. A membership query asks for the output of the function on an element of its domain. The query is answered by an infallible, honest oracle. This learning model is equivalent to standard black box interpolation where one substitutes inputs into a black box oracle computing a function from some class, and uses the observed outputs to deduce what the hidden function must be.

There are only a few non-trivial boolean concept classes that are known to be learnable in polynomial time with membership queries. One example is the class of (monotone) read-once formulas over the basis (AND, OR) [2]. A formula is read-once if no variable appears more than once. The basis of a formula is the set of operators allowed. In the first part of this paper we prove that a generalization of that class, the class of *monotone read-once threshold formulas*, is also learnable in polynomial time with membership queries (in this paper running time is measured as a function of the number of inputs to the target formula, n). This larger class consists of read-once formulas over the basis of boolean threshold functions. This basis includes the basis (AND, OR).

In the second part of the paper, we consider other learning models. We present a catalogue of generic transformations that can be used to convert an algorithm in one learning model (the *base model*) into an algorithm in a different model (the *target* model). Such transformations can be regarded as reductions, in that they reduce the problem of designing an algorithm in one learning model into the problem of designing an algorithm in another learning model.

Angluin, Hellerstein, and Karpinski [2] applied a generic transformation to convert their algorithm for exactly identifying (i.e. interpolating) the class of monotone read-once formulas over (AND,OR) into a polynomial time membership and equivalence query algorithm for the non-monotone case (read-once formulas over (AND,OR,NOT)). (An equivalence query allows the learner to propose a complete hypothesis for the unknown function, at which point an oracle either tells the learner that the hypothesis is correct, or else supplies a counterexample on which the two functions differ.) It is easily shown that there is no polynomial time membership query only algorithm for the non-monotone case.

In this paper we present a more extensive set of generic transformations. As in the [2] transformation, in these transformations we typically assume the technical condition that the classes being learned are projection closed (a class of functions is projection closed if taking any function in the class and "hard-wiring" a subset of input variables to specific values gives a function that is still in the class). In our transformations, either the base or target model has the property that the learning algorithm (in addition to being able to ask queries) is given as input a set of *justifying assignments* for the relevant variables. A justifying assignment for a variable X in a formula f is an assignment in which flipping the value of X would change the output of f. We say that algorithms in such models "use justifying assignments" (meaning that they receive them as input) just as we say that they use certain types of queries. Many of the transformations convert algorithms in models that use justifying assignments into algorithms in models that do not, and thus can be useful in some cases for simplifying the process of algorithm design.

Our interest in these transformations arises from the connections between the membership query algorithm presented in the first part of this paper, and a parallel line of research studying exact identification of read-once formulas in the more powerful membership and equivalence query model. The membership and equivalence query research began with the above mentioned algorithm for read-once formulas over the (AND,OR,NOT) basis [2] and culminated with a polynomial time algorithm to identify read-once formulas over the more general basis of arbitrary symmetric and constant fan-in gates [5]. Those results rely heavily on algorithms that are initially designed in the membership and justifying assignments model, and then converted to the membership and equivalence query model using a generic transformation (MJ→ME, described below) that is presented in this paper.

The algorithm of [5] includes as a subcase a polynomial time algorithm using membership and equivalence queries for learning (not necessarily monotone) read-once boolean threshold formulas, that is, read-once formulas over the basis of boolean threshold functions and negations. It is easily shown that for this class, and any of the non-monotone cases we discuss, there is no polynomial time learning algorithm using membership queries alone.

In this paper we present a generic transformation (MJ(U)→M(M), described below) that converts a membership query and justifying assignment algorithm for learning a *unate* class into a membership query only algorithm for the corresponding monotone class. A unate class is the generalization of a monotone class, where for each variable the formula may be either monotone in that variable or its negation. By applying this transformation to the membership and equivalence query algorithm for learning read-once threshold formulas, we get an algorithm that, like the one presented in the first part of this paper, learns monotone read-once threshold formulas using membership queries alone. However, the algorithm obtained by the transformation is a factor of n less efficient than the one presented in this paper, which is the most efficient known.

The first transformation we present, M(M)→MJ(U), is a variation of a simple transformation presented in [2]. It converts a membership query algorithm for learning a monotone class into a membership query and justifying assignments algorithm for learning the corresponding unate class. We then present a second transformation, MJ→ME which converts an algorithm for learning a class of formulas in the membership and justifying assignments model, into an algorithm for learning the same class of formulas in the membership and equivalence query model.

The main transformation presented in [2] is M(M)→ME(U) (Membership query algorithm for monotone class → Membership and equivalence query algorithm for unate class). By combining the M(M)→MJ(U) transformation with the MJ→ME transformation, we get an M(M)→ME(U) transformation which is slightly different from the one presented in [2]. As in [2], we also show that these transformations can be modified to include equivalence queries in the base and target models of the transformation (i.e. we argue that transformations of the form ME(M)→MJE(U), MJE→ME, and hence ME(M)→ME(U) all exist).

We then present the transformation MJ(U)→M(M), which is the reverse of the first transformation. This is also shown to hold when equivalence queries are allowed, so MJE(U)→ME(M) also exists.

In contrast, we show that the reverse of the MJ→ME transformation does not exist in general. We prove that a class of formulas presented presented by Angluin [1] can be learned in polynomial time in the model of membership and equivalence queries but not in the model of membership queries and justifying assignments.

We note that for some classes of formulas (including the restricted classes of read-once formulas discussed by Goldman, Kearns, and Schapire [6], and the arithmetic read-once formulas discussed by Bshouty, Hancock, and Hellerstein [4]), justifying assignments can be generated with high probability using random membership queries. For such classes, it is trivial to transform a learning algorithm using membership queries and justifying assignments into an algorithm that interpolates (with high probability) using deterministic and random membership queries.

The class of read-once formulas over the basis of boolean threshold and negation was previously studied by Heiman, Newman, and Wigderson [7] . They proved that each *non-degenerate* read-once formula over this basis expresses a unique function (this result follows implicitly from our learning algorithm), and proved bounds on the size of randomized decision trees computing such functions.

2 Basic Definitions

We view a formula as a rooted tree whose internal nodes are gates labelled by the function computed by the gate, and whose leaves contain variables or constants. The *basis* of a formula is the set of functions that appear as gates. A formula is *read-once* if no variable appears on more than one leaf. If a formula f is read-once, then for every pair of variables X and Y in f, there is a unique node farthest from the root that is an ancestor of both X and Y called their *lowest common ancestor*, which we write as $\mathrm{lca}(X, Y)$.

Let V_n denote the set $\{X_1, X_2, ..., X_n\}$. An assignment A to V_n can be denoted by giving the vector $[A(X_1), \ldots, A(X_n)]$, where $A(X_i)$ is the value assigned to X_i by A.

