Learning Boolean Read-Once Formulas over Generalized Bases

Nader H. Bshouty[*]
Department of Computer Science
The University of Calgary
2500 University Drive N.W.
Calgary, Alberta, Canada T2N 1N4
e-mail: `bshouty@cpsc.ucalgary.ca`

Thomas R. Hancock[†]
Siemens Corporate Research
755 College Road East
Princeton, NJ 08540
e-mail: `hancock@learning.siemens.com`

Lisa Hellerstein
Department of EECS
Northwestern University
2145 Sheridan Road
Evanston, IL 60208-3118
e-mail: `hstein@eecs.nwu.edu`

Abstract

A read-once formula is one in which each variable appears on at most a single input. Angluin, Hellerstein, and Karpinski give a polynomial time algorithm that uses membership and equivalence queries to identify exactly read-once boolean formulas over the basis $\{\mathrm{AND}, \mathrm{OR}, \mathrm{NOT}\}$ [AHK93]. The goal of this work is to consider natural generalizations of these gates, in order to develop exact identification algorithms for more powerful classes of formulas. We show that read-once formulas over a basis of arbitrary boolean functions of constant fan-in k or less (i.e. any $f : \{0,1\}^{1 \le c \le k} \rightarrow \{0,1\}$) are exactly identifiable in polynomial time using membership and equivalence queries. We show that read-once formulas over the basis of arbitrary symmetric boolean functions are also exactly identifiable in polynomial time in this model. Given standard cryptographic assumptions, there is no polynomial time identification algorithm for read-twice formulas over either of these bases using membership and equivalence queries. We further show that for any basis class B meeting certain technical conditions, any polynomial time identification algorithm for read-once formulas over B can be extended to a polynomial time identification algorithm for read-once formulas over the union of B and the arbitrary functions of fan-in k or less. As a result, read-once formulas over the union of arbitrary symmetric and arbitrary constant fan-in gates are also exactly identifiable in polynomial time using membership and equivalence queries.

[*]This research was supported in part by the NSERC of Canada.

[†]This research took place while Tom Hancock was a graduate student at Harvard University, supported by ONR grant N00014-85-K-0445 and NSF grant NSF-CCR-89-02500.

1 Introduction

A read-once formula is one in which each variable appears on at most a single input. These can be viewed as circuits in which each gate, including the input units, has a fan-out of one. The basis of a formula is the set of functions computed by its gates. Angluin, Hellerstein, and Karpinski give a polynomial time algorithm that uses membership and equivalence queries to exactly identify a hidden target read-once boolean formulas over the basis $\{\mathrm{AND}, \mathrm{OR}, \mathrm{NOT}\}$ [AHK93]. (In this learning model proposed by Angluin [A88], a membership query allows the learner to present an arbitrary input instance and be told the target formula's true value on that input, and an equivalence query allows the learner to propose a candidate formula and either be told that indeed this hypothesis is correct, or else be given a counterexample on which the hypothesis and the target disagree.) An early generalization of their result shows that in this model the more general class of read-once formulas over the basis of arbitrary threshold functions remains efficiently learnable [BHHK91]. In this paper we consider much more powerful sets of boolean basis functions and show that read-once formulas over these generalized bases remain learnable. These results are the culmination of a line of research on learning boolean read-once formulas with queries.

The techniques of Angluin, Hellerstein, and Karpinski are fairly readily extended to allow exclusive or (XOR) as an additional basis function. This result can then be interpreted as saying that read-once formulas over the basis of all boolean functions of fan-in at most 2 are learnable. In this paper we extend this result to hold for an arbitrary constant k. We present a polynomial time algorithm that uses membership and equivalence queries to identify read-once formulas over this basis set $\{f : \{0,1\}^{1 \leq c \leq k} \to \{0,1\}\}$.

Another natural generalization of the basis $\{\mathrm{AND}, \mathrm{OR}, \mathrm{NOT}\}$ (and the boolean threshold basis) is to arbitrary symmetric functions (i.e. functions, such as majority, whose output is invariant under permutations of the input bits). We show that read-once formulas over the basis of arbitrary symmetric functions are polynomial time identifiable with membership and equivalence queries.

We further show that we can combine these two algorithms, to obtain the result that read-once formulas over the union of symmetric and arbitrary constant fan-in functions are identifiable in polynomial time. More generally, we show that if B is a basis (meeting certain technical conditions) for which there exists a polynomial time membership and equivalence query algorithm to identify read-once formulas over B, then there also exists a polynomial time membership and equivalence query algorithm to identify read-once formulas over the union of B and the arbitrary functions of fan-in k or less. An analogous statement holds in Valiant's PAC model, with the addition of membership queries, so that if read-once formulas over B are polynomial time PAC learnable with membership queries, then so are read-once formulas over B and fan-in k functions. We can therefore extend a number other learning results by choosing B to be either binary matroid port functions [HC93] (a generalization of switch configurations [RS90]), read-once perceptron networks [HGM91], or μ branching programs [RW93]. Boolean formulas over the constant fan-in basis can simulate decision trees, using the three input if-then-else function that evaluates either its second or third argument depending on the value of the first argument. Thus our results generalize earlier work on μ-formula decision trees [Han90].

There are a number of technical challenges involved in learning read-once formulas over our bases. For example, there is no longer necessarily a unique topology to represent a formula (even if we require that negations be pushed to the leaves and that adjacent AND or OR gates be merged together). We introduce the notion of prime functions to establish a canonical form for our formulas.

Another problem is that, in contrast to the situation in the $\{\mathrm{AND}, \mathrm{OR}, \mathrm{NOT}\}$ basis, a formula over our bases may consist of gates that are all of the same type but cannot be merged together. Thus simply finding out the type of the lowest common ancestor gate of every pair of variables in the formula (as is done for the $\{\mathrm{AND}, \mathrm{OR}, \mathrm{NOT}\}$ basis) is not sufficient to reconstruct formulas over our bases. The arbitrary fan-in basis presents the problem of handling asymmetric basis functions.

In developing learning algorithms for the basis of arbitrary symmetric functions, a more subtle but significant source of difficulty lies in what information we need to obtain via equivalence queries. Angluin, Hellerstein, and Karpinski use equivalence queries only to generate a set of "justifying assignments" for the variables (i.e. input settings for which flipping the value of a particular variable changes the output of f). They then present an exact identification algorithm that uses just membership queries and justifying assignments. The same approach is used in the threshold basis algorithm [BHHK91]. This approach is not adequate for the symmetric basis, since one can show that there is no polynomial time identification algorithm for such read-once formulas using only membership queries and justifying assignments. Thus to achieve our positive results, we are forced to use our equivalence queries in a qualitatively more substantial way.

Under certain cryptographic assumptions (such as the hardness of factoring Blum integers), Angluin and Kharitonov have shown that there is no polynomial time algorithm to identify boolean formulas in this query model (or to PAC learn them with membership queries), even when each variable appears at most three times [AK91]. For our bases we show that this is true even if variables appear at most twice. The key new basis function for this argument is the function that computes whether three values are equal. Thus our results are tight in the sense that they cannot be achieved for read-twice or greater formulas (under the cryptographic assumptions).

The rest of this paper is organized as follows. In Section 2 we present our definitions and terminology, and we prove a few basic results about our bases. In Section 3 we present a number of basic techniques for learning read-once formulas (and perhaps other classes) that will reduce our learning task to a number of simpler sub-problems. In Section 4 we present the algorithm to learn read-once formulas over the basis of arbitrary functions of constant fan-in, and in Section 5 we generalize this argument to show how constant fan-in functions can be added to another basis for which a learning algorithm is known. In Section 6 we present the algorithm to learn read-once formulas over the basis of arbitrary symmetric functions Finally, in Section 7 we present the negative results for read-twice formulas.

2 Definitions and Preliminary Material

2.1 Boolean read-once formulas

A formula can be viewed as a rooted tree where each internal node (or *gate*) is labeled by the function computed by the gate, and where each leaf contains a variable or constant. We define the *basis* of a formula to be the set of functions from which the internal nodes are labeled (usually assumed by default to be $\{\mathrm{AND}, \mathrm{OR}, \mathrm{NOT}\}$)[1]. A formula is *read-once* if no variable appears on more than one leaf. If B is a set of basis functions, we shall let $\mathrm{ROF}(B)$ denote the set of all read-once formulas over that basis (for the identification problem we are interested in those formulas that are defined

[1]We do not address the technical issue of how a gate is "labeled", but for the purpose of stating performance bounds we assume that all our gates may be evaluated in time proportional to the number of inputs.

on a specific number of inputs, n). We assume without loss of generality that every basis function depends on all its inputs.

The two main basis sets we consider are the following:

- All boolean functions of k or fewer inputs (where k is an arbitrary constant). We denote this set as $\{f : \{0,1\}^{1 \leq c \leq k} \rightarrow \{0,1\}\}$.
- Arbitrary symmetric boolean functions. A function of multiple inputs is *symmetric* if its output is invariant under permutations of those inputs (i.e. the count of how many inputs are set to 1 is sufficient to determine the output of the function).

A boolean function is *unate* if for each variable, it is either monotone in that variable or its complement. Some of the functions in the above bases are unate (e.g. AND, OR, and NOT), and some are not (e.g. XOR).

The *depth* of a gate in the formula is the distance from the gate to the root (the root itself is as depth 0). I.e. the inputs to a gate at depth d are all at depth $d+1$.

For any pair of vertices v and w in the tree, there is a unique deepest node (farthest from the root) that is an ancestor of both v and w, called their *lowest common ancestor*, which we write as $\mathrm{lca}(v,w)$. We say that a set W of variables have a common lca if there is a single vertex that is the lca of every pair of variables in W. In this paper we do not consider a node in a tree to be its own descendant.

We define the *skeleton* of a formula f to be the tree derived from f by deleting any unary gates (i.e. negations) and erasing the labels from any remaining gates. (Thus the skeleton describes the topology of the formula, which is typically highly constrained if not uniquely determined in the case of read-once formulas, while leaving open the question of what the formula's actual gates compute.)

2.2 Assignments, partial assignments, and projections

An *assignment* to a set V of boolean variables is a function from V to $\{0,1\}$. If f is a formula defined on V, and a is an assignment to the variables in V, then we denote by $f(a)$ the output of the formula f when its inputs are set according to the assignment a. If $x^{(0)} \in \{0,1\}$, let $a_{x \leftarrow x^{(0)}}$ denote the assignment b such that $b(y) = a(y)$ for all $y \in V - \{x\}$, and $b(x) = x^{(0)}$. Let $a_{\neg x}$ denote the assignment $a_{x \leftarrow \neg a(x)}$. We shall also use the notation $a_{x \leftarrow x^{(0)}, y \leftarrow y^{(0)}}$ to denote the assignment obtained from a by setting both x to $x^{(0)}$ and y to $y^{(0)}$, and we shall write $a_{W \leftarrow x^{(0)}}$ where W is a set of variables to denote the assignment obtained from a by setting each variable $x \in W$ to the value $x^{(0)}$.

We speak of the Hamming distance between two assignments (viewed as bit vectors) to mean the number of variables on which they disagree. Two assignments are *complementary* if they disagree on every variable. An assignment a is *positive* if $f(a) = 1$ and *negative* if $f(a) = 0$. We shall sometimes refer synonymously to *examples* rather than assignments.

We say that a formula f *is defined on* the variable set V if all variables appearing in f are members of V. If f is defined on V, a is an assignment to V, $x \in V$, and $f(a) \neq f(a_{\neg x})$, then a is *justifying* for x in V. 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. We say variable x is *relevant* (for f) if f depends on x.

We say an assignment p to a set of variables V is *partial* if it assigns values to only some subset of V. If p does not assign a value to a variable x we write $p(x) = *$. If p is a partial assignment and a is a total assignment, we let p/a denote the *extension* of p obtained by setting the unassigned variables according to a, i.e.

$$(p/a)(x) \quad = \quad \begin{cases} p(x) & \text{if } p(x) \neq * \\ a(x) & \text{if } p(x) = *. \end{cases}$$

For a formula f and a partial assignment p we define the *projection* f_p to be the formula induced on f by replacing those variables assigned values by p by the appropriate constants. That is, we *hard-wire* to its assigned value each input that is assigned a value by p. As a consequence of our definitions it is easy to see that for any partial assignment p and any assignment a,

$$f(p/a) \quad = \quad f_p(a).$$

We say a class of formulas C is *projection closed* if any projection of a formula from C is also in C (or has an equivalent representation in C). For all bases B that we consider, the class ROF(B) is projection closed. Our results below will repeatedly state a condition that the basis B be projection closed, which implies that ROF(B) is projection closed. (When we say a basis is projection closed we ignore the constant projections, which may not be in the basis, but can be realized with zero gate formulas.) While it is possible for B not to be projection closed while ROF(B) is, all our results can cover such a case by replacing the basis B with its closure under projections, which does not affect the expressive power of the class of formulas.

A *c-justifying assignment* for a set W of c variables is a partial assignment p that assigns values to the variables not in W in such a way that f_p depends on each variable in W. If a is a justifying assignment for x and y then $a_{x\leftarrow *}$ is a one-justifying assignment for x, and $a_{x,y\leftarrow *}$ is a two-justifying assignment for $\{x, y\}$.

2.3 Symmetric boolean functions

We describe a symmetric function f on n inputs by a *vector* $\langle \alpha_0, \alpha_1, \ldots, \alpha_n \rangle$, where α_i is the value of f when exactly i inputs are set to 1. We name the functions with vectors of the form $\langle 1, 0, 0, \ldots, 0, 0, 1 \rangle$ as *consensus* functions (these test whether all inputs have the same value).

