Exact Learning of Read-Twice DNF Formulas (extended abstract)

Howard Aizenstein*
Department of Computer Science
and College of Medicine
University of Illinois
Urbana, Illinois 61801

Leonard Pitt†
Department of Computer Science
University of Illinois
Urbana, Illinois 61801

Abstract

A polynomial-time algorithm is presented for exactly learning the class of read-twice DNF formulas *— Boolean formulas in disjunctive normal form where each variable appears at most twice. The (standard) protocol used allows the learning algorithm to query whether a given assignment of Boolean variables satisfies the DNF formula to be learned (membership queries), as well as to obtain counterexamples to the correctness of its current hypothesis which can be any arbitrary DNF formula (equivalence queries). The formula output by the learning algorithm is logically equivalent to the formula to be learned.*

1 Introduction

The Problem

A central open problem in the theory of learning is whether or not the class of DNF formulas can be learned in polynomial time with respect to any of a number of reasonable learning protocols. A number of algorithms have been given for learning restricted subclasses of DNF formulas, including algorithms for learning monomials, monotone DNF formulas, kCNF formulas, kDNF formulas, and Horn sentences. [1, 2, 9, 10]. Algorithms have also been given for learning DNF when examples are chosen from particular probability distributions (e.g., the uniform distribution) [7, 8, 11]).

In this paper we give a polynomial-time algorithm for learning *read-twice DNF formulas* (those formulas in disjunctive normal form where each variable appears at most twice). Let f denote the *target* read-twice DNF formula to be learned. The following protocol is used: The learning algorithm may propose as a hypothesis any DNF formula H by making an *equivalence query* to an oracle. (In the literature, this has sometimes been referred to as an "extended" equivalence query, as the hypothesis of the algorithm need not be in read-twice form.) If H is logically equivalent to f then the answer to the query is "yes" and the learning algorithm has succeeded and halts. Otherwise, the answer to the equivalence query is "no" and the algorithm receives a *counterexample* – a truth assignment of the variables that satisfies f but does not satisfy H (a *positive* counterexample) or vice-versa (a *negative* counterexample). A *membership query* is any assignment a to the variables, and the answer to the membership query is "yes" if a satisfies the target formula, and "no" otherwise. In Section 4 we sketch the proof of our main theorem:

Theorem 1 *If f is a read-twice DNF formula over n variables, then learn-read-twice-dnf (Figure 1) will find an equivalent DNF formula H in $O(n^6)$ time, using $O(n^6)$ membership queries, and using $O(n^2)$ equivalence queries (each of which may be an arbitrary DNF formula).*

Read-twice DNF ??

Read-twice DNF is a particularly interesting subclass of DNF for a number of reasons. By a recent result involving reductions among learning problems [4, 5] it is known that if the class of read-*thrice* DNF formulas can be learned according to the exact learning protocol used here, then in fact the class of general DNF formulas can be learned in the distribution-free (PAC) model of learning introduced by Valiant [10], provided the algorithm is also allowed to make membership queries. Further, assuming that secure public-key cryptosytems exist, the membership queries may

*Email: aizen@cs.uiuc.edu. Telephone: (217) 384-8119

†Supported in part by NSF Grants IRI-8809570 and IRI-9014840. Author's email address: pitt@cs.uiuc.edu. Telephone: (217) 244-6027

be eliminated to obtain a "bounded polynomial prediction" algorithm [4] for DNF, resolving the question of whether DNF formulas are learnable in a reasonable model of feasible inference. The class of read-twice DNF is thus a most general subclass of DNF whose learnability remains open and is not equivalent to that of the general DNF learning problem.

The learnability of read-*once* DNF formulas in the learning model presented here follows from the more general result of [3], showing that read-once Boolean formulas are exactly learnable using equivalence and membership queries. Thus our result closes the gap between read-once DNF (known learnable) and read-thrice DNF (equivalent to the general DNF learning problem). More importantly, read-twice formulas appear to be qualitatively different from their read-once counterparts with respect to learnability: In a read-once formula, the interaction that a given variable may have with other variables is very restricted, and it is possible to obtain information about the role that a variable x plays in the syntactic structure of the formula by first determining (by selective querying) the influence that x has on the function. However, in the case where x may occur two or more times, its different occurrences can interact with other variables in such a way that it becomes more difficult to isolate the role played by a particular occurrence. Consequently, the algorithm presented here is more involved, and incorporates some new techniques to resolve issues that may arise in the case of general DNF.

Related work

A number of algorithms for learning other restricted subclasses of DNF were pointed out in the first paragraph of this abstract. Here we briefly discuss the two most relevant recent results. Angluin, Hellerstein, and Karpinksi give an algorithm for exact learning *arbitrary* read-once Boolean formulas using equivalence and membership queries [3]. This work was motivated by theirs, and extends one of the basic techniques (that of iteratively flipping bits of a positive example to obtain a positive-negative example pair whose Hamming distance is one.)