We say that a formula f is defined on the variable set V_n if the variables in f are a subset of V_n. If A is an assignment to the variables in V_n and f is defined on V_n, then we denote by $f(A)$ the output of the formula f when its inputs are set according to the assignment A.

If V' is any subset of V_n, $1_{V'}$ denotes the vector that assigns 1 to every element of V' and 0 to every element of $V_n - V'$.

For $X \in V_n$, let $A_{X \leftarrow k}$ denote the assignment B such that $B(Y) = A(Y)$ for all $Y \in V_n - \{X\}$, and $B(X) = k$. Let $A_{\neg X}$ denote the assignment B such that $B(Y) = A(Y)$ for all $Y \in V_n - \{X\}$, and $B(X) = \neg A(X)$.

If f is defined on V_n, A is an assignment to V_n, $X \in V_n$, and $f(A) \neq f(A_{\neg X})$, then A is *justifying* for X in f.

Let $V' \subseteq V_n$. We say that a formula f *depends on* the variables in V' if for every $X \in V'$, there is a justifying assignment for X in f.

A *partial assignment* P to V_n can be denoted by a vector $[P(X_1), \ldots, P(X_n)]$ where each $P(X_i) \in \{0, 1, *\}$. We say that a variable X_i in V_n is *assigned by* P if $P(X_i) \neq *$. If A is an assignment to V_n, and P is a partial assignment to V_n, then we denote by P/A the assignment C to V such that $C(X_i) = P(X_i)$ for all X_i such that $P(X_i) \neq *$, and $C(X_i) = A(X_i)$ for all X_i such that $P(X_i) = *$.

If a formula f is defined on V_n, then each partial assignment P to V_n induces a *projection* f_P of f which is the formula obtained from f by replacing by the appropriate constants those variables in f to which P assigns a value.

A set C of formulas defined over V_n is *projection closed* if for any partial assignment P to V_n and any $f \in C$, it is true that $f_P \in C$.

Let Th_k^m denote the boolean function on m variables which has the value 1 if at least k of the m variables are set to 1, and which has the value 0 otherwise. The *boolean threshold functions* are functions of the form Th_k^m. Note that Th_1^m computes the OR of m variables, and Th_m^m computes the AND of m variables. Thus the boolean threshold basis includes the basis (AND,OR).

A boolean formula is *monotone* if all of its gates compute monotone functions. A boolean formula is *unate* if all negations in the formula occur next to the variables, all (other) gates in the formula compute monotone functions, and for every variable X in the formula, either X always occurs with a negation, or it always occurs without a negation.

If f is any monotone boolean formula over V, let $U(f)$ denote the class of all formulas f' obtained from f by selecting a subset V' of V and replacing every occurrence of X_i in V' by $\neg X_i$. If M is a class of monotone boolean formulas, let $U(M)$ denote the union of $U(f)$ for all $f \in M$. All elements of $U(M)$ are unate, and we call $U(M)$ the *unate class corresponding to M.*

We define the class of *read-once threshold formulas* to be the class of read-once formulas over the basis of boolean threshold functions and negation. The class of *monotone read-once threshold formulas* consists of read-once formulas whose gates all compute functions of the form Th_k^m (no negations).

Because $\neg Th_k^m(X_1, X_2, ..., X_m) = Th_{m-k}^m(\neg X_1, \neg X_2, .., \neg X_m)$, it is possible to rewrite every read-once threshold formula so that all negations occur next to the variables. Thus every read-once threshold formula is equivalent to a unate read-once threshold formula. We will therefore assume, without loss of generality, that all read-once threshold formulas are unate. It follows that the unate class corresponding to the class of monotone read-once threshold formulas is the class of (not necessarily monotone) read-once threshold formulas.

Let f be a monotone boolean formula defined on V_n. A set of variables $S \subseteq V_n$ is a *minterm* of f if for every assignment A that assigns 1 to every variable in S we have $f(A) = 1$, and this property does not hold for any proper subset S' of S. A set $T \subseteq V_n$ of variables is a *maxterm* of f if for any assignment B that assigns 0 to all the variables in T we have $f(B) = 0$, and this property does not hold for any proper subset T' of T.

2.1 Identification with queries and justifying assignments

The learning criterion we consider is *exact identification.* There is a formula f called the *target formula*, which is a member of a class of formulas C defined over the variable set V_n. The goal of the learning algorithm is to halt and output a formula f from C that is logically equivalent to f.

In a *membership query*, the learning algorithm supplies an assignment A to the variables in V_n as input to a *membership oracle*, and receives in return the value of $f(A)$. Note that because $f_P(A) = f(P/A)$ it is possible to simulate a membership oracle for the projection f_P using a membership oracle for f.

In an *equivalence query*, the learning algorithm supplies a formula h from the class C as input to an *equivalence oracle*, and the reply of the oracle is either "yes", signifying that h is equivalent to f, or a *counterexample*, which is an assignment B such that $h(B) \neq f(B)$. The counterexample can be any such assignment B, and an algorithm that learns using equivalence queries is expected to perform properly no matter which counterexamples are produced.

A *set of justifying assignments* for a formula f contains, for every relevant variable X in f, a pair (X, A) such that A is a justifying assignment for X in f. When we say that an algorithm uses justifying assignments, we mean that the algorithm must be given as input a set of justifying assignments for the target function f.

3 Learning Monotone Read-Once Threshold Formulas

We present an algorithm that exactly learns monotone read-once threshold formulas in polynomial time using membership queries. We first describe some basic results about minterms and maxterms for monotone functions in general. We then sketch the algorithm based on a key subroutine. Following a section of technical lemmas, we described this subroutine. We conclude with a detailed description of the algorithm and with a theorem stating its correctness and complexity.

3.1 Finding Minterms and Maxterms

Our algorithm for learning monotone read-once threshold formulas makes repeated use of the standard greedy procedure (see e.g. [2]) for finding minterms of a monotone function f defined on V_n, using a membership oracle for f. This procedure *Findmin* takes as input a subset of variables $Q \subseteq V_n$ containing a minterm of f, and outputs a minterm contained in Q. We assume that $Q = \{X_{i_1}, X_{i_2}, \ldots, X_{i_m}\}$ where $i_1 < i_2 < \ldots < i_m$. It is significant for our proofs that this procedure tests the variables in Q in a fixed order.

```
Findmin(Q)

1. For $j := 1, \ldots, m$
   If $X_{i_j} \in Q$, but $f(1_{Q-\{X_{i_j}\}}) = 1$, then reset $Q$ to $Q - \{X_{i_j}\}$
2. Return $Q$.
```

We also make use of the dual procedure *Findmax* to find a maxterm contained in $Q \subseteq V_n$.

3.2 Minterm and Maxterm intersections

Any pair of a minterm S and a maxterm T must clearly intersect in some variables (lest we get the contradiction that the formula must evaluate to both 1 and 0 on any assignment that sets all the variables in S to 1 and all the variables in T to 0). We now prove another simple fact about minterm and maxterm intersections.