We define the *dual* of a symmetric function g with a vector $\langle \alpha_0, \alpha_1, \ldots, \alpha_m \rangle$ to be the function $g^{(dual)}$ with the vector $\langle \neg\alpha_m, \neg\alpha_{m-1}, \ldots, \neg\alpha_0 \rangle$. It is easy to verify the following generalization of deMorgan's law:

$$\neg g(x_1, \ldots, x_m) \quad = \quad g^{(dual)}(\neg x_1, \ldots, \neg x_m).$$

This fact allows us to assume that in the symmetric basis all negation appears at the leaves (i.e. all NOT gates have variables as their inputs).

2.4 Prime boolean functions

A function is *prime* if any read-once formula computing the function (over an arbitrary basis) cannot include more than one non-unary gate. In other words, a function f over n variables that depends on all its inputs is a prime function if there is no choice of functions f_1 and f_2 (as read-once formulas or otherwise) such that for some permutation σ on $\{1,\ldots,n\}$ and some $1 < k < n$,

$$f(x_1,\ldots,x_n) \quad = \quad f_1(f_2(x_{\sigma(1)},\ldots,x_{\sigma(k)}), x_{\sigma(k+1)},\ldots x_{\sigma(n)})$$

(The term prime comes by analogy to factoring, where we view the description of a function as a composition of sub-functions over smaller but non-trivial subsets of its inputs as being analogous to the description of an integer as a product of non-trivial factors.)

Since we deal only with bases that are projection closed, we shall be able to assume without loss of generality that our read-once formulas contain only gates computing prime functions.

If a function f in a projection closed basis is not prime and may be decomposed as above, then the basis must also include functions f_1 and f_2, as they are computable as projections of f. (Actually the projections that are in the basis may not be exactly f_1 and f_2, but rather may have some of the inputs and possibly the output negated, But that is possible only in cases where projection closure implies negation is in the basis, in which case it remains true that both f_1 and f_2 can be expressed by basis functions.) Thus we can replace any gate in labeled with f by a pair of gates labeled with f_1 and f_2. Hence for a projection closed basis we may assume without loss of generality that all gates compute prime functions.

We define a relation $\sim$ by saying $f_1 \sim f_2$ if there exist $\xi, w_1,\ldots,w_n \in \{0,1\}$ and a permutation σ_y such that

$$f_1(x_1,\ldots,x_n) \quad = \quad \xi + f_2(x_{\sigma_y(1)} + w_1,\ldots,x_{\sigma_y(n)} + w_n)$$

where "+" refers to addition modulo 2 (i.e. exclusive or). We let $[f]$ denote the equivalence class of f according to this relation. It is easy to show the following lemma. This implies that if we assume all the gates in a read-once formula are prime, then whether a set of variables share an lca is invariant across equivalent formulas.

Lemma 1 *Suppose f is a boolean read-once formula (over an arbitrary basis) and p is a partial assignment such that f_p is a prime function depending on variables $x_1,\ldots,x_n$. Then $x_1,\ldots,x_n$ have a common lca in f. If lca($x_1,\ldots,x_n$) has only n children, then it is labeled with a function from $[f_p]$.*

Proof:

If the variables do not share an lca then the projection f_p will be a read-once formula with more than one gate of fan-in at least two, and hence not prime (by definition). If lca($x_1,\ldots,x_n$) exists and computes the function $g(x_1,\ldots,x_n)$, then f_p will compute $\xi + g(x_1 + w_1,\ldots,x_n + w_n)$ (for $\xi, w_i \in \{0,1\}$ and + being XOR). It follows that $g \in [f_p]$.□

The following two lemmas give general properties of prime functions. We first prove that "almost all" boolean functions are prime.

Lemma 2 *The number of prime functions of n inputs is more than $M - O(M^{1/2})$, where $M = 2^{2^n}$*

Proof:

If f is not prime, then

$$f(x_1,\ldots,x_n) \quad = \quad f_1(f_2(x_{i_1},\ldots,x_{i_k}), x_{j_1},\ldots x_{j_{n-k}})$$

where $n > k > 1$, $\{i_1, \ldots, i_k\} \cup \{j_1, \ldots, j_{n-k}\} = \{1, \ldots, n\}$. There are at most $n(n!)$ possible ways of choosing $x_{i_1}, \ldots, x_{i_k}, x_{j_1}, \ldots, x_{j_{n-k}}$. There are at most $O(2^{2^{n-2}})$ ways to choose f_1 and f_2. Therefore there are at most $M^{1/2}$ ways of constructing a non-prime function. All other functions are prime. $\square$

Lemma 3 *All non-constant symmetric functions are prime with the exception of those in* $[AND]$ *(=* $[OR]$*) and* $[XOR]$ *with more than two inputs.*

Proof:

Let $f(x_1, ..., x_n)$ be a non-constant symmetric function such that $n > 2$. Let h be a read-once formula computing f in which each gate computes a prime function.

Suppose that h contains a gate G with fan-in $2 < k \leq n-1$. Let g be the prime function labeling that gate. Let $W = \{x_{j_1}, \cdots, x_{j_k}\}$ be a set of variables that have gate G as their common lca. There exists a k-justifying assignment p for W such that $f_p = g'(x_{j_1}, \cdots, x_{j_k})$ for some $g' \in [g]$. By the symmetry of f, for any other set W' of k variables of f, there exists a k-justifying assignment p' for W' such that $f_{p'}$ computes this same prime, symmetric, function g'. It follows by Lemma 1 that each set of k variables of f have a common lca. This implies that h has just one non-unary gate with n inputs, and thus f is prime.

Suppose instead that all the gates in h have fan-in at most two. It is easily verified that because f is symmetric, these fan-in 2 gates must combine to compute a function in [AND] (= [OR]) or [XOR]. $\square$

Our algorithm will perform primality testing on various projections of the target formula. These projections will depend on only a constant number of variables so we can do this testing exhaustively in time polynomial in n. We know of no other technique more efficient than trying all possible decompositions.

We shall refer to a partial assignment as prime (or symmetric, or asymmetric) to mean that the projection it induces is a prime (or symmetric, or asymmetric) function.

2.5 Identification with queries

We consider learning in a model developed by Angluin [A88]. The learning goal 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. 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 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*. 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 $h(b) \neq f(b)$.

In the learning procedures given in this paper, we assume that the algorithm has access to membership and equivalence oracles for a target formula f, and to a variable set V. We do not include these oracles, or the set V, as inputs to the procedures, but specify them when the procedures are called.

We follow Angluin, Hellerstein, and Karpinski [AHK93] in determining how much time to charge for making a query. We charge both for setting up the query and for invoking the oracle. In a membership query consisting of an assignment to n variables, we charge unit time for specifying each of the n assignments. Therefore, the set-up cost of a membership query is typically $O(n)$ (it can be lower if the query is formed by changing only a small number of bits in the previous query, as is the case in several of our algorithms). The set-up cost of an equivalence query is the time it takes to "write down" the hypothesis in some sort of "reasonable" representation. In this paper, our equivalence queries usually involve either read-once formulas over the symmetric basis, or read-once formulas over the fan-in k basis. Each such formula contains $O(n)$ nodes, and the labels of all the nodes in the formula can be represented by a string of $O(m)$ bits where m is the number of inputs to the formula (ignoring constant factors involving k). We therefore consider the set-up cost of these equivalence queries to be $O(n)$.

We charge unit time for invoking either the membership oracle, or the equivalence oracle, once the query is set up.

3 General Techniques

In this section we present some techniques generally useful in identification algorithms for read-once formulas (and perhaps other concept classes as well). Much of this machinery is carried over from previous work on read-once formulas [AHK93, BHHK91] and is also used in other read-once formula algorithms [HGM91, BHH92].

We first observe a standard reduction stated in Lemma 4, showing that as long as the target class is projection closed, we may assume without loss of generality that the learning algorithm has a justifying assignment for each relevant variable. In Section 3.1 we show how to learn a formula given its skeleton (or at least a partition of variables according to whether they appear in some subformula).

Lemma 4 ([BHHK91]) *Suppose C is a projection closed class for which there is polynomial time exact identification algorithm that uses membership queries, equivalence queries, and a justifying assignment for each relevant variable. Then there is a polynomial time exact identification algorithm for C that uses only membership and equivalence queries.*

Proof:

The algorithm using justifying assignments is called as a subroutine from within a single main loop that uses equivalence queries to gradually find justifying assignments for more and more relevant variables. On each iteration of the loop we have a subset V' of V, and a partial assignment p assigning values to the variables in $V - V'$ and leaving the variables in V' unassigned. For every variable $x \in V'$, we know a justifying assignment for x in f_p. For the first iteration of the loop, we set V' to the empty set, and we set p to an arbitrary assignment to V (so f_p is constant).

We then run the algorithm $\mathcal{A}$ that uses the justifying assignments to learn the projection f_p. We simulate a membership oracle for f_p using the membership oracle for f. If $\mathcal{A}$ makes a query $h \equiv f_p$? we make the equivalence query $h \equiv f$? If the answer is "yes", we're done. Otherwise we get a counterexample c for which either $h(c) \neq f_p(c)$ or $f_p(c) \neq f(c)$. In the former case we answer $\mathcal{A}$'s equivalence query "no", and give c as a counterexample. In the latter case we terminate $\mathcal{A}$, and run a routine *Bit-Flip* (Figure 1) that returns a new, larger subset V' of V, a new associated projection p, and justifying assignments (with respect to f_p) for the variables in the new V'. We then repeat the loop.

Termination of the main routine is guaranteed by the fact that at each iteration we increase the size of V'.

Bit-Flip takes as input V', the projection p, and the counterexample a. It returns a new and strictly larger set of variables for the next V', as well as justifying assignments for all the new variables added to V'.

Bit-Flip(V', p, a, A)

1. Set b to a. Set W to be the set of variables in $V - V'$ such that $b(x) \neq p(x)$.
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. Return $V' \cup W$ and $p' = p_{W \leftarrow *}$. (Note that a justifying assignment for a variable with the target function f_p will still be justifying for the new target $f_{p'}$.)

Figure 1: Subroutine *Bit-Flip*

This processing cannot 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 $V' \cup W$, and the new p assigns $*$ to all variables in W. Assignment b is justifying for those new variables in f_p and in f. $\Box$

Using the above technique to convert an algorithm using membership queries, equivalence queries, and justifying assignments, to one that uses only membership and equivalence queries, adds a factor of $O(n)$ overhead.

3.1 Learning a Read-Once Formula Given a Skeleton

In this section we present a technique to identify an unknown formula from $\mathrm{ROF}(B)$ given its skeleton. This is a general technique that works for any basis B meeting certain technical conditions. Thus we reduce the problem of read-once formula identification to that of finding a correct skeleton. Note that it is possible in principle for two read-once formulas $f_1 \equiv f_2$ to have significantly different skeletons. This will turn out not to be the case for the bases we consider (although some variation is possible, as for example $\mathrm{AND}(x, \mathrm{AND}(y, z)) \equiv \mathrm{AND}(\mathrm{AND}(x, y), z)$).

Lemma 5 *Suppose B is a projection closed set of basis functions. Then given a justifying assignment for each relevant variable,* $\mathrm{ROF}(B)$ *is identifiable in polynomial time with membership queries if 1) There is a polynomial time routine that uses justifying assignments and membership queries to determine the skeleton of any formula $f \in \mathrm{ROF}(B)$ (or of a formula equivalent to f). 2) There is a polynomial time routine that uses justifying assignments and membership queries to identify any formula in* $\mathrm{ROF}(B)$ *that has only one non-unary gate. If there are polynomial time routines for problems (1) and (2) using membership and equivalence queries, then* $\mathrm{ROF}(B)$ *is identifiable in polynomial time using membership and equivalence queries.*

Proof:

First learn the skeleton. To find a formula equivalent to f it is then necessary to determine which basis functions are computed at internal nodes and to determine where negations need be inserted (if negations are in the basis). For the moment, assume that f does not contain any negations, except possibly at the leaves.

The basic idea is to run parallel copies of the algorithm for problem (2), where each copy is applied to a different node of the skeleton. For this to be feasible we need to be able to provide justifying assignments for each gate in the skeleton (where the "variables" are now the subformulas that are the inputs to the gate) and to answer the membership and possibly equivalence queries for the gate. To show how to do this (using justifying assignments and oracles for f) let us first suppose we are able to perform the following two tasks for any gate G in the skeleton,

Evaluation: Determine the output of G on an arbitrary assignment,

Inversion: Given a desired value for G, find an assignment on which G outputs that value.

Suppose for the moment that we have efficient solutions to the evaluation and inversion problems for each gate. We argue that we can run our single gate subroutine to learn the function computed at each gate in the skeleton. To find a justifying assignment for some input to a gate G, use a justifying assignment for a variable that appears in the corresponding subformula of G. Then use the evaluation procedure to determine the values assumed by all G's inputs on this assignment. The result is a justifying assignment for the specified input to G. When the routine to learn gate G makes a membership query, it specifies values for each of G's inputs. By inversion, obtain settings for each variable that is a descendant of G, and then use evaluation to determine G's output on this assignment (answering the membership query). When the subroutine makes an equivalence query for G, pause and wait until all other subroutines are ready to make equivalence queries (or have halted successfully). Then form a hypothesis h by labeling each gate of the skeleton with the hypothesis of the corresponding equivalence query. Make the equivalence query $h \equiv f$?, and if the answer is "yes", we output h and halt. If we get a counterexample c, then clearly there must be some node(s) in the skeleton for which the output of h from that node's gate is different from the output of f (certainly true at the root). Since h and f agree on all the input variable settings when evaluating c, there must be some deepest node in the skeleton at whose output h and f differ (as we propagate values bottom up from the leaves when evaluating c, at some point we reach a first node in the skeleton where h's gate produces one value and f's gate produces the other). Thus there is at at least one gate G in h for which the inputs are set correctly (we can check this for each gate by evaluation), but the output is wrong (also determined by evaluation). For each such gate we restart the subroutine to learn that gate, giving it c as a counterexample to the last equivalence query. We let the subroutine(s) continue until the next round of equivalence queries.