More relevant is the recent independent result of Hancock [6], giving an algorithm for PAC-learning read-twice DNF using randomly generated examples and membership queries. In the PAC model, only an approximately correct formula need be found with high probability. The result here is stronger: using standard transformations [1], the equivalence queries can be replaced with randomly generated examples from an arbitrary distribution while meeting the definitions of PAC learning. However, our algorithm is more involved, due to the more demanding criterion of exact identification. Both algorithms have time complexity $O(n^6)$.

Hancock suggests that one reason read-twice formulas (even in the case of DNF) are more difficult to learn using the exact learning protocol than using the PAC protocol, is that there "is no longer necessarily a unique representation (up to certain isomorphisms) for the target function." Typically, exact learning algorithms are designed to infer such unique representations. The additional complexity of our algorithm seems to confirm Hancock's intuition that exact learning in the absence of such "normal forms" is significantly more involved than PAC-learning. Further, unlike most exact learning algorithms, our deterministic algorithm does not always find the same (logically equivalent) formula given a particular target function — the (logically equivalent) formula learned may depend on the counterexamples obtained during the learning process.

2 Definitions and Properties

A *literal* is either a variable x or its negation $\bar{x}$. A *term* is a set of literals. A DNF formula is a set of terms. An assignment a is a function $a : \{x_1, x_2, \ldots x_n, \bar{x}_1, \bar{x}_2, \ldots, \bar{x}_n\} \rightarrow \{0,1\}$ such that for each i, $\{a(x_i), a(\bar{x}_i)\} = \{0,1\}$. Assignment a satisfies term t (also written $t(a) = 1$) iff $(\forall x \in t)\ a(x) = 1$. Assignment a satisfies DNF formula f (written $f(a) = 1$) iff $(\exists t \in f)\ t(a) = 1$. Although a DNF formula f is technically a set of terms, we sometimes write "x is in f" to indicate that literal x appears in some term t of f. When it is important to emphasize the distinction, we will write *lits*(f) to denote $\cup\{t : t \in f\}$. Throughout the rest of this abstract, let f denote the read-twice DNF formula to be learned. Without loss of generality we assume that f is *reduced*: each term of f is a minterm of f, and no term of f is implied by the disjunction of the others.

Denote assignment a with literal x fixed to 0 as $a_{x \leftarrow 0}$. Throughout, $a_{x \leftarrow 0}$ will represent a with the truth value of x flipped, since we will only write "$a_{x \leftarrow 0}$" when a assigns x to 1. The *sensitive set* (of literals for a) is defined by $S(a) = \{x : f(a) \neq f(a_{x \leftarrow 0})\}$, thus flipping the value of any $x \in S(a)$ causes f to change values. If $S(a) \neq \emptyset$ then a is a *sensitive assignment.* If $f(a) = 1$ and $x \in S(a)$ then a is a *positive sensitive assignment for* x.

The literals in f fall into two classes. A literal x is *biform* (in f) if x and $\bar{x}$ both occur in f; x is *unate* (in f) if x occurs in f and $\bar{x}$ does not occur in f. If t is a term of f, then *unate*(t) refers to the set of literals in t which are unate (in f). If the literal x appears only once in f, then t_x refers to the unique term in f containing the literal x. (Note if x is biform then t_x and $t_{\bar{x}}$ are defined.) The following definitions and properties are used throughout the paper.

Definition 1 *A term t is* almost satisfied *by assignment a with respect to literal x iff x is the only literal in t assigned 0 by a.*

Definition 2 *Assignment a is a* valid test *for y with respect to x iff a is a positive sensitive assignment for x and if y is biform then $t_{\bar{y}}(a_{x\leftarrow 0, y\leftarrow 0}) = 0$.*

Property 1 *If f is any DNF formula and a is an assignment such that $f(a) = 1$, then $S(a) \subseteq \cap\{t : t(a) = 1\}$.*

Property 2 *If f is any DNF formula and a is a positive sensitive assignment for x, then for any literal y, $y \in S(a_{x\leftarrow 0})$ if and only if there exists a term t such that t is almost satisfied by assignment $a_{x\leftarrow 0}$ with respect to literal $\bar{y}$.*

Property 3 *If f is any DNF formula, a is a positive sensitive assignment for x, and $y \in S(a_{x\leftarrow 0})$, then $\bar{y} \in lits(f)$.*

Property 4 *If a is an assignment and t is the only term satisfied by a, then unate$(t) \subseteq S(a)$.*

The remainder of the properties apply only to read-twice DNF formulas f.