Lemma 1 *For any monotone function g, if S is a minterm of g, and X is a variable in S, then there exists a maxterm T of g such that $T \cap S = \{X\}$. Dually, if T is a maxterm, and X is in T, then there exists a minterm S of g such that $T \cap S = \{X\}$.*

Proof: Consider the set $(V_n - S) \cup \{X\}$. This set intersects S because it contains X. Every other minterm S' of g must contain an element not in $S - \{X\}$, because otherwise S' is a subset of S violating the minimality condition for S. Therefore

$(V_n - S) \cup \{X\}$ intersects every minterm, implying that $(V_n - S) \cup \{X\}$ must contain a maxterm (or equivalently, the assignment that sets just those variables to 0 satisfies no minterms, and hence f is 0 on that assignment, which by definition means that those variables include a maxterm). That maxterm T must intersect S, which is possible only if $T \bigcap S = \{X\}$. The dual is proved analogously. □

Note that this lemma is to a certain extent constructive, since given X and the minterm S (or maxterm T), we can find the maxterm T (or minterm S) by the appropriate call to *Findmin* (*Findmax*).

3.3 Outline of the Algorithm

The algorithm recursively constructs the target formula f according to a depth first traversal of the formula tree. More precisely, suppose the root of f computes Th_k^m and that the inputs to the root are the outputs of the subformulas $f_1, f_2, \ldots, f_m$. The algorithm first builds some f_{i_1} (recursively), then some f_{i_2}, etc. In the process, it discovers the values of m and k.

The algorithm is based on the following observation. To learn f_1 (say) recursively we need to use our membership oracle for f to simulate a membership oracle for f_1. We can do this by using a partial assignment P that assigns values to enough of the variables appearing in f_2,...,f_m to project all those extraneous subformulas to be constant (1 or 0) and that furthermore projects exactly $k - 1$ of them to be 1 and the remaining $m - k$ to 0. As long as P assigns values to no variables in f_1 it follows that $f_1 \equiv f_P$. As mentioned above we can simulate a membership query for $f_1 = f_P$ by the rule $f_P(A) = f(P/A)$. The most difficult part of the algorithm is constructing such a partial assignment P.

We now sketch the algorithm. Assume without loss of generality that the target formula f contains no constants in its leaves, and that it is *non-degenerate*, in the sense that there are no adjacent AND gates or adjacent OR gates along a root leaf path. Any monotone read-once threshold formula can be rewritten to satisfy these conditions.

To begin, the algorithm generates a minterm S (by calling *Findmin* with the set of all variables). It then picks an $X \in S$ and finds a maxterm T by calling *Findmax*$((V_n - S) \cup \{X\})$ (as discussed in the proof of Lemma 1, this guarantees $S \cap T = \{X\}$).

The algorithm checks whether $S = T = \{X\}$. It is easily verified that this occurs iff f is identically equal to the variable X. In this case (the base case) the algorithm stops and outputs X.

Suppose that we are not at the base case, and that the root of f computes Th_k^m as above. Without loss of generality, assume variable X appears in f_1. Because f is read-once, S is composed of minterms of exactly k of $f_1, f_2, \ldots, f_m$ (including f_1). Without loss of generality, assume S is composed of the minterms of $f_1, f_2, \ldots, f_k$. T is composed of the maxterms of $m-k+1$ of $f_1, f_2, \ldots, f_m$. Because $S \cap T = \{X\}$, T does not contain maxterms of $f_2, f_3, \ldots, f_k$ (lest otherwise its intersection with S would include some variables from these subformulas). It follows that T contains maxterms of $f_{k+1}, f_{k+2}, \ldots, f_m$, and of f_1.

Suppose we can discover the subsets of variables $S' \subset S$ and $T' \subset T$ that do not appear in f_1. Then by setting P to assign the variables in S' to 1 and the variables in T' to 0 (while leaving $V_n - (S' \cup T')$ unassigned), we have the projection needed to recursively learn the subformula f_1. In Section 3.5 we describe subroutines *LcaRootS* and *LcaRootT* that do exactly this. Subroutine *LcaRootT* takes as input a variable X, a minterm S and a maxterm T such that $S \cap T = \{X\}$. It outputs the set of variables $Y \in T - \{X\}$ such that $lca(X, Y)$ is the root of f. This is precisely the subset T' of variables in T that do not appear in f_1. Subroutine *LcaRootS* is the dual that outputs the analogous subset S' of S.

Using these two routines, the algorithm finds f_1 recursively by simulating calls to the membership oracle for f_1 using the oracle for f. The algorithm then finds the subformulas $f_2, f_3, \ldots, f_k$ as follows. Until all variables in $S - \{X\}$ have appeared in some recursively generated subformula, the algorithm executes the following loop. First it picks some arbitrary Y in $S - \{X\}$, such that Y has not yet appeared in a recursively generated subformula of f. Let $f' \in \{f_2, \ldots, f_k\}$ be the subformula containing Y. The algorithm runs Findmax$((V_n - S) \bigcup Y)$ to generate a maxterm T_Y such that $S \cap T_Y = \{Y\}$. It then uses *LcaRootT* and *LcaRootS* on S and T_Y (as it did with S and T) to find a projection of f that is equal to f'. As the final step of the loop, the algorithm finds f' recursively. By counting the number of iterations of this loop, the algorithm learns the value of k.

In a dual way, the algorithm recursively generates $f_{k+1}, \ldots, f_m$ and learns the value of $m - k$.

The algorithm ends by outputting the formula $Th_k^m(f_1, f_2, \ldots, f_m)$.

The above description is fairly complete, except that it omits the description of *LcaRootS* and *LcaRootT*. In the following two sections, we discuss those routines. The correctness of the entire algorithm then follows easily.

3.4 A pair of technical lemmas

The routines *LcaRootS* and *LcaRootT* are based on two technical lemmas. The lemmas describe properties of the minterms and maxterms generated by our algorithm using *Findmin* and *Findmax*.

In the first lemma, we prove the following. Let V' be a subset of the variables of f. Suppose *Findmin*(V') is run with an oracle for f, and produces a minterm that includes as a subset a minterm for some subformula g of f. Then *Findmin*(V') would produce exactly that subset as a minterm if it were run with a membership oracle for g rather than f. Note that the membership oracle for g ignores any settings to variables not appearing in g. The dual lemma for maxterms also holds.

The first lemma has the following immediate and important consequence. Suppose we call *Findmin* twice with input sets Q_1 and Q_2 that differ, but include the same subset of variables from some particular subformula g. Suppose further that the output of these calls are two minterms that include variables from g. Then the two minterms will include the same set of variables from f' — the set generated by running Findmin on either Q_1 or Q_2 with the oracle for g.

We indicate a call to *Findmin* using a membership oracle for a formula g, by writing $Findmin^g$.