Now we need to show how to solve the evaluation and inversion problems for an arbitrary gate G. Pick a variable x that is a descendant of G, and let a be a justifying assignment for x. Let p be the partial assignment obtained from a by setting to $*$ (i.e. unsetting) all descendants of G. We claim that without loss of generality, the output of G on an assignment is equivalent to f_p. Assuming this for the moment, then we can solve the computation problem using a simulated oracle for f_p, and we can also solve the inversion problem, since to obtain a value t as the output of G we need just to pick whichever of $a_{x\leftarrow 1}$ or $a_{x\leftarrow 0}$ sets f_p to t. In truth though, if the basis B is non-monotone, the output of G in f might be the negation of f_p (and since B is projection closed, negation must be in

B). We handle this by observing that there is a formula equivalent to f with the same skeleton, but where there is a negation gate applied to the output of G, and another negation gate immediately above that. If we temporarily extend the basis to include compositions of unary basis functions with non-unary basis functions (which we can do without changing the power of the class), then we can say that there is a modified formula $\hat{f}$ with $\hat{f} \equiv f$ and the same skeleton as f, such that indeed the projections f_p for each gate give the formula's true output of the gate. Under this extended basis, our initial assumption that f contains no negations except at the leaves is justified.

Our routine to solve problem (2) still works over this slightly enhanced basis. Now we can solve the evaluation and inversion problems for $\hat{f}$, and hence learn that formula as above.

The final sentence of the lemma follows from the fact that in this case we can generate justifying assignments by Lemma 4. □

The resource bounds of this process come from the sum of those for building the skeleton and for running the algorithm to learn single gate formulas, where in the latter case there is $O(n)$ overhead for each membership and equivalence query. There is also a potential factor of n overhead to find justifying assignments (as in Lemma 4).

3.1.1 Learning a read-once formula with a meta-skeleton or partitions

Lemma 5 shows that we can reduce our learning problem to the two subproblems of constructing a skeleton and learning a single gate formula. The lemma can also be applied to two situations where we do not find a complete skeleton, but have useful partial skeleton information.

One situation is where the basis B can be expressed as $B_1 \cup \cdots \cup B_t$ where the B_i's are *types* of basis functions. We shall only consider cases where the B_i's are disjoint except for NOT. We define the *meta-skeleton* of f with respect to the basis decomposition $B = B_1 \cup \cdots \cup B_t$ as the skeleton of a formula f' from $\mathrm{ROF}\,(\mathrm{ROF}\,(B_1) \cup \cdots \cup \mathrm{ROF}\,(B_t))$ where $f' \equiv f$ and where f' has no two adjacent gates labeled with formulas over the same basis type. In other words the gates of the formula are labeled with functions that are themselves read-once formulas over one of the B_i's. A meta-skeleton can be obtained by taking a skeleton of f and combining adjacent nodes that were labeled with basis functions from the same B_i. Applying Lemma 5 to the class $\mathrm{ROF}\,(\mathrm{ROF}\,(B_1) \cup \cdots \cup \mathrm{ROF}\,(B_t))$ gives the following result.

Lemma 6 *Suppose $B = B_1 \cup \cdots \cup B_t$ is a projection closed set of basis functions. Then the class* $\mathrm{ROF}\,(B)$ *is learnable in polynomial time with membership and equivalence queries if for any $f \in \mathrm{ROF}\,(B)$, 1) There is a polynomial time routine that learns the meta-skeleton with respect to $B_1 \cup \cdots \cup B_t$ of a formula in* $\mathrm{ROF}\,(B)$ *that is equivalent to f using membership and equivalence queries, and 2) For each $B_i \in B$ there is a polynomial time routine that learns formulas in* $\mathrm{ROF}\,(B_i)$ *using membership and equivalence queries.*

We define the *type* of a node in f, with respect to the decomposition $B_1 \cup \cdots \cup B_t$, to be the basis class B_i containing the function whose label appears on the node. The following lemma (which is a variation of a lemma used in Angluin et. al. [AHK93]) states that if we can determine the type of the lca of all pairs of variables in f, then we can construct the meta-skeleton of f in polynomial time. This reduces problem (1) to the problem of determining the types of the lca's of all pairs of variables in f.

Lemma 7 *Let f be read-once formula over the basis B. Let $B_1 \cup \cdots \cup B_t$ be a decomposition of the basis B such that the B_i's are disjoint (except possibly for the function NOT). Then there is a polynomial time procedure that, given the type of lca(x,y) for all pairs of variables $\{x, y\}$ in f, constructs the meta-skeleton of f with respect to the decomposition $B_1 \cup \cdots \cup B_t$.*

Proof:

Let f' be a formula from ROF (ROF $(B_1) \cup \cdots \cup$ ROF (B_t)) such that $f' \equiv f$ and f' has no two adjacent gates labeled with formulas over the same basis type (as described above). The skeleton of f' is the meta-skeleton of f.

For each basis type B_i, we define the *B_i-graph* of f'. The vertex set of the B_i-graph of f' consists of the variables in f'. The edge set consists of all pairs of variables $\{x, y\}$ in f such that lca(x,y) is not of type B_i.

We define the *type graphs* of f' to be all B_i-graphs of f' such that there is a node of type B_i occurring in f'.

Given the type of lca(x,y) for all pairs of variables in f, it is easy to construct the type graphs of f'. We show that we can use these graphs to recursively construct the skeleton of f'.

Our construction relies on the following observations. If f' contains at least one node, and the root of f' is labeled with a function from ROF (B_j), then the B_j-graph of f' is disconnected, and for every type $B_i \neq B_j$, the B_i-graph is connected. Suppose f' has at its root a node labeled with a function from ROF (B_z), and the children of the root are the roots of the subtrees $f_1', f_2', ..., f_k'$, where $k \geq 2$. Since f' does not contain two adjacent nodes labeled with functions of the same type, the roots of $f_1', f_2', ..., f_k'$ are not of type B_z. It follows that the B_z-graph of f' is disconnected. and consists of k connected components $C_1, ..., C_k$, which contain the variables appearing in $f_1', f_2', ..., f_k'$ respectively. Furthermore, for all other types $B_i \neq B_z$, the B_i-graph of f' is connected.

Finally, for all $j \in [1..k]$, the B_i-graph of $f_j{}'$ is the subgraph of the B_i-graph of f' induced by the vertices in C_i.

To construct f' from its type graphs, we first check whether the type graphs all consist of a single leaf labeled with a variable y. If so, f' must consist of the single variable y. Otherwise, we perform depth-first search on the type graphs to find the unique type B_z such that the B_z-graph of f' is disconnected. In performing depth first search on the B_z-graph, we also determine the connected components $C_1, ..., C_k$ of the graph. For $1 \leq j \leq k$, the B_i-graph of $f_j{}'$ is the subgraph of the B_i-graph of f' induced by the variables in C_j. Using these subgraphs, we then recursively construct $f_1', ..., f_k'$.

The skeleton of f' consists of a root node whose children are the roots of the subtrees $f_1', ..., f_k'$.
$\square$

Another situation where we do not find the complete skeleton is the case where we find a *partition* of the variables $V = V_1 \cup V_2$ where V_1 is exactly the set of variables in a subformula of f. Such a decomposition represents a two node skeleton for the target formula over the basis ROF (B) (i.e. we view our class as ROF (ROF (B)) instead of ROF (B)). Thus the technique of Lemma 5 provides a recursive algorithm in this case, where by finding a partition we solve a problem of size n by combining the answers of two smaller problems whose total size is (basically) n. We state this in the following lemma (a subformula of f is trivial if it includes a single variable or all the variables).

Lemma 8 *Suppose B is a projection closed set of basis functions. Then the class* ROF (B) *is learnable in polynomial time with membership and equivalence queries if for any $f \in$* ROF (B)*, 1)*

There is a polynomial time routine that identifies the set of variables that appear in some non-trivial subformula of f (should such exist) using membership and equivalence queries, and 2) There is a polynomial time routine that identifies formulas in ROF (B) *that have a single non-unary gate using membership and equivalence queries.*

4 Learning read-once formulas over the fan-in k basis

In this section we show how to learn read-once formulas over the basis $\{f : \{0,1\}^{1 \leq c \leq k} \rightarrow \{0,1\}\}$. We assume without loss of generality that all gates in the target formula are prime.

We start in Section 4.1 by solving the special case of $k = 2$. Here without loss of generality we need only consider the basis {AND,OR,XOR,NOT}. Given the machinery of the previous section, this requires a fairly minor extension to Angluin, Hellerstein, and Karpinski's algorithm for the {AND,OR,NOT} basis.

To learn ROF $\left(\{f : \{0,1\}^{1 \leq c \leq k} \rightarrow \{0,1\}\}\right)$ for an arbitrary k, we begin by generating c-justifying assignments for many different sets of variables. We are able to do this extensively enough to guarantee that for any gate G of f with $c \leq k$ inputs, we obtain a c-justifying assignment for some set of variables W that includes one from each subformula of that gate. So if f contains a gate G labeled with a function from $\{f : \{0,1\}^{3 \leq c \leq k} \rightarrow \{0,1\}$ and f prime $\}$, we shall have a prime c-justifying assignment for a set of variables whose lca is G. A further technique allows us to partition all the variables according to whether they are descendants of G, and if so in which subformula they appear. We describe these two techniques in Section 4.2.

As long as f has some gate of fan-in three or more, this allows us to apply our recursive partitioning technique of Lemma 8. When all gates have fan-in two or less, we learn f over the fan-in two basis. In Section 4.3 we prove our main theorem for the basis $\{f : \{0,1\}^{1 \leq c \leq k} \rightarrow \{0,1\}\}$.

In the next section we shall generalize this result to produce an algorithm to learn read-once formulas over $B \cup \{f : \{0,1\}^{1 \leq c \leq k} \rightarrow \{0,1\}\}$ for a general basis B (given that ROF (B) is learnable and B meets certain technical conditions). In preparation for that result, this section will use some slightly more general approaches than we immediately require.

4.1 Learning read-once formulas over the fan-in 2 basis

In this section we consider the case where the only gates in the formula are AND, OR, XOR, and NOT. We learn formulas over this basis by finding a meta-skeleton with respect to the bases {AND,OR,NOT} and {XOR,NOT}. We already know how to learn read-once formulas over {AND, OR, NOT} [AHK93], and learning formulas over {XOR,NOT} is trivial since all such formulas collapse to compute either XOR, its negation, or a constant. Thus by applying Lemmas 6 and 7, we can reduce our learning problem to deciding for each pair of variables whether their lca is XOR or is in {AND,OR}. The following lemma shows how to do this (in a slightly more general fashion than we need for the moment).

Lemma 9 *Let B be a basis that includes XOR, but does not include any other function whose projection is XOR or its complement. Then there is a polynomial time membership query and justifying assignment algorithm that determines, for any formula in ROF(B), whether the type of the lca of each pair of variables x,y in the formula is XOR.*

Proof: To determine the type of $\mathrm{lca}(x, y)$, we start with a justifying assignment a for x and a justifying assignment b for y. We loop through each variable z on which $a(z) \neq b(z)$, and if $a_{\neg z}$ is still justifying for x we reset a to $a_{\neg z}$. (we call this process *walking* a towards b). Suppose $\mathrm{lca}(x, y)$ is an XOR gate. Then changing any variable in y's subformula of this gate cannot affect the fact that a is justifying for x. Thus after a single pass of such changes, it must be true that $p = a_{x,y\leftarrow *}$ is two-justifying for x and y and that f_p is $\mathrm{XOR}(x, y)$ or $\neg\mathrm{XOR}(x, y)$. Conversely if $\mathrm{lca}(x, y)$ is not an XOR gate, then f_p cannot compute XOR or $\neg$XOR unless the projection of that gate does, which is not possible by the conditions of the lemma. Thus with $O(n^3)$ time and membership queries we are able to determine, for each pair of variables, whether the type of the lca is XOR. □

4.2 Routines for finding k-justifying assignments and partitioning

In this section we present two key routines for generating justifying assignments for formulas over the fan-in k basis.

The routine *Grow-k-Justifying-Assignments* (Figure 2) takes a set J_1 of one-justifying assignments for each relevant variable, and returns a set J of partial assignments with the property that for each gate G of fan-in $c \leq k$, there is a c-justifying assignment in J for a set of variables that includes one variable from each subformula of G. (We can efficiently determine whether a function of k or fewer inputs depends on a variable simply by building a complete truth table with membership queries).