Property 5 *If a is a positive sensitive assignment for biform literal x, then t_x is the only term satisfied by a, and $S(a) \subseteq t_x \subseteq S(a) \cup (S(a_{x\leftarrow 0}) - \{\bar{x}\})$.*

Property 6 *If a is a positive sensitive assignment then a satisfies one or two terms from f.*

Property 7 *If a is an assignment such that r and s are the only terms satisfied by a, then $S(a) = r \cap s$.*

3 Intuition behind the algorithm

The justification for some sections of code is embodied in lemmas that do not appear in this extended abstract due to space constraints. We focus on the basic ideas behind the algorithm. We begin with a simple algorithm for learning read-once DNF formulas using membership and equivalence queries. We then discuss ways in which the simple algorithm is extended to give a polynomial-time algorithm for read-twice DNF formulas.

It is easily argued that if f is a read-once DNF formula and a is a positive sensitive assignment then a satisfies exactly one term of f, and the term is exactly the set $S(a)$. The idea behind the simple algorithm is to iteratively find positive sensitive assignments a, and for each found, add the term $S(a)$ to the current hypothesized DNF. Initially, the algorithm obtains a positive and negative example of f (*pos* and *neg*) by conjecturing the constant DNF expressions FALSE and TRUE, respectively. The first positive sensitive assignment is found by iteratively flipping bits in *pos* to agree with the bits of *neg* until some intermediate assignment a is found such that for some literal x, $f(a) = 1 \neq 0 = f(a_{x\leftarrow 0})$, the values of f being obtained by membership queries. Then $S(a)$ is a term of the target formula f. We refer to this technique as "walking" *pos* towards *neg*.

The simple algorithm finds subsequent terms of f by finding positive sensitive assignments *that do not satisfy any of the terms learned so far.* If H is the current hypothesis for f, then inductively H is exactly a subset of terms of f, and any counterexample returned by an equivalence query must be a positive counterexample p that satisfies a term of f not yet in H. If p is a sensitive assignment, then $S(p)$ is a new term of f. Otherwise, p must be walked towards *neg* so as to obtain a positive sensitive assignment a, and then $S(a)$ is added to H. However, for this to work we must guarantee that a does not satisfy some term *already* in H. The trick is when walking from p towards *neg*, we only flip literals *not* appearing in H, so all terms in H remain unsatisfied. It is not difficult to show that during such a walk a positive sensitive assignment must be encountered.

Extension to read-twice DNF formulas

The extension to read-twice DNF formulas is not straightforward, and does not lend itself to an accurate high-level description within the allotted space. Section 4 contains the skeleton of the proof of Theorem 1. Here we give some intuitions about the algorithm, which appears in Figure 1.

A fundamental problem in extending the simple algorithm for learning read-once DNF formulas to the read-twice case is that for a positive sensitive assignment a, it is not necessarily the case that $S(a)$ is a term of the target formula f. However, it is

Figure 1: The Algorithm

H is a collection of pairs $\langle h, pos(h)\rangle$ where h is a term and $pos(h)$ is a positive example of f that satisfies h. For notational convenience we sometimes treat H as the DNF formula whose terms are the first elements of the pairs in H. Thus when we write equivalence-query(H), or $H(v)$, H refers to the DNF formula.

learn-read-twice-dnf

1. $H \leftarrow \phi$ /* Current hypothesis */
   $A \leftarrow \phi$ /* literals we already have a term for */
2. $n \leftarrow$ equivalence-query($TRUE$)
3. $v \leftarrow$ equivalence-query(H)
4. IF v is "true" THEN HALT
5. ELSE /* v is a counterexample */
6. IF $H(v) = 0$ THEN
   /* v is a positive counterexample, $f(v) = 1$ */
6a. $(a, x) \leftarrow$ fsa-positive-counterexample(H, v, n)
6b. grow-term(H, a, x)
7. IF $H(v) = 1$ THEN
   /* v is a negative counterexample, $f(v) = 0$ */
7a. choose $\langle h, pos(h)\rangle$ from H such that $h(v) = 1$
7b. $(a, x) \leftarrow$ fsa-negative-counterexample($H, pos(h), v$)
8. IF a is a positive sensitive assignment for x
   THEN grow-term(H, a, x)
9. ELSE
9a. remove $\langle h, pos(h)\rangle$ from H
9b. IF h is unmarked THEN $\forall y \in S(v)$:
9c. $h' \leftarrow h \cup \{\bar{y}\}$
9d mark h'
9e. $sensitive(\bar{y}) \leftarrow v_{y \leftarrow 0}$
9f. $pos(h') \leftarrow pos(h)$
9g. $H \leftarrow H \cup \{\langle h', pos(h')\rangle\}$
10. $\forall x \in (lits(H) - A)$, grow-term($H, sensitive(x), x$)
11. GOTO 3.

grow-term(H, a, x)