Lemma 2 *Let f be a monotone read-once threshold formula defined on the variable set V_n. Let g be a subformula of f, and let Z be the set of variables appearing in g. Let V' be a subset of V_n such that $Z \subseteq V'$. Let S be the minterm of f output by $Findmin^f(V')$. If $S \cap Z \neq \emptyset$, then $S \cap Z$ is the minterm of g output by $Findmin^g(Z)$.*

Proof: Assume $S \cap Z \neq \emptyset$.

In order to show the lemma, it suffices to show the following two facts.

1. $Findmin^f(V')$ tests the variables of Z in the same order as $Findmin^g(Z)$
2. For every X_i in Z, the output of the membership query in $Findmin^f(V')$ that tests X_i is the same as the output of the membership query in $Findmin^g(Z)$ that tests X_i.

Fact 1 follows immediately from the definition of *Findmin*, which specifies that the variables in the input set are tested in increasing order of their indices.

Fact 2 follows from an observation and a claim. The set $Q \subseteq V'$ is the set of variables that is revised during the loop in $Findmin\, f(V')$ and is eventually output as minterm S. Let $S_j \subseteq V'$ be the set Q at the beginning of the jth iteration of the loop in $Findmin^f(V')$, which tests whether X_{i_j} should be included in the output minterm S. The observation is that if $X_{i_j} \notin Z$, then the value of $S_j \cap Z$ is the same as the value of $S_{j+1} \cap Z$ after the iteration. Thus the value of $Q \cap Z$ remains unchanged while *Findmin* tests variables not in Z.

The claim is that if $X_{i_j} \in Z$, then $f(1_{S_j - \{X_{i_j}\}})$ (i.e. the value returned by the membership query in the jth iteration of the loop) is equal to $g(1_{S_j \cap Z - \{X_{i_j}\}})$. A simple inductive argument combining the observation and the claim proves Fact 2.

We now prove the claim. By assumption, $S \cap Z \neq \emptyset$. Because g is a subformula of f, f is read-once, and S is a minterm of f, S must contain exactly one minterm of g. After every iteration of the loop in $Findmin^f(V')$, S_j contains a set which is a superset of S. S contains a minterm of g, and therefore $g(1_S) = 1$. By monotonicity, $g(1_{S_j}) = 1$ after every iteration of the loop. If $f(1_{S_j - \{X_{i_j}\}}) = 1$, then X_{i_j} is removed from S_j to form S_{j+1}. Therefore, $f(1_{S_j - \{X_{i_j}\}}) = 1$ implies that $g(1_{S_j - \{X_{i_j}\}}) = g(1_{S_j \cap Z - \{X_{i_j}\}}) = 1$.

Conversely, suppose $g(1_{S_j \cap Z - \{X_{i_j}\}}) = 1$. Then $g(1_{S_j - \{X_{i_j}\}}) = 1$. The assignment $1_{S_j - \{X_{i_j}\}}$ is obtained from the assignment 1_{S_j} by changing the setting of the variable X_{i_j} from 1 to 0. Since $g(1_{S_j}) = g(1_{S_j - \{X_{i_j}\}}) = 1$, changing the assignment of X_{i_j} in 1_{S_j} from 1 to 0 does not affect the output of g. The formula f is read-once, and g is a subformula of f, so changing the assignment of X_{i_j} in 1_{S_j} from 1 to 0 does not affect the output of f either. Therefore $f(1_{S_j - \{X_{i_j}\}}) = 1$. □

The second lemma is more complicated. It states that if minterms S and S_Y both intersect maxterm T in only one variable (X and Y respectively), and are generated by *Findmin* and *Findmax* on sets with particular properties, then the only difference between S and S_Y is in variables that appear in the two subformulas of $lca(X, Y)$ that contain X and Y. Specifically, S will contain a minterm for X's subformula and no variables from Y's, whereas the reverse will be true for S_Y. Thus, given S and S_Y we can determine which variables of S appear in the subformula of lca(X,Y) containing X, and which don't. Again, the dual result holds (flipping "min" and "max").

Lemma 3 *Let f be a monotone read-once threshold formula defined on the variable* *set V_n. Let T be a maxterm of f. Let S be the output of $Findmin^f(V')$ for some $V' \supset V_n - T$ and suppose $S \cap T = \{X\}$. If $Y \in T - \{X\}$ and S_Y is the minterm output by $Findmin^f((V_n - T) \cup \{Y\})$, then*

1) *$S_Y - (S_Y \cap S)$ is a minterm of the subformula rooted at the child of $lca(X, Y)$ containing Y.*

2) *$S - (S \cap S_Y)$ is a minterm of the subformula rooted at the child of $lca(X, Y)$ containing X.*

Proof: Consider a gate on the path from X to the root. Suppose the gate computes Th_k^m. T contains maxterms of exactly $m - k + 1$ of the m subformulas whose outputs are inputs to this gate, including the subformula containing X. S contains a minterm of the subformula containing X, and of the remaining $k - 1$ subformulas for which T does not contain a maxterm.

Similarly, if we consider a gate on the path from Y to the root computing Th_k^m, T will contain maxterms of exactly $m - k + 1$ of the m subformulas, including the subformula containing Y. S_Y will contain a minterm of the subformula containing Y, and of the remaining $k - 1$ subformulas for which T does not contain a maxterm.

A minterm for a read-once formula contains for any particular subformula either a minterm of that subformula or no variables from the subformula. Since S contains a minterm for the subformula of $lca(X, Y)$ containing X, and S_Y does not, it follows that $S - (S \cap S_Y)$ will indeed contain a minterm for the subformula of $lca(X, Y)$ containing X (and analogously for $S_Y - (S_Y \cap S)$). To prove the lemma we now show that every other variable in S is also in S_Y, and vice versa.

Let G be a gate which is on the path from $lca(X, Y)$ to the root such that G is not equal to $lca(X, Y)$. S and S_Y contain minterms of the same set of subformulas (rooted at children of G). Since T contains no variables from the subformulas in this set that do not contain X and Y, V' and $(V_n - T) \cup \{Y\}$ include all the variables from such subformulas. Since S and S_Y were created by calling *Findmin* with these two sets of variables respectively, By Lemma 2, S and S_Y will contain exactly the same minterms of these subformulas (for each such subformula g, both S and S_Y include the minterm one would obtain by calling $Findmin^g(V_n)$) Similarly, S and S_Y will contain exactly the same minterms of the subformulas rooted at children of $lca(X, Y)$ that do not contain X or Y, and for which T does not contain a maxterm.□

3.5 Main lemma for the subroutines

The routine *LcaRootT* is based on the following lemma. The lemma describes a criterion for determining, given a minterm and maxterm intersecting in a single variable X (and generated in the manner described in the two technical lemmas), which variables in the maxterm appear in the same subformula of the root as does X. A dual lemma (on which *LcaRootS* is based) also holds.