Grow-k-Justifying-Assignments(J_1)

1. Repeat for $d = 2$ to k.
   (a) Initialize J_d to $\emptyset$.
   (b) For each $p \in J_{d-1}$ that is an $(d-1)$-justifying assignment for $W = \{x_1, \ldots, x_{d-1}\}$ and for each $y \in V - W$ with justifying assignment a,
      i. Initialize q to p. Loop through the variables $z \notin W$ and either
         - Reset q to $q' = q_{z\leftarrow a(z)}$ if q' is still a $(d-1)$-justifying assignment for W.
         - Otherwise add $q_{z\leftarrow *}$ to J_d as a d-justifying assignment for $W \cup \{z\}$ (if that set of variables is not yet represented in J_d).

      ii. If $f_{q_{y\leftarrow *}}$ depends on y, add $q_{y\leftarrow *}$ to J_d as a d-justifying assignment for $W \cup \{y\}$ (if that set of variables is not yet represented in J_d).
2. Return $J_1 \cup J_2 \cup \cdots \cup J_k$.

Figure 2: Subroutine *Grow-k-Justifying-Assignments*

We prove the following lemma that applies to read-once formulas over any basis.

Lemma 10 *Suppose J_1 is a set of one-justifying assignments including one for each relevant variable in a read-once formula f over an arbitrary basis. The routine Grow-k-Justifying-Assignments when invoked with a membership oracle for f will return a set J of partial assignments with the following property: For every gate G in f with fan-in $c \leq k$, there is a $p \in J$ where p is a c-justifying assignment for a set of variables that includes one variable from each subformula of G.*

Proof:

Suppose G is some gate with fan-in $c \leq k$. We prove the lemma by showing by induction that for all $d \leq c$, J_d will contain a d-justifying assignment for some set of variables W that contains a variable from each of d different subformulas of G. This claim is clearly true initially for $d = 1$ (since $J_1 \subseteq J$). Now consider the d'th iteration of step 1 for $2 \leq d \leq c$. In J_{d-1} there is a $(d-1)$-justifying assignment p for a W as claimed. Let y be some variable from a subformula of G that is not represented by W. After the step 1(b)i we shall either have made q and a agree on all the variables from G's subformulas not yet represented in W, or else we shall have already discovered (and added to J_d) a d-justifying assignment for some $W \cup \{z\}$ where z is taken from one of those previously unrepresented subformulas of G. In the latter case the claim is satisfied. In the former case, it will be true that $f_{q_{y\leftarrow *}}$ depends on y, and hence the claim will also remain true, since we add the d-justifying assignment for $W \cup \{y\}$ to J_d.$\Box$.

This routine requires $O(n^{k+1})$ time and membership queries (ignoring factors polynomial in k or c^k for a constant c, as we shall do throughout this paper).

Now suppose we have some c-justifying assignment p for a set W of $3 \leq c \leq k$ variables where f_p induces a prime function. By Lemma 1, this means that the variables in W must share an lca labeled with a prime function from $\{f : \{0,1\}^{3\leq c\leq k} \rightarrow \{0,1\}\}$ (call this gate G). To apply our learning techniques, we need to determine which other variables are descendants of G. Sometimes, we would also like to partition these descendants according to the subformulas of G in which they appear.

The routine *Walk-Prime-Projection* (Figure 3) takes the partial assignment p along with some variable $x \notin W$ with a justifying assignment a. It walks p towards a while preserving the properties that p is c-justifying for W, and that a is justifying for x. It then looks for new c-justifying assignments that prove x is a descendant of G. If x is a descendant of G, the routine will find such proof and will further identify some variable from W that appears in the same subformula of G as does x. The routine is valid for read-once formulas over arbitrary bases.

Lemma 11 *Suppose f is a read-once formula over an arbitrary basis, and a is a justifying assignment for the variable x. Further suppose p is a prime c-justifying assignment for some set W of $3 \leq c \leq k$ variables not including x and that W includes a variable from each subformula of some gate G in f. Then the routine Walk-Prime-Projection when invoked with a membership oracle for f will either return "w's subformula" if x is a descendant of G in the same subformula as is $w \in W$, or else will return "Not descendant" if x is not a descendant of W.*

Proof:

For the conditions of this lemma to hold, G must be labeled with a prime function of fan-in c.

We claim that any U satisfying the conditions at step 2a must be such that each u_i appears in the same subformula of G as does w_i. This is clearly the case when we chose $u_i = w_i$. Otherwise the fact that we obtain a prime c-justifying assignment for $(W - \{w_i\}) \cup \{u_i\}$ implies that the variables $(W - \{w_i\}) \cup \{u_i\}$ must share an lca in f. Since this set includes at least two variables from W (as $c \geq 3$), that lca must be G. Furthermore u_i must appear in the only subformula of G not represented by $W - \{w_i\}$ (i.e. w_i's). If $a_{U\cup\{x\}\leftarrow *}$ is a prime c-justifying assignment then the variables $U \cup \{x\}$ must share an lca, and since U contains more than one variable this lca must be the same as the lca of U, which is G. Moreover, x must appear in the only subformula of G not represented by the u_i's, which we've just shown is w's. Hence if the routine returns "w's subformula" it is correct.

Walk-Prime-Projection(p, W, a, x)

1. Walk p and a towards each other by repeatedly checking whether there is a variable $y \notin W \cup \{x\}$ for which $p(y) \neq a(y)$, such that either $p_{\neg y}$ is still c-justifying for W (in which case we reset p to $p_{\neg y}$) or $a_{\neg y}$ is still justifying for x (in which case we reset a). Proceed to the next step once no more such changes are possible.
2. Repeat the following for each variable $w \in W$ (we attempt to prove that x appears in the same subformula of $G = \text{lca}(W)$ as does w).
   (a) Let $W = \{w, w_1, \ldots, w_{c-1}\}$. Consider each set of variables $U = \{u_1, \ldots, u_{c-1}\}$ such that for each u_i either
      1. $u_i = w_i$.
      2. $p_{u_i \leftarrow *, w_i \leftarrow 0}$ or $p_{u_i \leftarrow *, w_i \leftarrow 1}$ is a prime c-justifying assignment for $(W - \{w_i\}) \cup \{u_i\}$.

      If $q = a_{U \cup \{x\} \leftarrow *}$ is a prime c-justifying assignment for $U \cup \{x\}$, return "w's subformula".
3. Return "Not descendant".

Figure 3: Subroutine *Walk-Prime-Projection*

Now suppose x is a descendant of G and appears in the same subformula of G as does $w \in W$. Consider a subformula of G that does not contain x or w. After step 1, either a and p agree on all the variables from the subformula except the single $w_i \in W$ to which p assigns $*$, or else there is some variable u_i we could not change in a to agree with p. In either case, the subformula contains some variable y such that the two assignments a and $a_{\neg y}$ induce different values on the subformula's input to G (in the first case, $y = w_i$, and in the second, $y = u_i$). Let U be a set of such variables y, one from each of the $c-1$ subformulas of G not containing x or w (i.e. changing u_i in a changes the value induced on the subformula of G that contains w_i). The partial assignment $q = a_{U \cup \{x\} \leftarrow *}$ induces a non-constant function on each of the inputs to G. Moreover, because a is justifying for x, the projection of f induced by $q = a_{U \cup \{x\} \leftarrow *}$ depends on the output of G. Therefore, the projection induced by $q = a_{U \cup \{x\} \leftarrow *}$ is a prime c-justifying assignment for $U \cup \{x\}$. In order to prove that "w's subformula" is output by the routine, it remains only to show that each of the variables in U satisfy one of the conditions of step 2a.

The first condition is clearly satisfied for each $u_i \in U$ such that $u_i = w_i$. If $u_i \neq w_i$, then p and a ended up disagreeing on u_i in w_i's subformula, and we could not change u_i in p. This implies that $p_{u_i \leftarrow *}$ is $(c+1)$-justifying for $W \cup \{u_i\}$. By fixing w_i so that the induced projection still depends on u_i, we obtain a partial assignment induces a non-constant function on each of the inputs to G, and induces a projection of f that depends on the output of G. This projection is prime, so u_i satisfies the second condition. □

Walk-Prime-Projection requires $O(n^{c-1})$ time and membership queries.

4.3 The algorithm and theorem

We can now prove the main theorem of this section.

Theorem 12 *There is a polynomial time membership and equivalence query identification algorithm for* ROF$\left(\{f : \{0,1\}^{1\leq c\leq k} \to \{0,1\}\}\right)$.

Proof:

Note that the class is projection closed, so by Lemma 4 we may add justifying assignments to the model.

We assume without loss of generality that the gates in the target formula f all compute prime functions. We also assume without loss of generality that all two-input gates in f are labeled with AND, OR, or XOR.

We invoke the routine *Grow-k-Justifying-Assignments* once. We search for a $p \in J$ where 1) f_p is a prime function that depends on at least three variables, and 2) There is no other such p that depends on more variables. By Lemma 10, the first fact implies that the variables W on which f_p depends share an lca G that is labeled with a function from $\{f : \{0,1\}^{3\leq c\leq k} \to \{0,1\}$ and f prime $\}$. The second fact implies that W includes a variable from every subformula of G (if it didn't, J would contain a prime justifying assignment for a larger set of variables that did include one from every subformula). Furthermore there must be such a p unless f contains only gates of fan-in at most two.

If there is no such p then we learn f as described above in Section 4.1. If there is such a p, we call *Walk-Prime-Projection* with each remaining variable. We use the information returned by *Walk-Prime-Projection* to partition the remaining variables into one set containing variables that are not descendants of G, and $|W|$ sets each containing the variables that are in a given subformula of G.

One of the sets in the partition must be non-trivial unless G is the only gate in f. Since learning one gate functions from $f : \{0,1\}^{3\leq c\leq k} \to \{0,1\}$ is simple (there are only a constant number of possibilities), the theorem follows from Lemma 8. $\square$

In the above algorithm, we can avoid re-invoking *Grow-k-Justifying-Assignments* each time we recurse with a partition by using the justifying assignments produced from the invocation of *Grow-k-Justifying-Assignments* at the start of the algorithm. We need a total of at most n^2 invocations of *Walk-Prime-Partition* (fewer than n for each partition), requiring $O(n^{k+1})$ time and membership queries. Adding in the factor n overhead in time and queries to generate justifying assignments and the factor n time to implement membership queries, we can obtain the total resource bounds of $O(n^{k+3})$ time, $O(n^{k+2})$ membership queries, and n equivalence queries.

There is no algorithm for this problem whose running time is polynomial in both n and k (such an algorithm would have to learn an arbitrary boolean function in less time than it takes to output its representation). It may be possible to have an algorithm whose running time is polynomial in n and 2^k, though that would require significant new techniques beyond ours, whose complexity is based on enumerating the possible subsets of variables that may share an lca.

5 Adding the Constant Fan-in Functions to a Basis

In this section we generalize the results of the previous section. We show how to convert an algorithm for learning a read-once formula over a basis B into an algorithm for learning read-once formulas over the basis $B \cup \{f : \{0,1\}^{1\leq c\leq k} \to \{0,1\}\}$. The results of this section, along with the results of Section 6, will imply that there is a polynomial time identification algorithm for the class $\mathrm{ROF}\left(\mathrm{SYMM} \cup \{f : \{0,1\}^{1 \le c \le k} \rightarrow \{0,1\}\}\right)$ (where SYMM is the set of symmetric boolean functions). The reader who is not interested in the generalizations of this section may skip it without any loss of continuity.

To achieve our results we first deal with the functions of fan-in less than three (essentially NOT, AND, OR, and XOR) as special cases. Suppose we have an algorithm for $\mathrm{ROF}(B)$, and suppose that B is projection closed. In Section 5.1 we shall describe an algorithm for learning $\mathrm{ROF}(B \cup \{\mathrm{NOT}, \mathrm{XOR}\})$ (provided B meets an additional technical condition). Then in Section 5.2 we show how to learn $\mathrm{ROF}(B \cup \{\mathrm{AND}, \mathrm{OR}\})$. We conclude in Section 5.3 by describing how to learn $\mathrm{ROF}\left(B \cup \{f : \{0,1\}^{3 \le c \le k} \rightarrow \{0,1\} \text{ and } f \text{ prime }\}\right)$.

The technical condition we require in addition to B being projection closed is that B either contains the function NOT, or it *absorbs negation*, by which we mean that any formula in $\mathrm{ROF}(B \cup \{\mathrm{NOT}\})$ can be rewritten so that all negations appear at the leaves. Note that any symmetric basis that includes the duals of all its elements has this property (e.g. the basis $\{\mathrm{AND}, \mathrm{OR}\}$), as does the basis of constant fan-in functions. If B does not satisfy the technical condition, we are only able to prove that we can add monotone functions of constant fan-in to the basis.

We shall state the intermediate results of this section in terms of exact identification algorithms using membership queries and justifying assignments. Recall that as long as the class is projection closed, an identification algorithm in this model implies one in the membership and equivalence query model by Lemma 4.

5.1 Adding "NOT" and "XOR" to a basis

Lemma 13 *Suppose B is a class of boolean functions for which there is a polynomial time algorithm to identify $\mathrm{ROF}(B)$ using justifying assignments and membership and equivalence queries, If the class $\mathrm{ROF}(B)$ is projection closed and B either contains the function NOT or absorbs negations, then there is a polynomial time algorithm to identify $\mathrm{ROF}(B \cup \{XOR, NOT\})$ also using justifying assignments and membership and equivalence queries.*

Proof:

First we show that we can add NOT to the basis. If it's already in B this is trivial, and if not then the projection closed property implies that every function in B is monotone. In the latter case every formula in $\mathrm{ROF}(B \cup \{\mathrm{NOT}\})$ can be rewritten with all negations at the leaves. The sign of a variable is then indicated by its justifying assignment, and the problem is easily reduced to one of identifying a formula in $\mathrm{ROF}(B)$.