1. $A \leftarrow A \cup \{x\}$ /* literal x witnesses progress */
2. $h \leftarrow$ find-sub-term(a, x)
3. $pos(h) \leftarrow a$
4. $H \leftarrow H \cup \{\langle h, pos(h)\rangle\}$
5. IF $h = S(a)$ and $\exists\ u \notin (S(a_{x \leftarrow 0}) \cup S(a))$ such that
   $S(a_{u \leftarrow 0}) \supset S(a)$
   THEN /* a satisfies 2 terms, but $a_{u \leftarrow 0}$ only one */
5a. $a' \leftarrow a_{u \leftarrow 0}$
5b. $h' \leftarrow S(a')$
5c. $\forall y \in S(a') : sensitive(y) \leftarrow a'$
5d. $pos(h') \leftarrow a$
5e. $H \leftarrow H \cup \{\langle h', pos(h')\rangle\}$

find-sub-term(a, x)

1. $Q \leftarrow \{a\}$
   $Y \leftarrow (S(a_{x \leftarrow 0}) - \{\bar{x}\})$
2. LOOP: SEARCH for $\langle w, y, u\rangle$ such that:
   /* search for $w_{u \leftarrow 0}$, a valid test for y w.r.t. x */
   $w \in Q$,
   $y \in S(w_{x \leftarrow 0}) \cap Y$, and
   $u \in S(w_{x \leftarrow 0, y \leftarrow 0}) - S(w_{x \leftarrow 0})) - \{\bar{y}\} - \cup_{v \in Q} S(v)$
3 IF no such $\langle w, y, u\rangle$ is found THEN GOTO line 7
4. ELSE $Y \leftarrow Y - \{y\}$.
5. $Q \leftarrow Q \cup \{w_{u \leftarrow 0}\}$
6. CONTINUE
7. $h \leftarrow \cup\{S(v) : v \in Q\}$
8. $\forall y \in h$: $sensitive(y) \leftarrow$ some $v \in Q$ such that $y \in S(v)$
9. RETURN h

fsa-positive-counterexample(H, p, n)

1. $L1 \leftarrow \{x : x \in lits(H), \bar{x} \notin lits(H), p(x) = 1, n(x) = 0\}$
2. $L2 \leftarrow \{x : x \in lits(H), \bar{x} \in lits(H),$ and $p(x) \neq n(x)\}$
3. $L3 \leftarrow \{x : x \notin (L1 \cup L2),$ and $p(x) \neq n(x)\}$
4. $w \leftarrow p$
5. FOR $x \in L1$
6. IF $f(w_{x \leftarrow 0}) = 0$ THEN RETURN (w, x)
   ELSE $w \leftarrow w_{x \leftarrow 0}$
7. FOR $x \in L2 : w \leftarrow w_{x \leftarrow 0}$
8. FOR $x \in L3$:
   IF $f(w_{x \leftarrow 0}) = 0$ THEN RETURN (w, x)
   ELSE $w \leftarrow w_{x \leftarrow 0}$

fsa-negative-counterexample(H, p, n)

1. $L1 \leftarrow \{x : x \in lits(H), \bar{x} \notin lits(H), p(x) = 1, n(x) = 0\}$
2. $L2 \leftarrow \{x : x \in lits(H), \bar{x} \in lits(H),$ and $p(x) \neq n(x)\}$
3. $L3 \leftarrow \{x : x \notin (L1 \cup L2),$ and $p(x) \neq n(x)\}$
4. $w \leftarrow p$
5. FOR $x \in (L1 \cup L2) : w \leftarrow w_{x \leftarrow 0}$
6. FOR $x \in L3$:
   IF $f(w_{x \leftarrow 0}) = 0$ THEN RETURN (w, x)
   ELSE $w \leftarrow w_{x \leftarrow 0}$
7. RETURN (w, x)

always the case that $S(a)$ is a subset of every term satisfied by a. (Note that even if a is a sensitive assignment, there may be two terms that it satisfies.) Thus when given a positive sensitive assignment a, a key problem is to determine how to extend $S(a)$ to include more of the literals of the term that a satisfies. The subroutine find-sub-term (which is called by grow-term) uses a polynomial number of membership queries and extends $S(a)$ to approximate a term t satisfied by a. We give a brief description of how grow-term and find-sub-term work.

Let x be a literal in $S(a)$. If x is unate then find-sub-term finds the entire unate portion of a term satisfied by a, whereas if x is biform find-sub-term finds an entire term satisfied by a except for perhaps one literal. We consider separately the cases for x unate and x biform. It is easily argued that when x is unate a might satisfy one or two terms, but when x is biform a satisfies exactly one term.

The case where x is unate and a satisfies just one term is fairly straightforward: $S(a)$ includes all of the unate literals in t and $S(a)$ is included in the new term h constructed by find-sub-term. It is possible to show that when a satisfies two terms and $S(a)$ is not already the entire unate portion of one of them, then there must be some literal u such that $S(a_{u\leftarrow 0})$ includes the unate portion of one of the terms. The code of line 5 of grow-term finds such a literal u, and thus adds the entire unate portion of at least one of the terms that is satisfied by a.