Lemma 4 *Let f be a monotone read-once threshold formula defined on the variable set V_n. Let T be a maxterm of f. Let S be the output of $Findmin^f(V')$ for some $V' \supseteq V_n - T$ and suppose $S \cap T = \{X\}$. For all Y in $T - \{X\}$, let S_Y be the minterm output by $Findmin^f((V_n - T) \cup \{Y\})$. If there exists a Y in $T - \{X\}$ such that $S \cap S_Y$ is empty, then*

$$\{Y \in T - \{X\} |\ lca(X,Y) = root\ of\ f\} = \{Y \in T - \{X\} |\ S_Y \cap S = \emptyset\}.$$

If there is no Y in $T - \{X\}$ such that $S \cap S_Y$ is empty, then

$$\{Y \in T - \{X\} | lca(X,Y) = root\ of\ f\} =$$
$$\{Y \in T - \{X\} | \forall Z \in S - (S_Y \cap S),\ S \cup S_Y - \{Z\} contains\ a\ minterm\}.$$

Proof: There are two cases.

- Case 1: The root of f is an OR.

  In this case there is at least one variable Y in $T - \{X\}$ such that $lca(X,Y)$ is the root. S_Y is a minterm of the subformula that contains Y and is rooted at a child of the root of f. It follows that $S \cap S_Y = \emptyset$.

  Now let Y be a member of $T - \{X\}$ such that $lca(X,Y)$ is not the root. Let G be the gate which is the child of the root, on the path from X to the root. The gate G is also on the path from Y to the root. Since f is a (non-degenerate) monotone read-once threshold formula, G is not an OR gate. Therefore, there exists a subformula h rooted at a child of G such that T does not contain a maxterm of h. Clearly V' and $(V_n - T) \cup \{Y\}$ both contain all the variables of h. Since $S \cap T = \{X\}$, S must contain minterms of all the subformulas of G for which T does not contain a maxterm, and hence S contains a minterm of h. Similarly, S_Y contains a minterm of h. By Lemma 2, S and S_Y will contain the same minterm of h, and therefore $S \cap S_Y$ is not empty.

- Case 2: Root of f is not an OR.

  Suppose the root is Th_k^m $(k \neq 1)$. By the same reasoning as in the second part of Case 1, for all Y in $T - \{X\}$, $S \cap S_Y$ is not empty.

  If $lca(X,Y) =$ root, then for all Z in $S \cap S_Y$, $S \cup S_Y - \{Z\}$ contains a minterm, because setting $S \cup S_Y$ to 1 forces $k+1$ of the subformulas rooted at children of the root of f to 1.

  If $lca(X,Y)$ is not the root, then setting $S \cup S_Y$ to 1 forces exactly k of the subformulas rooted at children of the root to be 1. By Lemmas 2 and 3, $S \cap S_Y$ must contain a minterm of some subformula h rooted at a child of the root of f, such that h does not contain X (or Y). Let Z be a variable in the minterm of h contained in $S \cap S_Y$. Setting $S \cup S_Y - \{Z\}$ to 1 will force only $k-1$ of the wires into the root to 1, because $S \cup S_Y - \{Z\}$ does not contain a minterm of h. Therefore $S \cup S_Y - \{Z\}$ does not contain a minterm of f. $\Box$

We present the basic subroutines *LcaRootS* and *LcaRootT*, and then we present the complete algorithm.

3.6 *LcaRootT* and *LcaRootS*

LcaRootT takes as input a minterm S, a maxterm T, and a variable X, such that $S \cap T = \{X\}$, and S is the output of $Findmin^f(V')$, where $V' \supseteq V_n - T$. Using the criterion of Lemma 4, it computes and then outputs the set of variables Y in $T - \{X\}$ such that $lca(X,Y)$ is the root of f.

```
$LcaRootT^f(S,T,X)$

1. for all $Y$ in $T - \{X\}$        $S_Y := Findmin^f((V_n - T) \cup \{Y\})$.
2. if there exists a $Y$ in $T - \{X\}$ such that $S \cap S_Y$ is empty then        return( $\{Y \in T - \{X\} | \; S_Y \cap S = \emptyset\}$ ).
3. $Q := \emptyset$

       for all $Y$ in $T - \{X\}$ do
           for all $Z$ in $S \cap S_Y$ do
               if $f(1_{S \cup S_Y - \{Z\}}) = 0$ then
                   $Q := Q \cup \{Y\}$.
```

```
4. return( $T - \{X\} - Q$)
```

The dual subroutine, *LcaRootS*, finds the set of Y in $S - \{X\}$ such that $lca(X, Y)$ is the root of f.

3.7 The algorithm

```
$MROTLearn^f$

1. $S := Findmin^f(V_n)$
2. Pick an $X$ in $S$.
   $T := Findmax^f((V_n - S) \cup \{X\})$
3. if $S = T = \{X\}$, then return($X$) (the formula $f$ is equal to $X$)
4. $T' := LcaRootT^f(S, T, X)$
5. $S' := LcaRootS^f(S, T, X)$
6. (a) $k := 1$ (counts number of inputs to root of $f$ set to 1 by a minterm of $f$)
   (b) $j := 1$ (counts number of inputs to root of $f$ set to 0 by a maxterm of $f$)
   (c) $Q := S'$
   (d) $R := T'$
   (e) Let $f_1$ be the projection of $f$ induced by setting the variables in $S'$
       to 1, and the variables in $T'$ to 0. Recursively learn $f_1$ by running
       $MROTLearn^{f_1}$, simulating calls to the membership oracle of $f_1$ with
       calls to the membership oracle of $f$.
7. while $Q \neq \emptyset$ do
   (a) Pick an $X'$ in $Q$.
   (b) $T_{X'} := Findmax^f((V_n - S) \cup \{X'\})$
   (c) $S' := LcaRootS^f(S, T_{X'}, X')$
   (d) $k := k + 1$.
   (e) $Q := Q \cap S'$.
```

```
    (f) Let $f_k$ be the projection of $f$ induced by setting the variables in $S'$
        to 1, and the variables in $T'$ to 0. Recursively learn $f_k$ by running
        $MROTLearn^{f_k}$, simulating calls to the membership oracle of $f_k$ with
        calls to the membership oracle of $f$.
8. while $R \neq \emptyset$ do
    (a) Pick an $X'$ in $R$.
    (b) $S_{X'} := Findmin^f((V_n - T) \cup \{X'\})$
    (c) $T' := LcaRootT^f(S'_X, T, X')$.
    (d) $j := j + 1$.
    (e) $R := R \cap T'$.
    (f) Let $f_{k+j-1}$ be the projection of $f$ induced by setting the variables in
        $S'_X \cap S$ to 1, and the variables in $T'$ to 0. Recursively learn $f_{k+j-1}$ by
        running $MROTLearn^{f_{k+j-1}}$, simulating calls to the membership oracle
        of $f_{k+j-1}$ with calls to the membership oracle of $f$.
9. Output the formula $Th_k^{k+j-1}(f_1, f_2, f_3, \cdots, f_{k+j-1})$
```

3.8 Correctness and complexity

Theorem 1 *There is a learning algorithm that exactly identifies any monotone read-once threshold formula in time $O(n^3)$ using $O(n^3)$ membership queries.*

Proof: The correctness of our algorithm follows from the discussion in the previous sections.