Now assuming we have NOT in the basis, consider adding XOR. If B already includes XOR or its complement this is trivial. Otherwise no projection of a function in B computes XOR or its complement. By Lemma 9, we can use justifying assignments and membership queries to determine for each pair of variables x, y whether the type of $\mathrm{lca}(x, y)$ is XOR. By Lemmas 6 and 7, this reduces the problem of identifying formulas in $\mathrm{ROF}(B \cup \{\mathrm{XOR}, \mathrm{NOT}\})$ to the problems of identifying formulas in $\mathrm{ROF}(B \cup \{\mathrm{NOT}\})$ and formulas in $\mathrm{ROF}(\{\mathrm{XOR}, \mathrm{NOT}\})$. We have already addressed the former problem, and as mentioned earlier the latter problem is trivial since all possible in $\mathrm{ROF}(\{\mathrm{XOR}, \mathrm{NOT}\})$ target formulas are equivalent to a constant, XOR, or $\neg$XOR. $\square$

5.2 Adding "AND" and "OR" to a basis

To show that we can add AND and OR to a basis while preserving identifiability, we first state three technical lemmas. These lemmas imply that non-trivial bases that are projection closed include at least two of $\{\mathrm{AND}, \mathrm{OR}, \mathrm{XOR}\}$.

Lemma 14 *If B is a projection closed class of boolean functions that does not include NOT, and if the only binary function in B is AND, then $B = \{AND\}$.*

Proof:

We show this by induction on the number of variables n. The claim is obvious for $n = 2$. Assume it is true for functions of $n-1$ variables, and consider a function $f(x_1, \ldots, x_n)$ in B over n variables. Now (using xy to denote $\mathrm{AND}(x, y)$),

$$f(x_1, \ldots, x_n) = \mathrm{OR}(f_1(x_2, \ldots, x_n)\neg(x_1), f_2(x_2, \ldots, x_n)x_1)$$

Since $f(0, x_2, \ldots, x_n) = f_1(x_2, \ldots, x_n)$, f_1 is a projection of f, and hence is in B and by induction $f_1(x_2, \ldots, x_n) = x_{i_1} \cdots x_{i_l}$ for some $i_1, \ldots, i_l$. We also have $f_2(x_2, \ldots, x_n) = x_{j_1} \cdots x_{j_m}$. If f_1 and f_2 depend on a common variable, say x_2, then $f(x_1, \ldots, x_n) = x_2 g(x_1, x_3, \ldots, x_n) =$ (by the inductive hypothesis) $x_1 x_2 \cdots x_n$. But if there are no common variables we get a contradiction by substituting $x_{i_1} = \cdots = x_{i_l} = 1$ and $x_{j_1} = 0$ to get the projection $\neg x_1$.$\Box$

A dual argument proves this similar lemma.

Lemma 15 *If B is a projection closed class of boolean functions that does not include NOT, and if the only binary function in B is OR, then $B = \{OR\}$.*

Lemma 16 *If B is a projection closed class of boolean functions that does not include any unate binary functions, then $B \subseteq \{XOR, NOT, \neg XOR\}$.*

Proof:

We show this by induction on the number of variables n. The claim is obvious for $n = 2$. Suppose the claim is true for functions with $n-1$ or fewer variables. As in Lemma 14, we have that

$$f(x_1, \ldots, x_n) = \mathrm{OR}(f_1(x_2, \ldots, x_n)\neg x_1, f_2(x_2, \ldots, x_n)x_1)$$

where here $f_1(x_2, \ldots, x_n) = g_1(\mathrm{XOR}(x_{i_1}, \ldots, x_{i_l}))$ and $f_2(x_2, \ldots, x_n) = g_2(\mathrm{XOR}(x_{j_1}, \ldots, x_{i_m}))$ (where g_1 and g_2 are either the identity function or negation). If f_1 and f_2 contain a common variable, say x_2, then $f(x_1, \ldots, x_n) = \mathrm{XOR}(x_2, f(x_1, 0, x_3, \ldots, x_n))$, which by the inductive hypothesis is $\mathrm{XOR}(x_1, x_2, \ldots, x_n)$ or its complement. Therefore, suppose they have no common elements. Then set $x_{i_1} = \cdots = x_{i_l}$ to 0 or 1 so that the output of g_1 is 1, and set $x_{j_2} = \cdots = x_{j_{m-1}} = 0$. This gives the unate projection $\mathrm{OR}(x_1, g_2(x_{j_1}))$.$\Box$

Now we prove the main result for this section.

Lemma 17 *Suppose B is a projection closed class of boolean functions and there is a polynomial time algorithm to identify* ROF (B) *using justifying assignments and membership and equivalence queries. Then there is a polynomial time algorithm to identify* ROF $(B \cup \{AND, OR\})$ *also using justifying assignments and membership and equivalence queries.*

Proof:

If B includes any two of $\{\text{AND}, \text{OR}, \text{NOT}\}$ then the claim is trivial. Assume this is not the case.

If B does not contain NOT then, because B projection closed, it only includes monotone functions. If, in addition, it contains only one of $\{\text{AND}, \text{OR}\}$ then Lemma 14 or 15 implies that $B = \{\text{AND}\}$ or $\{\text{OR}\}$ respectively (it is not possible for a non-empty B to contain neither).

If B contains NOT then the claim is trivial if B contains any unate binary functions. If B contains no unate binary functions, Lemma 16 implies that $B \subseteq \{\text{NOT}, \text{XOR}, \neg\text{XOR}\}$. In the latter case $B \cup \{\text{AND}, \text{OR}\} \subseteq \{\text{AND,OR,NOT,XOR,}\neg\text{XOR}\}$, and an identification algorithm follows from Angluin, Hellerstein, and Karpinski [AHK93] and Lemma 13.□

5.3 Adding fan-in $3 \leq c \leq k$ functions to a basis

The previous sections dealt with adding the two input functions to the basis. Here we consider adding arbitrary functions of fan-in c for $3 \leq c \leq k$. We assume without loss of generality that all such functions are prime.

In the proof of the following theorem, we make use of the routines *Grow-k-Justifying-Assignments* and *Walk-Prime-Projection* introduced in the previous section. Recall that these routines were valid for read-once formulas over arbitrary bases.

Theorem 18 *Suppose B is a projection closed set of basis functions for which there is a polynomial time membership and equivalence query identification algorithm for* $\text{ROF}(B)$*. If B includes the function NOT or if B absorbs negation, then there is a polynomial time membership and equivalence query identification algorithm for* $\text{ROF}\left(B \cup \{f : \{0,1\}^{1 \leq c \leq k} \rightarrow \{0,1\}\}\right)$*. If B is monotone, there is a polynomial time membership and equivalence query identification algorithm for* $\text{ROF}\left(B \cup \{f : \{0,1\}^{1 \leq c \leq k} \rightarrow \{0,1\} \mid f \text{ is monotone}\}\right)$*.*

Proof:

Note that in both cases the class is projection closed (since B is), so by Lemma 4 we may add justifying assignments to the model. If B absorbs negations or includes NOT we may assume it includes both NOT and XOR by Lemma 13. In any case we may assume that B includes AND and OR by Lemma 17. Thus we need only concentrate on adding the prime functions of fan-in three or more.

The claim is clearly trivial unless f includes some gate G labeled with a function g not representable in $\text{ROF}(B)$. We argue that any projection f_p that depends on exactly one variable from each subformula of G will also not be representable in $\text{ROF}(B)$. This is easily shown to be true if B cannot be augmented to include negation, in which case f is monotone. If f is monotone the only possibility is $f_p \equiv g$. If, however, B includes negation, then since f_p computes a function in $[g]$, any representation in $\text{ROF}(B)$ for f_p could be converted to a representation for g by appropriate placement of negations.

We invoke the routine *Grow-k-Justifying-Assignments* once. As long as f is not representable in $\text{ROF}(B)$, we shall find a $p \in J$ where 1) f_p is a prime function that is not representable in $\text{ROF}(B)$, and 2) There is no other such p that depends on more variables. The first fact implies that the variables W on which f_p depends share an lca that is a function not in B (and hence is in

$f : \{0,1\}^{3 \leq c \leq k} \rightarrow \{0,1\}$), and the second implies that W includes a variable from every subformula of this gate. Lemma 10 implies that J will include such a p if f is not representable in ROF(B).

We use such a partial assignment to partition V by calling *Walk-Prime-Projection* with each remaining variable. (This will split up the remaining variables according to whether they are descendants of the gate, and if so, in which subtree they appear. One of those sets must be non-trivial unless this is the only gate in f.) Since learning one gate functions from $f : \{0,1\}^{3 \leq c \leq k} \rightarrow \{0,1\}$ is simple (there are only a constant number of possibilities), the theorem follows from Lemma 8. □

The complexity of the processing described here is essentially the sum of the costs of our previous algorithm to learn read-once formulas over the fan-in k basis and of repeatedly executing the algorithm to learn ROF(B) (which we potentially multiply by a factor of n overhead to generate the justifying assignments).

The next result will follow from Theorem 39 in Section 6

Corollary 19 *There is a polynomial time algorithm using membership and equivalence queries to exactly identify read-once formulas over arbitrary basis functions that are symmetric or have fan-in k or less.*

The next three results follow from other work. The first is a consequence of the work of Hancock, Golea, and Marchand [HGM91] and the fact that Lemmas 4 and 8 have analogues in the PAC and membership query model of learning. The second is a consequence of the work of Hellerstein and Coullard [HC93] on binary matroid ports (a generalization of switch configurations studied by Raghavan and Schach [RS90]). The third results is a consequence of the work of Raghavan and Wilkins on μ-branching programs.

Corollary 20 *In the PAC model augmented with membership queries there is a polynomial time algorithm for learning read-once formulas over the basis of half-spaces and arbitrary functions of fan-in k or less.*

Corollary 21 *There is a polynomial time algorithm using membership and equivalence queries to exactly identify read-once formulas over the basis of binary matroid port functions (and hence also switch configurations) and arbitrary functions of fan-in k or less.*

Corollary 22 *There is a polynomial time algorithm using membership and equivalence queries to exactly identify read-once formulas over the basis of μ-branching programs and and arbitrary functions of fan-in k or less.*

6 Identification of Read-Once Formulas over a Symmetric Basis

The problem of learning read-once formulas over a symmetric basis presents some difficulties that are not encountered in learning read-once formulas over the basis of gates of fan-in k, or over any of the bases (such as {AND, OR, NOT} and threshold) considered in previous work [AHK93, BHHK91]. By Lemma 8 we can learn read-once formulas over the symmetric basis provided that we can

1. learn those formulas that contain at most one non-unary gate, and

2. either build skeletons or find partitions.

To learn formulas with fan-in k or with threshold gates, problems (1) and (2) can be solved by taking a list of justifying assignments for each of the relevant variables, and asking appropriate membership queries. However, justifying assignments and membership queries provably do not suffice for solving these problems for the symmetric basis in polynomial time. Therefore to learn over the symmetric basis we must use not only membership queries and justifying assignments, but equivalence queries as well. Problem (1) can be solved fairly easily with membership queries and equivalence queries, and we present the solution below in Section 6.1.

Problem (2) is more interesting than Problem (1). The following example proves that it cannot be solved using membership queries and justifying assignments. Consider a partition of the variables $x_1, ..., x_n$ into two equal sets (assume n is even), $\{x_{i_1}, ..., x_{i_{n/2}}\}$ and $\{x_{i_{n/2+1}}, ..., x_n\}$. Let g be the consensus function and consider the formula

$$h = \mathrm{AND}(g(x_{i_1}, ..., x_{i_{n/2}}), g(x_{i_{n/2+1}}, ..., x_{i_n})).$$

Even given an assignment that is justifying for every variable (such as the all 1's assignment), it may take exponentially many membership queries to distinguish this function from $g(x_1, ..., x_n)$.

The technique used in finding partitions in the fan-in k case was to exploit justifying assignments to find c-justifying assignments for subsets of $c \leq k$. These c-justifying assignments provided enough information to find partitions. Sometimes it is possible to use such an approach to find a partition for a ROF over the symmetric basis. In particular, we show in Section 6.4 that if we can produce a projection p assigning all but three relevant variables, then in all of the following cases it is possible to generate a partition using p, membership queries, and justifying assignments (in the second case, this is assuming that f has more than one non-unary gate):

1. f_p is *incomplete.* It depends on some, but not all of the two or three variables.

2. f_p is *prime.* It computes a prime function of its three variables.

3. f_p is *asymmetric.* It is not a symmetric function of literals of its variables.

Unfortunately such projections may not exist. In the previous AND of consensus example any c-justifying assignment for a constant c induces an AND projection (non-prime and symmetric) over the variables or their negations. On the other hand if we knew a priori that f was an AND of consensus gates we could learn it by the following straightforward algorithm. Notice that if a is a positive assignment that sets some non-trivial subset S of the variables V to 1, and the remainder of the variables to 0, then no consensus gate contains inputs from both S and $V - S$. The algorithm begins by making an equivalence query with the hypothesis $g(x_1, ..., x_n)$, where g is the consensus function. If the oracle returns a counterexample a, that counterexample must be a positive one setting some non-trivial subset S of the variables to 0, and the remainder in $V - S$ to 1. This gives us a partition of the variables; the variables in S must be the inputs to some subset of consensus gates, and the variables in $V - S$ must be the inputs to the remainder of the consensus gates. If the target formula is the AND of 2 consensus gates, then the inputs to one of those gates must be S, and the inputs to the other one of those gates must be $V - S$. The algorithm makes an equivalence query with this hypothesis. If a counterexample is obtained, that counterexample must again be a positive example, allowing us to partition the variables further into sets of inputs to distinct consensus gates. The algorithm continues this process until the correct partition is eventually obtained.