If x is biform, then a satisfies exactly one term t_x. Consider why a literal y might be in t_x but not in $S(a)$. (By Property 5, $y \in (S(a_{x\leftarrow 0}) - \{\bar{x}\})$.) This can only happen if a term $t_{\bar{y}}$ containing the negation of y becomes satisfied when we flip the value of literal y in assignment a. In other words, $t_{\bar{y}}$ is almost satisfied by assignment a with respect to $\bar{y}$ (Definition 1). Suppose that we can find a literal z also in $t_{\bar{y}}$ (but different from $\bar{y}$) to flip so that $a_{z\leftarrow 0}$ does not almost satisfy $t_{\bar{y}}$, and such that $a_{z\leftarrow 0}$ is still a positive sensitive assignment for x. Then when y is flipped in $a_{z\leftarrow 0}$ to obtain $a_{y\leftarrow 0, z\leftarrow 0}$, the value of f changes from 1 to 0 if y is in the same term as x, because (1) t_x is no longer satisfied, and (2) $t_{\bar{y}}$ does not become satisfied. In fact, y is in the same term as x if and only if $y \in S(a_{z\leftarrow 0})$. We call an assignment such as $a_{z\leftarrow 0}$ a *valid test* for y with respect to x. Subroutine find-sub-term attempts to find literals y and z such that $a_{z\leftarrow 0}$ is a valid test for y with respect to x.

One problem with this approach is that although $y \in t_x$, every $z \in t_{\bar{y}}$ that might be flipped is such that $a_{y\leftarrow 0, z\leftarrow 0}$ satisfies another term $t_{\bar{z}}$. However, it is possible to argue that for *some* y and z this will not occur. Further, after one such valid test a' is found, additional valid tests may be found by flipping literals of a' as well as of a. By a type of chaining, the algorithm thus finds sequences of literals to flip to construct valid tests for all but at most one literal of t_x. The argument that this is possible is involved, and relies on a graph of the terms in the formula to show that the routine find-sub-term performs as suggested. The actual code in line 3 of find-sub-term that performs this search appears unrelated to the description contained here; additional technical lemmas show that what is searched for in the code is equivalent to the valid tests we have described.

Two additional problems arise due to the fact that only a *partial term* is found from each positive sensitive assignment obtained or generated by the algorithm.

First, given a new positive counterexample, the simple walking procedure that was described in the read-once case does not work; in order to obtain a positive sensitive assignment that satisfies some term not yet included (or partially included) in H, it is not sufficient to flip single literals not in H. The subroutine fsa-positive-counterexample ("fsa" abbreviates "find sensitive assignment") instead sometimes "walks", and sometimes "leaps" by flipping whole groups of literals at once. A subtle argument shows that it is possible to do this and still guarantee that a sensitive assignment is found. This technique is one which may be applicable in other learning problems where one must isolate the influences of multiply occurring variables. In the read-once case, the "walking" can be proved correct because whenever a literal appears in H, in fact the entire term containing the literal is in H. Here, in order to to prove that the "leaping" of fsa-positive-counterexample performs correctly, we require a weaker closure property: H must contain a partial term from f satisfying certain requirements for every literal in H. To ensure that this is the case, the last line of the main algorithm runs grow-term to find a partial term containing each literal x which has not yet had a partial term found for it by grow-term. This closure property is also needed to prove the correctness of the analogous subroutine fsa-negative-counterexample discussed below.

The second problem arising from including only partial terms in H is that of extending these terms to complete terms: How are the missing biform literals found for those partial terms of H containing only the unate portions of terms of f, and how is the missing literal found for those terms of H missing only one

literal? Both of these problems are handled by using negative counterexamples.

First consider the case where the equivalence query returns a negative counterexample n for a term h containing only the unate portion of a term t in f. We would like to argue that in this case the routine fsa-negative-counterexample walks and leaps from a positive example satisfying t (such an assignment, called $pos(h)$, was saved when h was added to H) towards n, and finds a positive sensitive assignment for one of the missing biform literals in $t - h$ (at which point grow-term and find-sub-term will extend the partial term to contain all but at most one literal, which is dealt with in the next case). However, only a weaker, yet still sufficient guarantee can be made. If a positive sensitive assignment is not found that allows the algorithm to extend the partial term h, then it can be argued that a positive sensitive assignment is found that allows grow-term and find-sub-term to either (a) extend some other partial term, or (b) add a portion h of some term t of f such that H contains no other partial term for t, or (c) add to H a partial term containing some new literal of f not yet in H. Each of these cases makes "progress" at moving the hypothesis closer to being logically equivalent to the target formula, and each can occur at most a polynomial number of times.

Next, consider the case when the equivalence query returns a negative counterexample n for a term missing only one literal. Here it can be argued that if fsa-negative-counterexample returns a positive sensitive assignment, then progress is made by grow-term and find-sub-term as described in the cases (a), (b), (c), above, or else no positive sensitive assignment is found, in which case the missing biform literal is guaranteed to be added in line 9 of learn-read-twice-dnf, where all possible additional literals (subject to certain technical constraints) are added onto the partial term h. It is necessary that we mark the terms which are added at this step, so that in subsequent iterations of the main loop, counterexamples to the incorrect single-literal extensions of h will not cause the addition of even more terms to H, etc., resulting in exponential growth.