The routines *Findmin* and *Findmax* each take time $O(n)$ and make $O(n)$ queries. The routine *LcaRootT* makes $O(n^2)$ queries and can be implemented to run in time $O(n^2)$ (this includes the calls to *Findmin*).

The complexity of the main algorithm can be calculated by "charging" the costs of the steps to the edges and nodes of the target formula f. In each execution of $MROTLearn^f$, we charge some of the steps to f, and some of the steps to the edges joining the root to its children. Recursive calls to $MROTLearn^{f_k}$ are charged recursively to the subformula f_k.

More specifically, we charge steps 1 - 6(d), step 9, and the checking of the loop conditions in steps 7 and 8, to the root of f. We recursively charge calls to $MROTLearn^{f_k}$ in steps 6(e), 7(f), and 8(f) to the subformulas f_k. For each iteration of step 7, we charge steps 7(a) through 7(e) to the edge leading from the root of f to the root of the subformula f_k defined in step 7(f). Similarly, for each iteration of step 8, we charge steps 8(a) through 8(e) to the edge leading from the root of f to the root of the subformula f_{k+j-1} defined in step 8(f). Thus at each execution of $MROTLearn^{f}$, we charge time $O(n^2)$ to the root of f and $O(n^2)$ membership queries to the root of f. We also charge time $O(n^2)$ and $O(n^2)$ membership queries to each of the edges joining the root of f to its children.

The total number of nodes in f is $O(n)$, and the total number of edges is $O(n)$. Therefore the algorithm takes time $O(n^3)$ and makes $O(n^3)$ queries. □

Corollary 1.1 *There is a learning algorithm that exactly identifies any read-once threshold formula in time $O(n^4)$ using $O(n^4)$ membership queries and $O(n)$ equivalence queries.*

Proof: The class of read-once threshold formulas is the unate extension of the class of monotone read-once threshold formulas. The theorem follows directly from the results of Angluin, Hellerstein and Karpinski [2], who showed that if a class M can be learned in time $O(n^\alpha)$ with $O(n^\beta)$ membership queries, then the corresponding unate class can be learned in time $O(n^{\alpha+1})$ with $O(n^{\beta+1})$ membership queries, and $O(n)$ equivalence queries. □

4 Transformations

4.1 M(M)→MJ(U)

In this section we describe a simple method for converting an algorithm that identifies a monotone class using membership (and possibly equivalence) queries into an algorithm that learns the corresponding unate class using membership (and possibly equivalence) queries and justifying assignments.

Let f be a unate formula. Since f is unate, all negations in f appear at the leaves. If X is a variable in f and X is negated in f (i.e. there is a negation appearing next to all occurrences of X in f) then we say the sign of X is negative in f. Otherwise, we say the sign of X is positive. If the sign of X is negative, then (because f is unate) for all assignments A, $f(A_{X\leftarrow 0}) \geq f(A_{X\leftarrow 1})$. If the sign of X is positive, then for all assignments A, $f(A_{X\leftarrow 0}) \leq f(A_{X\leftarrow 1})$.

Given a unate formula f, and a set of justifying assignments for the variables in f, we can determine the signs of the variables in f using membership queries as follows. Let A be a justifying assignment for a variable X in f. Because A is justifying for X, the values of f on $A_{X\leftarrow 0}$ and $A_{X\leftarrow 1}$ are distinct. Therefore, if $A_{X\leftarrow 0} = 1$ then the sign of X is negative, and if $A_{X\leftarrow 0} = 0$ then the sign of X is positive. To determine the sign of X, it therefore suffices to ask the membership query $A_{X\leftarrow 0}$.

In [2], a simple transformation is presented. It converts an algorithm for learning a monotone class with membership queries into an algorithm for learning the corresponding unate class with membership queries, given the signs of the variables in the target formula as input. The transformation works by "replacing" the negated variables in the target formula with new, non-negated variables, and using the membership query algorithm to learn the resulting monotone formula. A simple variation, handling equivalence queries, is also described.

By combining the procedure described above for finding the signs of the variables, with the simple transformation from [2] which takes the signs as input, we get a method for transforming an algorithm that identifies a monotone class using membership (and equivalence) queries, into one that identifies the corresponding unate class using membership (and equivalence) queries and justifying assignments.

We thus have the following theorem. (Unlike our other transformations, the projection closed condition is not required here.)

Theorem 2 *Let M be a monotone class of formulas. If M can be exactly identified in polynomial time by an algorithm using membership queries then $U(M)$ can be exactly identified in polynomial time by an algorithm using membership queries and justifying assignments. Furthermore, if M can be exactly identified in polynomial time by an algorithm using membership and equivalence queries, then $U(M)$ can be exactly identified in polynomial time by an algorithm using membership queries, equivalence queries, and justifying assignments.*

The overhead of this transformation is an additional $O(n)$ in running time and queries, plus an additional constant amount of time whenever a bit is set for a membership query.

4.2 MJ→ME

In this section we describe how to convert an algorithm that identifies a projection closed class using membership queries and justifying assignments into an algorithm that learns the class using membership and equivalence queries.

Let AlgMJ be an algorithm using membership queries and justifying assignments. The transformation algorithm, ToME(AlgMJ), which uses membership and equivalence queries, is based on the following loop. At the start of the loop we have a subset $V' \subset V_n$, and we have a partial assignment P assigning values to the variables in $V_n - V'$ and leaving the variables in V' unassigned. For every $X \in V'$, we know a justifying assignment for X in f_P. For the first iteration of the loop, we set V' to empty, and we set P to an arbitrary assignment to V_n (so f_P is constant).

We run AlgMJ to learn f_P (simulating a membership oracle for f_P with the membership oracle for f). The output of AlgMJ is a formula $g \equiv f_P$. We ask the equivalence query "$g \equiv f$?" (i.e. $f_P \equiv f$?). If the answer is "yes" (which it will be when V' contains all the relevant variables of f), we are done. If the answer is "no" the equivalence oracle returns a counterexample A. We call a routine *ProjBitFlip* (described below) that returns a new, larger subset V' of V_n, a new associated partial assignment P, and justifying assignments (with respect to f_P) for the variables in the new V'. We then repeat the loop. Termination of ToME(AlgMJ) is guaranteed by the fact that at each iteration we increase the size of V'.

Routine *ProjBitFlip* takes as input V', the partial assignment P, the counterexample A (an assignment on which f_P differs from f), and a set S of justifying assignments for the variables in V' (the assignments in S are justifying with respect to f and f_P). The key processing in *ProjBitFlip* is a loop to greedily reduce the number of variables on which A and P differ.