Our algorithm to learn read-once formulas over the symmetric basis is a combination of the two approaches of searching for partitions, and of trying to learn f as an AND (or NAND) of consensus gates. Perhaps surprisingly, failure of the AND of consensus algorithm always produces a counterexample that will allow us to find a partition.

We begin in Section 6.1 by presenting the algorithm for learning a formula with at most one non-unary gate. In Sections 6.2 and 6.3 we present the two approaches mentioned above. In Section 6.4 we describe the methods for turning prime, asymmetric, and incomplete projections (assigning values to all but two or three of the variables) into partitions.

6.1 Learning formulas with only one symmetric non-unary gate

If f has a single non-unary gate it computes a symmetric function over some literal setting of the input variables. If we knew what those literal settings were, the identification process would be just to make $n+1$ membership queries on inputs that have from 0 up to n of the gate's inputs set to 1. Our algorithm starts by building such a hypothesis under the uninformed assumption that no variables are negated. A counterexample is then used to determine correct signs for the variables.

The key fact we use is that if two variables x and y have the same sign, then for any assignment a it will be true that $f(a_{x\leftarrow 1, y\leftarrow 0}) = f(a_{x\leftarrow 0, y\leftarrow 1})$. We search for assignments where this is not the case to establish that variables have different signs (signs are not absolute since we can get an equivalent function by negating each input and reversing the vector of the function). The details of the routine are given in Figure 4. The correctness of this routine proves the following lemma.

Lemma 23 *Suppose f is a symmetric function over some literal setting of its inputs. Then the subroutine LearnOneGate when invoked with those variables and membership and equivalence oracles for f will return a hypothesis equivalent to f.*

6.2 Finding an s-assignment (or a projection that yields a partition)

Formally, we say that a formula is an "AND of consensus gates", or ANDC, if its root is labeled with AND, and the children of the root have the following properties: they are either variables (possibly negated), or they are consensus gates whose inputs are variables (possibly negated). We are only interested in read-once ANDCs, and when we discuss properties of ANDCs we are referring implicitly only to those that are read-once. A NAND of consensus gates (NANDC) is the complement of an ANDC (also representable by an OR of consensus gates).

Our algorithm for learning a formula over the symmetric basis with more than one non-unary gate makes extensive use of s-assignments. We say that an assignment a is an *s-assignment* if

1. a is justifying for every relevant variable x and
2. for any pair of variables $\{x, y\}$, $a_{x,y\leftarrow *}$ is a unate two justifying assignment.

LearnOneGate

1. Let a_0 be the all 0's assignment. For $i = 1$ to n, let a_i be an assignment obtained from a_{i-1} by flipping some variable from 0 to 1 (so a_i has i variables turned on). Make an equivalence query on the hypothesis h with vector
$$\langle f(a_0), f(a_1), \ldots, f(a_n)\rangle.$$
If the answer is "yes" output h and halt. Otherwise let c be the counterexample returned.
2. Suppose c has m variables set to 1 (so $f(c) \neq h(a_m) = f(a_m)$). Find a pair of variables x and y on which a_m and c differ for which $c(x) = 1$ and $c(y) = 0$. If $f(c_{x\leftarrow 0, y\leftarrow 1}) = f(c)$, reset c to $c_{x\leftarrow 1, y\leftarrow 0}$ and repeat this step (c remains a counterexample). Otherwise continue.
3. The fact that $f(c_{x\leftarrow 0, y\leftarrow 1}) \neq f(c)$ implies that x and y have opposite signs. A third variable z with $c(z) = 1$ has the same sign as x only if
$$f(c_{z\leftarrow 0, y\leftarrow 1}) = f(c_{x\leftarrow 0, y\leftarrow 1}) \neq f(c)$$
This is a sufficient as well as necessary condition since this implies that z has a different sign from y. An analogous test tells us z's sign if $c(z) = 0$. Find the sign of each variable and construct a new hypothesis h' with the correct vector. Make an equivalence query on h'. If the answer is "yes" output h' and halt. Otherwise return "failure" (f is not symmetric).

Figure 4: Subroutine *LearnOneGate*

Our algorithm begins by obtaining a unate two-justifying assignment. This in fact is already done for us at step 2 of *LearnOneGate*. At that step we march the counterexample c towards the assignment a_m until we find a c for which $f(c) \neq f(c_{\neg x, \neg y})$. Since $f_{c_{x,y\leftarrow *}}$ cannot be constant or compute XOR or its complement given this condition, the only possibilities are that $c_{x,y\leftarrow *}$ is incomplete (in which case we partition) or is a unate two-justifying assignment.

Once a unate two-justifying assignment is found, we try to transform it into an s-assignment. This attempt will either succeed, or we shall find an incomplete, asymmetric, or prime projection (in which case, again, we proceed to find a partition).

If we find an s-assignment, we run an algorithm similar to the one described in the first part of Section 6 above that tries to learn ANDCs and NANDCs, given an s-assignment of the target formula as input. The algorithm either succeeds in learning f (if f is really an ANDC or NANDC), or succeeds in discovering a partition.

6.2.1 Converting a two-justifying assignment to an s-assignment, or finding a partition

We say that a unate two-justifying assignment p *reveals a partition* if, for some third variable u, the projection $f_{p_{u\leftarrow *}}$ is either asymmetric, prime, or incomplete.

Given a unate two-justifying assignment, we first test, for every third variable u, whether $f_{p_{u\leftarrow *}}$ reveals a partition. If not we convert p into an s-assignment, as described in the proof of the following lemma.

Lemma 24 *Suppose f is a read-once formula over a symmetric basis, and p is a unate two-justifying assignment. Then either p can be extended to an assignment a such that a is an s-assignment, or it is possible to generate a two-justifying assignment p' that reveals a partition (in polynomial time using membership queries),*

Proof:

For each variable u, check whether $f_{p_{u\leftarrow *}}$ reveals a partition. If so, we're done.

Suppose that for every variable u given a value by p, $f_{p_{u\leftarrow *}}$ does not reveal a partition. In this case, for every u given a value by p, $f_{p_{u\leftarrow *}}$ must depend on all its variables and compute a symmetric, unate, and non-prime function. Let $f_p = g(x', y')$ where x' and y' are literals of x and y, and where g is symmetric. The function g must be either AND or OR. It follows that for any u distinct from x and y, $f_{p_{u\leftarrow *}} = g(x', y', u')$, where u' is some literal of u. Suppose g is AND (the other case is symmetric). Then because f_p is justifying for x and y, it follows that u' is set to 1 by p. If we extend p by setting x' and y' to 1, then the resulting assignment is justifying for all the variables. Let a be the extension of p. If for every pair of variables z, w, $f_{a_{z,w\leftarrow *}}$ is unate, then a is an s-assignment; it will definitely depend on both z and w because a is sensitive to both. If for some pair of variables z, w, $f_{a_{z,w\leftarrow *}}$ is not unate, then $f_{a_{x,z,w\leftarrow *}}$ or $f_{a_{y,z,w\leftarrow *}}$ is asymmetric, and reveals a partition. □

6.3 Learning f as an ANDC (or finding a projection that yields a partition)

The algorithm described in this section takes as input an s-assignment a of f. Without loss of generality let us assume that a is the all 1's assignment and that $f(a) = 1$ (This assumption will in effect allow us to ignore the issue of negations on the leaves. It is without loss of generality since we can handle other values of a simply by redefining "1" for each variable to be the value that it takes on in a, which in effect means replacing variables that are 0 in a by their negations.) The routine *ANDCPart* in Figure 5 either learns an ANDC or returns a partial assignment p setting all but two or three variables, and for which f_p is prime, incomplete, or asymmetric.

The algorithm in this section is similar to the algorithm sketched at the beginning of Section 6 for learning ANDCs with no negations (that algorithm actually works only if all children of the root are consensus gates. A simple modification handles the more general case in which some of the children are variables). In that algorithm, we begin by asking an equivalence query which gives an initial positive counterexample. Note that in a true ANDC function all positive examples are s-assignments (i.e. flipping any one or two variables gives a negative assignment). If the counterexample is an s-assignment, then we use it to split the inputs into two sets corresponding (we believe) to different groups of consensus gates. The key property we are able to prove is that when we group our variables according to s-assignments, we preserve the fact that only positive counterexamples are possible, even if f is not in fact an ANDC. ¿From this it follows that if f is not an ANDC we shall eventually receive a positive counterexample that is not an s-assignment. By walking this toward our initial s-assignment (all 1's), we prove that we are able to find a unate two-justifying assignment for a pair of variables that cannot be extended to an s-assignment, and hence by Lemma 24 we are able to partition.

We present the algorithm in Figure 5. In the next two sections, we prove lemmas that establish its correctness.

ANDCPart(a)

1. Let $K = \{a\}$. K will hold the set of s-assignments.
2. Let $\mathcal{S} = \emptyset$. $\mathcal{S}$ will hold sets of subsets of variables thought to be inputs to different consensus gates.
3. Let $W = \{x_1, \ldots, x_n\}$. W will contain the variables not in a subset of $\mathcal{S}$, and hence possible inputs to the AND gate.
4. Let h be the hypothesis

$$\text{AND}(g(S_1), g(S_2), \ldots, g(S_k), x_{i_1}, x_{i_2}, \ldots, x_{i_j})$$

   where $S_1, \ldots, S_k$ are the subsets in $\mathcal{S}$, $g(S_i)$ is the consensus function on the variables in S_i, and $x_{i_1}, \ldots, x_{i_j}$ are the variables in W. [Note: Consider $\text{AND}(g(S_1))$ to be equal to $g(S_1)$).]
5. Make an equivalence query on h. If the answer is "yes", output h and halt. Otherwise let b be the counterexample.
6. If b is an s-assignment, then
   (a) If there exist x's in W that are set to 0 by q, let S_{k+1} be the set of such x's and add the set S_{k+1} to $\mathcal{S}$ and reset W to $W - S_{k+1}$. For all S_i in $\mathcal{S}$ such that b sets some of the variables in S_i to 0 and some to 1, replace S_i in $\mathcal{S}$ by two subsets S_{i_1} and S_{i_2} such that S_{i_1} contains the variables in S_i set to 1 in b, and S_{i_2} contains the variables in S_i set to 0 in b.
   (b) Let $K = K \bigcup \{b\}$.
   (c) Go back to step 4.
7. Else repeat forever (we shall prove this loop preserves the fact that b is not an s-assignment).
   (a) Let x and y be such that x and y are set to 0 by b. [There will always be such an x and y]
   (b) If $f(b_{x,y\leftarrow *})$ depends on neither x nor y, then let $b = b_{x,y\leftarrow 1}$.
   (c) Else if $f(b_{x,y\leftarrow *})$ reveals a partition then exit this routine and return$(b_{y,x\leftarrow *})$.
   (d) Else if $f(b_{x,y\leftarrow *})$ is unate, then use the techniques of Lemma 24 to find a projection that reveals a partition. Exit this routine and return the projection.
   (e) Else if $f(b_{x,y\leftarrow *})$ is not unate, let $b = b_{x,y\leftarrow 1}$.

Figure 5: Subroutine *ANDCPart*

6.3.1 Properties of s-assignments

We begin the proof of correctness with some important lemmas characterizing s-assignments.

Lemma 25 *Let f be a read-once formula over the symmetric basis. On an s-assignment, every gate G in f has all its inputs set to the same value.*

Proof:

Suppose there is an s-assignment c such that some gate G in f has one inputs set to 0 by c, and another input set to 1 by c. Let x be a variable whose root leaf path includes this 0 input, and let y be a variable whose root leaf path includes this 1 input. Let $\langle \alpha_0, \alpha_1, \ldots, \alpha_m \rangle$ be the vector of the function computed by G, and let k be the number of inputs of G set to 1 by c. Because c is a justifying assignment for both x and y, $\alpha_{k-1} \neq \alpha_k \neq \alpha_{k+1}$. It follows that $c_{x,y\leftarrow *}$ induces a non-unate projection on G, and therefore a non-unate projection of f, which contradicts the assumption that c is an s-assignment.□

Lemma 26 *Let f be a read-once formula over the symmetric basis. Either the output of f on all s-assignments is the same, or there are only two s-assignments a and b to f, and a is the complement of b.*

Proof:

Lemma 25 implies that for any variable x, the output of f on an s-assignment is completely determined by the value the s-assignment gives to x. Thus if two s-assignments give the same value to some variable x, the output of f must be the same on those two s-assignments.□

Lemma 27 *Let f be a read-once formula over the symmetric basis. Let b be an s-assignment of f. Then if c is an assignment such that the Hamming distance between b and c is one or two, $f(b) \neq f(c)$.*

Proof:

Since b is an s-assignment, it is justifying for every variable. Therefore, if c is Hamming distance one from b, $f(b) \neq f(c)$. Suppose c is Hamming distance two from b. If $f(b) = f(c) = f(b_{\neg x, \neg y})$ we would have the contradiction that $f_{b_{x,y\leftarrow *}}$ is not a unate function (it would have to be XOR or its complement).□

6.3.2 Properties of the algorithm

We will prove that the following invariants hold at the beginning of step 5 of the algorithm *ANDCPart*(a).

Invariants

1. h agrees with f on all the s-assignments in K.
2. If S_i is a subset in $\mathcal{S}$, then there is an s-assignment in K in which all the variables in S_i are set to 1, and an s-assignment in K in which all the variables in S_i are set to 0.

3. For every distinct S_i, S_j pair in $\mathcal{S}$, there is some s-assignment in K setting all the variables in one of them to 0, and all the variables in the other to 1.