4 Skeleton of Proof

We state the definitions, properties and some of the lemmas that are used to support Theorem 1. Due to space limitations, most proofs are omitted from this abstract, as are the statements of a number of supporting sub-lemmas. The proof of an interesting and central technical lemma is included in the appendix.

Earlier we described that grow-term makes progress by either extending a portion of a term in H, or by adding a portion of a new term not yet in H. The next definition describes the relevant portions of terms. Essentially, *complete* contains those terms of f that the current hypothesis H has already found, *missing_one* those terms of f that are approximated by a term in H that is missing at most one literal, and *all_unate* those terms of f for which H includes at least the entire unate portion.

Definition 3 *Below, $pos(h)$ is always assigned a positive sensitive assignment that satisfies some $t \supseteq h$. We define the following subsets of terms of f.*

complete $= f \cap H$.

missing_one $= \{t : t \in f$, *and for some literal* $z \in t$ *and unmarked* $h \in H$, $t = h \cup \{z\}$ *and* $t(pos(h)) = 1\}$ $\cup$ *complete*.

all_unate $= \{t : t \in f, \exists h \in H$ *such that* $unate(t) \subseteq h \subseteq t$ *and* $t(pos(h)) = 1\} \cup$ *missing_one*.

Note complete $\subseteq$ *missing_one* $\subseteq$ *all_unate.*

The arguments that find-sub-term and grow-term work correctly involve the following definition of "term graph" and its properties.

Definition 4 *If a is a positive sensitive assignment for the biform literal x then the* term graph $G(a, x)$ *is the directed graph (V, E) with labeled vertex set V and labeled edge set E such that*

- $V = \{t : \exists y \in S(a_{x \leftarrow 0})$ *such that t is almost satisfied by assignment $a_{x \leftarrow 0}$ with respect to the literal $\bar{y}$}$\}$
- *Every term t in the graph is labeled with the (unique) literal $\bar{y}$ such that t is almost satisfied by $a_{x \leftarrow 0}$ with respect to $\bar{y}$. (The label is unique because by the definition of almost satisfied, there can be at most one such literal $\bar{y}$.)*
- $E = \{\langle r, s\rangle : \exists u \in r, \bar{u} \in s$, *and* $a_{x \leftarrow 0}(u) = 1\}$
- *Every edge, $\langle r, s\rangle$ is labeled with the (unique) literal u such that $u \in r, \bar{u} \in s$, and $a_{x \leftarrow 0}(u) = 1$. (The label of an edge $\langle r, s\rangle$ is unique because if u is such a label, then $\bar{u} \in s$ and $a_{x \leftarrow 0}(u) = 1$, and $\bar{u}$ must then be the unique label of s.)*

The label $\bar{y}$ of a term (i.e., a vertex) in $G(a, x)$ is a literal in that term, so if $\bar{y}$ is biform then the term labeled $\bar{y}$ is $t_{\bar{y}}$. If y is the label of an edge from term r

to term s, then necessarily y is biform, with $r = t_y$ and $s = t_{\bar{y}}$. If a biform literal $\bar{y}$ is the label of a vertex, then there is only one vertex with label $\bar{y}$, since a second vertex with label $\bar{y}$ would imply that $\bar{y}$ appears twice, and y once, contradicting the assumption that f is a read-twice DNF.

Property 8 *For every literal $y \in S(a_{x \leftarrow 0})$, there is a term in $G(a,x)$ labeled $\bar{y}$. Since $\bar{x} \in S(a_{x \leftarrow 0})$, t_x is in $G(a,x)$ with label x.*

Property 9 *$\forall t \in G(a,x)$, indegree$(t) \leq 1$.*

Property 10 *Define the directed graph $G^*(a,x) = (V, E^*)$, where $E^* = E - \{e : e$ is an edge leading to $t_x\}$. Then the subgraph of $G^*(a,x)$ reachable from t_x is a tree rooted at t_x.*

The next two definitions are used in the proof of the correctness of find-sub-term.