ProjBitFlip(V',P,A, S)

1. Set B to A. Set W to $\{X \in V_n - V' | \; B(X) \neq P(X)\}$.
2. While there exists an $X \in W$ such that $f(B_{\neg X}) = f(B)$, set B to $B_{\neg X}$ and delete X from W.
3. Let P' be the partial assignment such that $P'(X) = P(X)$ for all $X \in V_n - W$, and $P'(X) = *$ for all $X \in W$.
4. Let $S' = S \bigcup \{(X, B) | X \in W\}$.

5. Output $V' \bigcup W$, P', and S'.

The processing in *ProjBitFlip* does not change either $f(B)$ or $f_P(B)$, so for the final B those two values still differ. This means $B \neq P/B$ and therefore W is not empty. Our new variable set is $V' \bigcup W$, and P' assigns $*$ to all variables in W. Assignment B is justifying for those new variables in $f_{P'}$ and in f.

For completeness, we present the transformation algorithm ToME(AlgMJ).

ToME(AlgMJ)

1. Let $V' = \emptyset$. Let $S = \emptyset$. Initialize P to an arbitrary assignment defined on V_n (giving values to all variables).
2. Do forever:
   (a) Call the procedure AlgMJ using the justifying assignments S, and simulating membership queries to the function f_P. Let g be the formula returned.
   (b) Make an equivalence query with g. If the reply is "yes" then output g and halt, otherwise, let B be the counterexample.
   (c) In this case, $g(B) = f_P(B) \neq f(B)$, that is, $f(P/B) \neq f(B)$. Call *ProjBitFlip*(V', P, B, S). Set V', P and S to the values W, P' and S' returned by *ProjBitFlip*.

Applying techniques from [2], we show that a modified version of ToME works if the input algorithm uses equivalence queries as well as membership queries and justifying assignments. The modification is mainly to step 2a, where we are simulating AlgMJ to learn f_P, even though we only have oracles for f. If AlgMJ asks an equivalence query "Does $g = f_P$?", we can do the following:

- Ask the equivalence oracle for f, "Does $g = f$?" If the reply is "yes", then halt and output g. Otherwise, the reply is no and a counterexample B. Ask the membership oracle for f for the value of $f(P/B)$ (which equals $f_P(B)$). Evaluate $g(B)$. If $g(B) \neq f_P(B)$, then B is a counterexample to the query asked by AlgMJ "Does $g = f_P$?", so continue the simulation by returning the counterexample B. If $g(B) = f_P(B)$, then $f_P(B) \neq f(B)$. Halt the simulation, and jump to step 2c (because we already have an assignment B such that $f_P(B) \neq f(B)$, which is what we need in step 2c to call *ProjBitFlip* and expand the set V').

This discussion proves the following theorem.

Theorem 3 *If C is a projection closed class of formulas such that C can be exactly identified in polynomial time by an algorithm using membership queries, equivalence queries, and justifying assignments, then C can be exactly identified in polynomial time by an algorithm using membership and equivalence queries.*

This transformation adds $O(n)$ equivalence queries, and produces a membership and equivalence algorithm whose running time and number of membership and equivalence queries are each a factor of $O(n)$ greater than for the original (transformed) algorithm.

4.3 ME→MJ does not exist

We describe a class of monotone DNF formulas (used originally by Angluin [1]) for which there exists a polynomial time membership and equivalence query learning algorithm, but for which no polynomial time membership query and justifying assignment algorithm exists.

The class consists of monotone DNF formulas defined over the variable set $V_n = \{X_1, ..., X_n\}$ and consisting of $m + 1$ terms for $m = n/2$. The first m terms of each formula are $X_1 \bigwedge X_2$, $X_3 \bigwedge X_4$, $\ldots$, $X_{n-1} \bigwedge X_n$. The last term is made up of exactly one variable from each of the first m terms. Angluin showed that this class is learnable (by the membership and equivalence query algorithm for learning monotone DNF), but it is not learnable with membership queries. The following simple argument shows that it is also not learnable with membership queries and justifying assignments.

Consider the following set of assignments – the assignment setting $X_1 and X_2$ to 1 and the other variables to 0, the assignment setting X_3 and X_4 to 1 and the other variables to 0, the assignment setting X_4 and X_5 to 1 and the other variables to 0, and so forth. For every function in the above class, this is a set of justifying assignments (e.g. the first assignment in the set is justifying for the variables X_1 and X_2). If a learner was attempting to identify a target formula from the above class, an adversary could provide this same set of justifying assignments. The set would give no information about the target formula (the learner could even generate it independently). Therefore, because the class cannot be learned in polynomial time with membership queries, it cannot be learned in polynomial time with membership queries and justifying assignment.

We note that although the above class is not projection closed, it is easy to show that the same result holds for its closure under projections.

4.4 MJ(U)→M(M)

In this section we describe how to convert an algorithm that identifies a projection closed unate class using membership queries and justifying assignments into an algorithm that learns the corresponding monotone class using just membership queries.

To perform this transformation, we need to show how we can generate justifying assignments for the variables in a monotone class using membership queries.

The transformation is based on the following observation. Suppose f is a monotone formula defined on the variable set V_n. Let V' be a subset of the variables in V_n. Let P_0 be the partial assignment setting the variables in $V_n - V'$ to 0 and leaving the variables in V' unassigned. Let P_1 be the partial assignment setting the variables in $V_n - V'$ to 1 and leaving the other variables in V' unassigned. Suppose there is a relevant variable X in $V_n - V'$. Let A be a justifying assignment for that variable such that $f(A) = 0$. Then, because f is monotone, $A(X) = 0$ and $f(A_{X \leftarrow 1}) = 1$. It also follows from the monotonicity of f that $f_{P_0}(A) = f(P_0/A) \leq f(A)$ and $f_{P_1}(A) = f(P_1/A) \geq f(A)$. Therefore, $f_{P_0}(A) = 0$ and $f_{P_1}(A) = 1$ and $f_{P_0} \neq f_{P_1}$. Thus if $V_n - V$ contains a relevant variable of f, then $f_{P_0} \neq f_{P_1}$. In contrast, it is clear that if $V_n - V'$ contains no relevant variables of f, then $f_{P_0} \equiv f_{P_1}$.

Let AlgMJU be an algorithm for learning a unate class C' of formulas using membership queries and justifying assignments. We present a transformation algorithm ToMM(AlgMJU) which learns the corresponding monotone class C of formulas using only membership queries.