4. For every s-assignment in K, the variables in W are all set to 1.

It is easy to show that Invariants 2-4 hold. We will show that Invariant 1 holds using the following Lemma.

Lemma 28 *If the output of f on all possible s-assignments is not the same, then the algorithm will make only one equivalence query. The counterexample to that query will be an assignment b such that b is not an s-assignment and $f(b) = 1$.*

Proof:

By Lemma 26, assignment a (all 1's) is the only positive s-assignment. The initial h has $h(a) = AND(a) = 1$ and no other positive assignments. Therefore the first counterexample b must be a positive assignment other than a, and hence not an s-assignment. Therefore we reach step 7 and return without ever asking another equivalence query. □

Lemma 29 *Invariant 1 is true.*

Proof:

Lemma 28 shows this is trivially true unless the output of f on all possible s-assignments is the same. Suppose this is the case (this output equals $f(a) = 1$). Note that an assignment is only added to K if it's an s-assignment, and h is only updated when an s-assignment is added.

Observe that each change to h caused by splitting an S_i or moving variables from W to $\mathcal{S}$ strictly increases the number of positive examples of h, so we never make h incorrect on an s-assignment in K. Furthermore the changes to $\mathcal{S}$ at step 6b guarantee that h will output 1 on the new s-assignment being added to K.□

We now prove two lemmas to show that each time the algorithm asks an equivalence query with hypothesis h, the counterexample received is a negative example of h. We first present another definition. For all $S_i \in \mathcal{S}$, let Founders(S_i) be the set of nodes G in f (gates or inputs) with the property that all variables in the subformula of f rooted at G are in S_i, and the parent of G has a descendant that is not in S_i.

Lemma 30 *Suppose the output of f on all possible s-assignments is the same. Then during the execution of algorithm ANDCPart, for every S_i in $\mathcal{S}$ and for every gate G in Founders(S_i), the output of G is the same whether all variables in S_i are set to 0, or all variables in S_i are set to 1.*

Proof:

If G is the root of f, this follows from Invariant 2. Suppose G is not the root of f. Let x be a variable in S_i, and let y be a variable not in S_i such that lca(x, y) is the parent of G. Either y is in some S_j not equal to S_i, or y is in W. We consider only the first case (the second is simpler). Let G_1 be the gate that is a sibling of G and has y as a descendant. By Lemma 25, the output of G on an s-assignment is completely determined by the value assigned to x. Let $\sigma_x : \{0, 1\} \to \{0, 1\}$ be such that for $b \in \{0, 1\}$, $\sigma_x(b)$ is the output of G on an s-assignment in which x is set to b. Let

$\sigma_y : \{0,1\} \to \{0,1\}$ be such that for $b \in \{0,1\}$, $\sigma_y(b)$ is the output of G_1 on an s-assignment in which y is set to b. Lemma 25 implies that if there exists an s-assignment which sets x to b_1 and y to b_2 then $\sigma_x(b_1) = \sigma_y(b_2)$. Since K contains the all 1's assignment $\sigma_x(1) = \sigma_y(1)$. If the lemma is not true for G then $\sigma_x(0) \neq \sigma_x(1)$. By Invariant 2, K contains an s-assignment a_{0*} that sets x to 0. Since $\sigma_x(0) \neq \sigma_y(1)$, that s-assignment must also set y to 0, and hence $\sigma_y(0) = \sigma_x(0)(\neq \sigma_x(1))$. By Invariant 3, K contains an s-assignment a' setting x to 1 and y to 0 or vice versa. But since $\sigma_x(0) \neq \sigma_y(1)$ and $\sigma_y(0) \neq \sigma_x(1)$, this contradicts Lemma 25.□

Lemma 31 *Each time the algorithm asks an equivalence query with a hypothesis h, the counterexample b that it receives is such that $h(b) = 0$ and $f(b) = 1$.*

Proof:

If the output of f on all s-assignments is not the same, then this follows directly from Lemma 28.

Suppose the output of f on all possible s-assignments is the same. Let b be a counterexample received by the algorithm. Suppose for the purposes of contradiction that $h(b) = 1$ and $f(b) = 0$.

Think of the truncated formula of f produced by substituting a macro-input for every subformula of f rooted at a gate in Founders(S_i). Since $h(b) = 1$, b must set all variables in S_i to the same value (0 or 1). Therefore by Lemma 30, b sets this macro-input to the same value as does a (which sets all variables to 1). Furthermore since $h(b) = 1$, b must agree with a in setting all variables in W to 1. Since all inputs to the truncated formula are variables in W or gates (or variables) in some Founders(S_i), this implies that the output of this truncated formula is the same on a and b. This contradicts the assumption that b is a counterexample.□

One consequence of Lemma 31 is that we can guarantee termination.

Lemma 32 *The algorithm makes at most n equivalence queries.*

Proof:

Each counterexample on which we repeat the equivalence query loop (rather than reaching step 7) is a positive s-assignment. But each time we add this s-assignment to K either some S_i in $\mathcal{S}$ is split into two sets, or some of the variables not in an S_i are placed in an S_i. This can happen at most n times.□

To complete the proof of correctness, we now show that if a counterexample b is received such that b is not an s-assignment, the algorithm will indeed find a partition.

Lemma 33 *At the start of each iteration of the loop at step 7, $f(b) = 1$, b is not an s-assignment, and there exist variables x and y that are set to 0 by b.*

Proof:

We only iterate the loop in cases where flipping two variables in b do not change $f(b)$ (i.e. where $f_{b_{x,y \leftarrow *}}$ is either constant or non-unate). Thus we preserve $f(b) = 1$. Each change to b is in just two variables. Thus by Lemma 27 we can never make b a positive s-assignment, since that would mean the previous setting for b must have been a negative assignment. We can never make b a negative s-assignment since we preserve $f(b) = 1$. By Lemma 27 the only positive assignment with two or fewer variables set to 0 is a itself, and b never equals a, so it follows that b always has at least three variables set to 0.□

Lemma 34 *If the algorithm asks an equivalence query and receives a counterexample b such that b is not an s-assignment for f, then on some iteration of the loop at step 7, it will return a projection that reveals a partition.*

Proof:

By Lemma 33, on each iteration of the loop there are variables x and y such that b sets x and y to 0. It is clear that on each iteration $f(b_{x,y\leftarrow *})$ satisfies one of the four "if" conditions. Each time b is updated the number of variables that are set to 1 by b increases by two. Thus after at most $n/2$ loop iterations, either $f_{b_{x,y\leftarrow *}}$ reveals a partition, or $f_{b_{x,y\leftarrow *}}$ does not reveal a partition and is unate. In the first case, the lemma is clearly true. In the second case Lemma 24 will produce a partition unless $b_{x,y\leftarrow *}$ can be extended to produce an s-assignment. But Lemma 27 shows there is no positive s-assignment in that range, and Lemma 26 shows that since all those extensions besides b itself set x or y to 1 there is also no negative s-assignment.$\square$

6.4 Techniques for finding partitions

In this section, we present our algorithms to produce partitions given an incomplete projection on two variables, or an asymmetric or prime projection on three variables. This allows us to apply the recursive technique of Lemma 8 to solve our problem by learning two subformulas of f.

6.4.1 Using a prime projection to find a partition

If a projection on a set of variables is prime, those variables must share an lca in f. A prime three-justifying assignment corresponds to a unique gate in the formula that is the lca of its three variables. Condition (1) of Lemma 35 below states a criterion for testing whether any fourth variable is also a descendant of that lca. Repeating this test we can find a partition of variables according to whether they appear in the subformula rooted at the three variables' lca. As long as this lca is not the root of the formula, this gives a non-trivial partition. If on the other hand this lca is the root, then using condition (2) we can split the remaining variables according to the root's subformulas. One of those subformulas must be non-trivial (assuming f has more than one gate). To apply Lemma 35, however, we need our prime three-justifying assignment to satisfy the condition of extensibility to a four (or five)-justifying assignment for the original three variables plus one (or two) more. While this may not be true of our initial three-justifying assignment, we show how to obtain such a extensible partial assignment in Lemma 36.

Lemma 35 *Suppose f is a read-once formula over a symmetric basis, and p is a three-justifying assignment for variables $\{x, y, z\}$ such that f_p is prime. If w is a fourth variable for which $f_{p_{w\leftarrow *}}$ depends on w, then the following two statements are true:*

1. *w is a descendant of lca(x, y) if and only if there are no functions g_1 and g_2 such that*

$$f_{p_{w\leftarrow *}} \equiv g_2(w, g_1(x, y, z))$$

2. *lca(x, w) is a descendant of lca(x, y) if and only if*

$$f_{p_{w\leftarrow *}} \equiv g_1(g_2(x, w), y, z))$$

where g_1 is a prime symmetric function.

Proof:

The only if directions of both claims follow readily from the topology of the formula. Claim (2) follows from the fact that a function of the given form cannot be expressed as a read-once formula without $\mathrm{lca}(x, w)$ being below $\mathrm{lca}(x, y, z)$. And claim (1) is true because if w is a descendant of $\mathrm{lca}(x, y)$ then (depending whether w appears in the same subformula of that gate as one of the other three variables) $f_{p_{w\leftarrow *}}$ must have either the form $g(w, x, y, z)$ where g is symmetric and prime (since it has a prime projection) or the form $g_1(x, y, g_2(w, z))$ (or likewise with w paired with x or y) where g_1 is prime. Neither type of projection can be expressed in the form of the lemma. □

To find a p as called for in the above lemma we prove the following result.

Lemma 36 *Suppose p is a c-justifying assignment for a set of c variables appearing in some read-once formula f over a symmetric basis. There is a polynomial time (if c is constant) membership query algorithm that takes a different variable w and a justifying assignment for w and produces a partial assignment p' such that $f_p \equiv f_{p'}$ and such that $f_{p'_{w\leftarrow *}}$ depends on w.*

Proof:

To find p' we use the standard greedy walk approach, starting with p and trying to flip a single bit (plus possibly w) to make p agree with a justifying assignment for w, while not changing the projection f_p. If we flip all the bits then, since the partial assignment agrees with a justifying assignment for w on every variable it assigns a value, we shall surely succeed. And we may succeed at some point along the way. The claim is that we cannot get stuck along the walk.

To show this, consider an f_p for which $f_{p_{w\leftarrow *}}$ does not depend on w. Let G be the deepest gate of f whose input depends on w in $f_{p_{w\leftarrow *}}$, but whose output does not. None of the unassigned variables in p can be descendants of G (since the symmetry of G implies that $f_{p_{w\leftarrow *}}$ can not depend on such variables without depending on w). There must be some subformula of G other than w's that contains a variable u on which p differs from w's justifying assignment. If flipping u changes f_p then (1) flipping u changes p's value on this input to G, and (2) p induces different values on the subformulas of G that contain u and w. In this case, flipping both u and w in p therefore leaves f_p unchanged. Thus we continue moving p closer to w's justifying assignment until we reach a p with the desired property.

This algorithm requires at $O(n^2)$ time and membership queries. □

6.4.2 Using an asymmetric projection to find a partition

Lemma 37 *There is a polynomial time routine to take an asymmetric (non-prime) three-justifying assignment and return a subset of variables that are exactly those appearing in some non-trivial subformula of f.*

Proof:

An asymmetric three-justifying assignment p induces a projection of the form $g_1(z', g_2(x', y'))$ where x', y' and z' are literals and where $g_1, g_2 \in \{\mathrm{AND}, \mathrm{OR}, \mathrm{XOR}\}$ with $g_1 \neq g_2$. Here we shall consider just the case where f_p is $\mathrm{AND}(z, \mathrm{OR}(x, y))$ (the other cases follow similarly).

We start by considering each remaining variables w, and finding (by Lemma 36) a partial assignment p' such that $f_{p'} \equiv f_p$ and $f_{p'_{w\leftarrow *}}$ depends on w. The hope is that such a projection will allow us to decide whether or not w appears in the subformula of f rooted at $\text{lca}(x, y)$, but unfortunately things may not be quite that easy.

Let $q = p'_{w \leftarrow *}$. If w is a descendant of $\text{lca}(x, y)$ then f_q must be one of the following functions (though possibly with w negated).

1. $\text{AND}(z, \text{OR}(x, y, w))$
2. $\text{AND}(z, g(x, y, w))$ where g is prime.
3. $\text{AND}(z, \text{OR}(g(x, w), y))$ or $\text{AND}(z, \text{OR}(x, g(y, w)))$ (where g is AND or XOR)

Cases 2 and 3 definitively establish that w is a descendant of $\text{lca}(x, y)$, since no other topology can induce such a projection. However if $\text{lca}(x, y)$ and $\text{lca}(x, z)$ are not immediate neighbors we can be misled if $\text{lca}(x, w)$ is between $\text{lca}(x, y)$ and $\text{lca}(x, z)$. Such a w can possibly "collapse down" to give a projection of case 1.

We would naively like to partition the variables by saying that the inputs to a subformula can be found by taking the set W of those variables that we've found are definitely descendants of $\text{lca}(x, y)$, as well as those variables for which we have ambiguous projections as in case 1. The set W will include all descendants of $\text{lca}(x, y)$ and only variables whose lca with x is strictly below $\text{lca}(x, z)$. This could fail, however, if W includes some w with $\text{lca}(x, w)$ above $\text{lca}(x, y)$, but not some v with $\text{lca}(x, v)$ between $\text{lca}(x, y)$ and $\text{lca}(x, w)$. We catch this case with the following processing.