Definition 5 *If a is a positive sensitive assignment for x then a node t (labeled $\bar{y}$) in V is* nice *iff $x \notin t$, y is biform, and $\exists z \in t - \{\bar{x}, \bar{y}\} - t_x$ such that there is not an edge labeled z in $G^*(a,x)$ emanating from t.*

Definition 6 *If a is a positive sensitive assignment for x then the literal y is in testable(a,x) iff y is biform, $y \in (S(a_{x \leftarrow 0}) - \{\bar{x}\})$, and $t_{\bar{y}}$ is in $G^*(a,x)$ and has a path to a nice node. Also define nontestable$(a,x) = (S(a_{x \leftarrow 0}) - \{\bar{x}\}) -$ testable(a,x).*

Correctness of fsa-positive-counterexample and fsa-negative-counterexample If the equivalence query in the main loop of learn-read-twice-dnf returns a counterexample then either fsa-positive-counterexample or fsa-negative-counterexample will be called. Lemmas 2 and 3 show that under the appropriate conditions these subroutines will find a "useful" positive sensitive assignment — one for a literal not yet in *lits*(H) or for a term not yet in *all_unate*; such an assignment is useful in that it enables grow-term to make progress on H by adding a new term to *all_unate*, *missing_one*, or *complete*. The proofs of Lemmas 2 and 3 require that H be "closed", hence Definition 7 and Lemma 1.

Definition 7 *H is closed (with respect to the target formula f) iff lits$(H) \subseteq$ lits(f) and $\forall x \in$ lits(H), if x is unate then $\exists t \in f$ such that $x \in t$ and $t \in$ all_unate, and if x is biform then $t_x \in$ missing_one.*

Lemma 1 *Whenever fsa-positive-counterexample or fsa-negative-counterexample is called by learn-read-twice-dnf, the hypothesis H is closed.*

Lemma 2 *If p is a positive counterexample of H, n is a negative example of f, and H is closed, then fsa-positive-counterexample(H,p,n) returns (w,x) such that w is a positive sensitive assignment for x, and either $x \notin$ lits(H) or $\forall t \in$ all_unate, $t(w) = 0$.*

Lemma 3 *If p is a positive example of f that satisfies $t \in$ (all_unate $-$ missing_one), n is a negative counterexample of H that satisfies unate(t), and H is closed, then fsa-negative-counterexample(H,p,n) returns (w,x) such that w is a positive sensitive assignment for x and $x \notin$ lits(H).*

Correctness of find-sub-term To show that grow-term makes progress on H when given an appropriate positive sensitive assignment and literal pair (a,x) we first argue about what find-sub-term finds (Lemmas 4, 5, and 6). The proofs of these lemmas rely on two technical lemmas (Lemmas 7 and 8) relating the operation of finding a sensitive set to the literals appearing in a term of the target formula. The second containment in each of Lemmas 4, 5, and 6 follows from a sub-lemma stating that all of the assignments added to Q in find-sub-term are positive sensitive assignments for x. The first containment in Lemma 4 is easy to show since find-sub-term includes $S(a)$ in the partial term h that it constructs, and by Property 4, this contains the unate portion of the term satisfied by a. The first containments of parts 2 and 3 of Lemma 5 are more involved, but rely on the same, and similar, properties.

Lemma 4 *If a is a positive sensitive assignment for unate literal x such that a only satisfies one term t, then unate$(t) \subseteq$ find-sub-term$(a,x) \subseteq t$.*

Lemma 5 *If a is a positive sensitive assignment for unate x and a satisfies terms r and s, then either:*

1. *find-sub-term$(a,x) = r \cap s$,*
2. *unate$(r) \subseteq$ find-sub-term$(a,x) \subseteq r$, or*
3. *unate$(s) \subseteq$ find-sub-term$(a,x) \subseteq s$.*

The proof of the first containment of the next lemma has the following flavor. Find-sub-term includes all of $S(a)$, thus by Property 5, the only literals that might be missing from t_x are in $S(a_{x \leftarrow 0}) - \{\bar{x}\}$. Each literal $y \in (S(a_{x \leftarrow 0}) - \{\bar{x}\})$ corresponds to a node in $G^*(a,x)$. The technical lemmas below (7 and 8) are used to show that a valid test is found for all literals in $(S(a_{x \leftarrow 0}) - \{\bar{x}\})$ whose corresponding node

has a path to a nice node. It can be shown that these correspond to testable literals, thus the only literals missing from the h that is returned by find-sub-term are those in $t_x \cap$ *nontestable*.

Lemma 6 *If a is a positive sensitive assignment for biform literal x, then $(t_x - \mathit{nontestable}(a,x)) \subseteq \mathit{find\text{-}sub\text{-}term}(a,x) \subseteq t_x$.*

Lemma 7 *If x is biform and a is a valid test for y with respect to x, then $y \in t_x \Leftrightarrow y \in S(a)$.*

Lemma 8 *If a is a positive sensitive assignment for biform literal x, and y is biform and in $S(a_{x\leftarrow 0})$, then $S(a_{x\leftarrow 0,y\leftarrow 0}) - S(a_{x\leftarrow 0}) = t_{\bar{y}} - S(a_{x\leftarrow 0})$.*

Correctness of grow-term To see that when given a useful assignment a and literal x grow-term works as desired to improve H, we first establish in Lemma 9 that grow-term adds to H a certain portion of a term satisfied by a. We then argue in the proofs of Lemmas 10 and 11 that these portions are sufficient to guarantee that x will be added to *lits*(H) or that a term will be added to *all_unate*, *missing_one*, or *complete*.