The transformation algorithm is based on the following loop. At the start of the loop, we have a subset V' of V_n and a set S of justifying assignments for the variables in V' (the assignments are justifying with respect to f). With V' we associate the projections f_{P_0} and f_{P_1} as above, where P_0 and P_1 are partial assignments obtained by setting the variables in $V_n - V'$ all to 0 and all to 1, respectively. We check to see whether for all (X, A) in S, A is also justifying for X in f_{P_0} and f_{P_1}. If this is not the case, then we will expand V' and S as follows. Note that because A is justifying for X in f, $f(A_{X\leftarrow 0}) = 0$ and $f(A_{X\leftarrow 1}) = 1$, and hence by monotonicity $f_{P_0}(A_{X\leftarrow 0}) = 0$ and $f_{P_1}(A_{X\leftarrow 1}) = 1$. It follows that if A is not justifying for X in f_{P_0}, then $f_{P_0}(P_1/A_{X\leftarrow 1}) = f_{P_0}(A_{X\leftarrow 1}) = f_{P_0}(A_{X\leftarrow 0}) = 0$, whereas we know that $f(P_1/A_{X\leftarrow 1}) = f_{P_1}(A_{X\leftarrow 1}) = 1$. Hence we can run *ProjBitFlip* with partial assignment P_0 and assignment $P_1/A_{X\leftarrow 1}$ to find a justifying assignment for a variable in Y in $V_n - V'$. We then add Y to V', add Y and its justifying assignment to S, and go back to the start of the loop. Similarly, if A is not justifying for X in f_{P_1}, then $f(P_0/A_{X\leftarrow 0}) = 0$ and $f_{P_1}(P_0/A_{X\leftarrow 0}) = 1$, and we can run *ProjBitFlip* with partial assignment P_1 and assignment $P_0/A_{X\leftarrow 0}$.

When we reach the point where all the assignments in S are justifying for both f_{P_0} and f_{P_1}, we run two parallel simulations of AlgMJU with the set S of justifying assignments as input. In one simulation we answer a membership queries by simulating a membership oracle for f_{P_0}, and in the other simulation we answer membership queries by simulating a membership oracle for f_{P_1}. If the simulations terminate without ever diverging (doing anything different) then the two simulations will output the same formula g, meaning that $g \equiv f_{P_1} \equiv f_{P_0} \equiv f$. In this case we halt and output g. If the two simulations diverge, then they do so because at some point the answer to a membership query on an assignment B is answered differently in the two simulations, and thus $f_{P_1}(B) \neq f_{P_0}(B)$, and so $f_{P_1}(P_0/B) \neq f(P_0/B)$. In this case we exploit the assignment P_0/B (calling *ProjBitFlip*) to find a justifying assignment for a variable Y in $V - V'$. We then add Y to V', add Y and its justifying assignment to S, and go back to the start of the loop.

We present the transformed algorithm below.

ToMM(AlgMJU)

```
1. Let $V' = \emptyset$. Let $S = \emptyset$.
2. Do forever:
    (a) Let $P_0$ be the partial assignment setting the variables in $V_n - V'$ to 0 and leaving the variables $V'$ unassigned. Let $P_1$ be the partial assignment setting the variables in $V_n - V'$ to 1 and leaving the variables in $V'$ unassigned.
    (b) If there is a pair $(X, A) \in S$ such that $A$ is not a justifying assignment for $X$ in $f_{P_0}$, then call ProjBitFlip($V'$,$P_0$,$P_1/A_{X\leftarrow 1}$, $S$) and set $V'$ and $S$ respectively to the values $W$, and $S'$ returned.
```

(c) Else if there is a pair $(X, A) \in S$ such that A is not a justifying assignment for X in f_{P_1}, then call *ProjBitFlip*(V',P_1,$P_0/A_{X\leftarrow 0}$, S) and set V' and S respectively to the values W and S' returned.

(d) Else run two parallel simulations of AlgMJU on input S to learn f_{P_0} and f_{P_1}. If the two simulations diverge on some membership query B then call *ProjBitFlip*(V',P_1,P_0/B, S) and set V' and S respectively to the values W and S'. Else if the two simulations do not diverge and they both output the same formula g, halt and output g.

We can also modify this MJ(U)→M(M) transformation to include equivalence queries in the base and target models, i.e. to produce the transformation MJE(U)→ME(M). The modification is basically the same as that described at the end of Section 4.2. Thus we have the following theorem.

Theorem 4 *Let $U(M)$ be a unate, projection closed class of formulas corresponding to a monotone class M. If $U(M)$ can be exactly identified in polynomial time by an algorithm using membership queries and justifying assignments, then M can be exactly identified in polynomial time by an algorithm using only membership queries. Furthermore, if $U(M)$ can be exactly identified in polynomial time by an algorithm using membership queries, equivalence queries, and justifying assignments, then M can be identified in polynomial time using only membership and equivalence queries.*

This transformation increases the algorithm's running time and number of membership queries by a factor of $O(n)$.

5 Conclusions

We have presented an exact identification algorithm that uses membership queries (i.e. black box interpolation) to identify monotone read-once threshold formulas. We have developed a list of generic transformations useful in understanding how this and other identification results in various different models can be related. In presenting these transformations our main interest was in polynomial time learnability rather than in tight bounds. We do not know whether these transformations are optimal in the sense that alternate reductions may be discoverable that use fewer queries and less time. Recently an alternate version of the MJ→ME transformation has been developed that uses a factor of $\log n$ fewer equivalence queries [3], but that uses more membership queries and running time. Even assuming we have optimal transformations, for the more difficult transformations we do not know whether optimal algorithms in the base model can yield optimal algorithms in the target model.

Besides the issue of efficiency it would be desirable to extend this work to develop a better understanding of the relation between results obtainable in the query models discussed here and results obtainable in more powerful or incomparable models that have been considered, such as subset/superset queries, constrained instance queries, and projective equivalence queries.

References

[1] D. Angluin. Queries and concept learning. In *Machine Learning*, 2:319–342, 1987.

[2] D. Angluin, L. Hellerstein, and M. Karpinski. Learning read-once formulas with queries. In *Journal of the Association for Computing Machinery*, 40:185–210, 1993.

[3] Nader H. Bshouty, Sally A. Goldman, Thomas R. Hancock, and Sleiman Matar. Asking questions to minimize errors. In *The 1993 Workshop on Computational Learning Theory.*

[4] Nader H. Bshouty, Thomas R. Hancock, and Lisa Hellerstein. Learning arithmetic read-once formulas. In *Proceedings of the 24th Annual ACM Symposium on the Theory of Computing*, 1992.

[5] Nader H. Bshouty, Thomas R. Hancock, and Lisa Hellerstein. Learning boolean read-once formulas with arbitrary symmetric and constant fan-in gates. In *The 1992 Workshop on Computational Learning Theory*, 1992. To appear in *JCSS.*

[6] Sally A. Goldman, Michael J. Kearns, and Robert E. Schapire. Exact identification of circuits using fixed points of amplification functions. In *Proceedings of the 31st Symposium on Foundations of Computer Science*, 1990.

[7] R. Heiman, I. Newman, A. Wigderson, On Read Once Threshold Formulas and their Randomized Decision Tree Complexity. In *IEEE Symp. on Structures in Complexity 1990, pp. 78-87.*