For each v not in W and each w in W because of a projection of case 1, find a five-justifying assignment for $\{x, y, z, v, w\}$ by Lemma 36. For this v if $\text{lca}(v, w)$ is indeed below or equal to $\text{lca}(x, w)$ then the five-justifying assignment must induce a projection of one of the following forms, possibly with v negated (recall that v is not a descendant of $\text{lca}(x, y)$ as otherwise it would already be in W):

1. $\text{AND}(z, \text{OR}(x, y, w, v))$.
2. $\text{AND}(z, \text{OR}(x, y, g(w, v)))$ where g is AND or XOR.
3. $\text{AND}(z, g(v, w, \text{OR}(x, y)))$ for some prime symmetric function g.
4. $\text{AND}(z, \text{OR}(w, g(v, \text{OR}(x, y))))$ where g is AND or XOR.

In the first case we can add v to W and repeat this processing. In the second case we can restart our search for a partition based on an asymmetric three-justifying assignment for $\{x, w, v\}$ (this can be viewed as progress since each time we do this the higher of our two lca's has been moved deeper in the formula, and hence the number of times we restart is bounded by the depth of the formula). In the third case we can extract a prime three-justifying assignment, and partition as above. And in the fourth case we can restart using an asymmetric three-justifying assignment for $\{v, x, y\}$.

We continue the above processing (restarting at most n times). Each time we grow W repeatedly until no further extensions are possible. At that point W must contain exactly those variables that are descendants of some one of the gates on the path from $\text{lca}(x, y)$ (inclusive) to $\text{lca}(x, z)$ (exclusive), and we have succeeded. $\Box$

For each of the n (or fewer) restarts we need to obtain at most n^2 five-justifying assignments (for our original three variables, plus some other two). This is the most expensive part of the processing, and gives us bounds of $O(n^4)$ time and membership queries.

6.4.3 Using an irrelevant variable to find an asymmetric projection

We show that an incomplete projection can be manipulated to find an asymmetric projection, from which we find a partition as above.

Perhaps the most basic evidence of asymmetry in f is a partial assignment p that assigns values to all but two variables such that the induced projection f_p depends on one but not both of its two inputs. In this section we show how such a projection (along with a justifying assignment for the irrelevant variable) can be manipulated to produce an asymmetric projection on three variables. This then can be used to partition by the techniques of the previous section.

Lemma 38 *Let f be a read-once formula over a symmetric basis. Suppose partial assignment p assigns values to all variables but $\{x, y\}$, and f_p depends on x but not y. There is a polynomial time algorithm using membership queries that takes as input p and a justifying assignment for y, and returns an asymmetric three-justifying assignment for some set of variables $\{x, y, z\}$.*

Proof:

The algorithm uses a technique very similar to that of Lemma 36 to modify p to get a p' for which flipping one more variable z makes it depend on y.

For each z where $p(z) \neq a(z)$ we consider $p' = p_{\neg z}$. If we find a z for which $f_{p'}$ still depends on x and not y we update p to p' and continue. If we find a $f_{p'}$ that depends on y we have succeeded since the projection $f_{p_{z \leftarrow *}}$ depends on all three variables and is asymmetric (since it has an asymmetric projection f_p). We continue this process greedily as long as possible. If we change all bits in p to agree with a then clearly we will get dependence on y and have succeeded. Thus we must show only that p does not get stuck part way there.

Since each f_p does not depend on y, we know that p must block y below $\mathrm{lca}(x, y)$. Let G be the blocker gate, so a and p disagree on some variable z from other subtrees of G. The only way we cannot either move p closer to a or return is if $f_{p_{\neg z}}$ is constant. This can happen only if $p_{\neg z}$ forces the output of G to the opposite value from what it is forced to by p (in both cases regardless of y). This would imply that on $p_{z,y \leftarrow *}$, the output of G depends on one input but not another, which contradicts the symmetricity of that gate. Thus as long as y is blocked below $\mathrm{lca}(x, y)$ there is some z we can change in p to move closer to x (if we cannot return successfully). And if y ever becomes unblocked we shall return successfully.

The routine runs in $O(n^2)$ time and uses $O(n^2)$ membership queries.□

6.5 The main theorem

We now state our main result for Section 6.

Theorem 39 *There is a polynomial time identification algorithm using membership and equivalence queries for the class of boolean read-once formulas over the basis of arbitrary symmetric functions. The algorithm requires $O(n^6)$ time, $O(n^6)$ membership queries, and at most n^3 equivalence queries.*

Proof:

If the target formula f has a single non-unary gate this follows from Lemma 23. If f is an ANDC (or NANDC, symmetrically) this follows from Lemma 31, Lemma 32, and the fact that every positive example is an s-assignment. Otherwise, this follows from Lemma 34, the partitioning techniques of Section 6.4, and Lemma 8.

The bound on time and membership queries comes from the fact that we may find up to n partitions of the target formula before we terminate, and each such partitioning phase requires up to $O(n^4)$ time and queries. An additional factor of n comes from the technique to find justifying assignments (Lemma 4). Routines *LearnOneGate* and *ANDCPart* will each be invoked at most n times. The latter routine makes up to n equivalence queries. Combined with the factor of n for obtaining justifying assignment, this gives us the n^3 equivalence query bound. □

7 Negative Results For Read-Twice Formulas

Here we show that given common cryptographic assumptions, the results of this paper cannot be generalized to achieve polynomial time identification algorithms for boolean formulas over these bases that have at most *two* appearances of each variable. First we show that the two appearance case is no easier than the general case, and then we appeal to a previous result stating that the general case is hard.

Lemma 40 *If B is a basis that contains AND and the consensus function on three inputs, then there is a polynomial time membership and equivalence query algorithm to identify read-twice formulas over B only if there is one to identify arbitrary formulas over B.*

Proof:

Suppose f is an arbitrary formula over B defined on $V_n = \{x_1, \ldots, x_n\}$. Suppose f has s leaves. We shall create a read-twice formula f' defined over the variable set $V' = \{x_{i,j} \mid 1 \leq i \leq n, 0 \leq j \leq 2s\}$ (the idea is that each $x_{i,j}$ will be a copy of x_i). Let $\hat{f}$ be f with each appearance of x_i replaced by a distinct $x_{i,j}$ where j is odd (so $\hat{f}$ is read-once). Let f' be an AND of $\hat{f}$ as well as ns consensus gates defined as follows. For each $1 \leq i \leq n$, $1 \leq j \leq s$, there is a consensus gate with the three inputs $x_{i,2j-2}$, $x_{i,2j-1}$, and $x_{i,2j}$.

Note that f' is read-twice. If j is odd, $x_{i,j}$ appears (at most) once in $\hat{f}$ and once as an input to a consensus gate. If j is even $x_{i,j}$ appears as an input to two consensus gates.

Let $\mathcal{A}$ be an algorithm to identify read-twice formulas over B. We use $\mathcal{A}$ to identify f' over V' (and hence f over V) by simulating oracles for f' using oracles for f.

For any membership query on an assignment a' over V', first check whether for some i, j, k we have $a'(x_{i,j}) \neq a'(x_{i,k})$. If so, return "0". This condition implies that there is some $j \leq l < k$ for which $a'(x_{i,l}) \neq a'(x_{i,l+1})$, which means there is some consensus gate in f', of which those two variables are inputs, whose output is set to 0 by a'. Otherwise (in which case all the consensus gates in f' output 1 and $f'(a') = \hat{f}(a')$), return the value of $f(a)$ where a is the assignment over V_n that has $a(x_i) = a'(x_{i,1})$ $(= a'(x_{i,2}) = \ldots = a'(x_{i,2s}))$.

If $\mathcal{A}$ makes an equivalence query on h', make an equivalence query "$h \equiv f$?" where h is the formula obtained from h' be replacing each $x_{i,j}$ with x_i. If the answer is "yes", we're done. Otherwise for the counterexample c, give $\mathcal{A}$ the counterexample c' obtained by setting $c'(x_{i,j})$ to $c(x_i)$ for each $1 \leq i \leq n$ and $1 \leq j \leq 2s$ (note that $f'(c') = \hat{f}(c') = f(c) \neq h(c) = h'(c')$).

When $\mathcal{A}$ produces a formula equivalent to f', it can be converted to a formula equivalent to f as above. Since the size and number of variables of f' are polynomial in the size and the number of variables of f, this simulation takes only polynomial overhead. Thus we have reduced the problem of identifying arbitrary formulas to identifying read-twice formulas.□

A dual of this lemma with OR instead of AND also holds. Combining this with the results of Angluin and Kharitonov [AK91], we obtain the following.

Theorem 41 *If we assume the intractability of any of the following three problems: testing quadratic residues modulo a composite, inverting RSA encryption, or factoring Blum integers, then there is no polynomial time algorithm that uses membership and equivalence queries to identify read-twice formulas over any basis that includes AND, OR, NOT, and the three-input consensus function.*

Note that such bases include both the fan-in k basis (for $k \geq 3$) and the symmetric basis. If we allow only fan-in two basis functions (e.g. the basis $\{\mathrm{AND}, \mathrm{OR}\}$), it is unknown whether read-twice formulas are identifiable (although under the same cryptographic assumptions it is known that read-thrice formulas are not [AK91]).

8 Conclusions

The learning algorithms presented in this paper show that the class of read-once formulas is learnable in the membership and equivalence query model for much more powerful families of basis functions than one might initially suspect. This leads to the interesting open question of whether it is in fact possible to perform efficient exact identification of $\mathrm{ROF}(B)$ for *any* basis B for which the class B has an efficient exact identification algorithm (e.g. a basis of Horn sentences [AFP90] or of k-term DNF formulas [BR92]). Or alternatively, can a counterexample to this conjecture be found?

Another key open question is that mentioned above of learning read-twice formulas. While this is not possible over our basis, this remains open for simpler bases such as $\{\mathrm{AND}, \mathrm{OR}\}$. Although this problem is solved for depth two formulas (i.e. DNF or CNF) [Han91, AP91], it remains open for depth three or greater, even in the monotone case. Also learning the conjunction or disjunction of two read-once formulas is an open problem.

Another open question mentioned at the end of Section 4 is whether the result for the fan-in k basis can be extended to handle fan-in of $O(\log n)$ (i.e. can the running time be improved from $poly(n^k)$ to $poly(n, 2^k)$)? Our approach has n^k performance due to implicitly enumerating subsets of k out of n variables, so a qualitative improvement to parts of the algorithm would be required.

References

[AP91] H. Aizenstein and L. Pitt. Exact learning of read-twice DNF formulas. In *32nd Annual Symposium on Foundations of Computer Science*, pages 170–179, October 1991.

[A88] D. Angluin. Queries and concept learning. *Machine Learning*, 2(4):319–342, 1988.

[AFP90] D. Angluin, M. Frazier, and L. Pitt. Learning conjunctions of horn clauses. In *31st Annual Symposium on Foundations of Computer Science*, pages 186–192, October 1990.

[AHK93] D. Angluin, L. Hellerstein, and M. Karpinski. Learning read-once formulas with queries. *Journal of the Association for Computing Machinery*, 40:185–210, 1993.

[AK91] D. Angluin and M. Kharitonov. When won't membership queries help? In *Proceedings of the Twenty Third Annual ACM Symposium on Theory of Computing*, pages 444–454, 1991.

[BR92] A. Blum and S. Rudich. Fast learning of k-term DNF formulas with queries. In *Proceedings of the Twenty Fourth Annual ACM Symposium on Theory of Computing*, pages 382–389, May 1992.

[BHH92] N. H. Bshouty, T. Hancock, and L. Hellerstein. Learning arithmetic read-once formulas. In *Proceedings of the Twenty Fourth Annual ACM Symposium on Theory of Computing*, pages 370–381, 1992.

[BHH92a] N. H. Bshouty, T. Hancock, and L. Hellerstein. Learning Boolean Read-Once Formulas with Arbitrary Symmetric and Constant Fan-in Gates. In *Proceedings of the Fifth Annual ACM Workshop on Computational Learning Theory (COLT 92)*, pp. 1–15, 1992.

[BHHK91] N. H. Bshouty, T. Hancock, L. Hellerstein, and M. Karpinski. Read-once threshold formulas, justifying assignments, and transformations. Technical report, International Computer Science Institute TR-92-020, 1991. To appear, *Computational Complexity.*

[Han90] T. Hancock. Identifying μ-formula decision trees with queries. In *Proceedings of the Third Annual ACM Workshop on Computational Learning Theory (COLT 90)*, pages 23–37, 1990.

[Han91] T. Hancock. Learning 2μ DNF formulas and $k\mu$ decision trees. In *Proceedings of the Fourth Annual Workshop on Computational Learning Theory*, pages 199–209, August 1991.

[HGM91] T. Hancock, M. Golea, and M. Marchand. Learning nonoverlapping perceptron networks from examples and membership queries. Technical report, Harvard University TR-26-91, 1991. To appear, *Machine Learning.*

[HH91] T. Hancock and L. Hellerstein. Learning read-once formulas over fields and extended bases. In *Proceedings of the Fourth Annual ACM Workshop on Computational Learning Theory (COLT 91)*, pages 326–336, 1991.

[HC93] L. Hellerstein and C. Coullard. Learning Binary Matroid Ports. Unpublished manuscript.

[RS90] V. Raghavan and S. R. Schach. Learning switch configurations. In *Proceedings of the Third Annual ACM Workshop on Computational Learning Theory (COLT 90)*, pages 38–51, 1990.

[RW93] V. Raghavan and D. Wilkins. Learning μ branching programs with queries In *Proceedings of the Sixth Annual ACM Workshop on Computational Learning Theory (COLT 93)*, 1993.