Conditions 1 and 3 of Lemma 9 are easily proven since in these cases find-sub-term has found a sufficient portion of $t(a)$ (Lemmas 4 and 6 above). In the case where a satisfies 2 terms it can be shown that line 5 of grow-term boosts Lemma 5 to condition 2 of Lemma 9.

Lemma 9 *If a is a positive sensitive assignment for x, then after grow-term(H,a,x) returns, H contains an unmarked pair $\langle h, \mathit{pos}(h)\rangle$ such that:*

1. *If x is unate and a satisfies only one term $t \in f$, then $\mathit{unate}(t) \subseteq h \subseteq t$ and $t(\mathit{pos}(h)) = 1$;*
2. *If x is unate and a satisfies two terms $\{r,s\} \subseteq f$, then either $\mathit{unate}(r) \subseteq h \subseteq r$ and $r(\mathit{pos}(h)) = 1$, or $\mathit{unate}(s) \subseteq h \subseteq s$ and $s(\mathit{pos}(h)) = 1$;*
3. *If x is biform, then $t_x - \mathit{nontestable}(a,x) \subseteq h \subseteq t_x$ and $t_x(\mathit{pos}(h)) = 1$.*

The difficult part in proving Lemma 10 from Lemma 9 is showing that when condition 3 of Lemma 9 holds, the term h found is missing at most one literal, thus t_x will be added to *missing_one*. To do this, we show that $|t_x \cap \mathit{nontestable}| \leq 1$; the proof is included in the appendix.

Lemma 10 *If a is a positive sensitive assignment for x and $\forall t \in \mathit{all\_unate}, t(a) = 0$, then grow-term$(H,a,x)$ adds a term to all_unate, missing_one, or complete (and no term is ever removed from all_unate, missing_one, or complete).*

Lemma 11 *If a is a positive sensitive assignment for x and $x \notin \mathit{lits}(H)$, then grow-term$(H,a,x)$ adds x to lits(H) (and x is never removed from lits(H)).*

Proof of Theorem 1 Since the algorithm continues running until an equivalent DNF is hypothesized (lines 3,4), if it halts it is correct. We bound the number of iterations of the main loop. In every iteration a counterexample is found, and the algorithm either

1. finds a positive sensitive assignment and literal (a,x) (from line 6a or 7b) such that by Lemmas 2 and 3 either (a) $x \notin \mathit{lits}(H)$, or (b) $\forall t \in \mathit{all\_unate}, t(a) = 0$, or
2. removes a term h from H and if h is unmarked adds at most n marked terms to h.

When 1(a) occurs, by Lemma 11, a new literal is added to H. Since H contains at most $2n$ literals, this can occur at most $2n$ times. By Lemma 10, each time 1(b) occurs a term is added to *all_unate*, *missing_one*, or *complete*. Since each of these sets is bounded by the number of terms of f, which is at most $2n$ since f is read-twice, 1(b) can occur at most $6n$ times. Thus condition 1 can occur at most $8n$ times.

Next we show that condition 2 can occur at most $16n + 8n^2$ times. Consider all the ways that terms can be added to H:

- Each time condition 1 occurs, grow-term is called, and at most two unmarked terms are added to H. Thus a total of $2 \cdot 8n = 16n$ unmarked terms can be added to H.
- For every unmarked term, n marked terms can be added by condition 2. Thus a total of $n \cdot 16n = 16n^2$ marked terms can be added.

Therefore at most $16n + 16n^2$ terms (either marked or unmarked) can be added to H, so at most this many can be removed. Condition 2 can occur at most as many times, hence the total number of iterations of the main loop is at most $8n + (16n + 16n^2) = 24n + 16n^2$. This also bounds the number of equivalence queries

A straightforward analysis shows that the search in line 2 of find-sub-term, which dominates the complexity of the entire algorithm, accounts for $O(n^6)$ steps on a unit cost RAM with word size $O(\log n)$. This is also a bound on the number of membership queries. □

Appendix

Lemma 12 *If a is a positive sensitive assignment for biform literal x then $|t_x \cap nontestable(a,x)| \leq 1$.*

It can be argued that every *nontestable* literal $y \in t_x$ corresponds to a child $t_{\bar{y}}$ of t_x in $G^*(a,x)$ such that $t_{\bar{y}}$ does not have a path to a nice node. Since the subgraph of $G^*(a,x)$ reachable from t_x is a tree rooted at t_x, $t_{\bar{y}}$ is the root of a subtree (denoted by $tree(t_{\bar{y}})$). To prove that at most one child of $t_{\bar{x}}$ does not have a path to a nice node, we assume to the contrary that at least two children of t_x, $t_{\bar{y}}$ and $t_{\bar{z}}$, do not have a path to a nice node; thus there are no nice nodes in $tree(t_{\bar{y}})$ or in $tree(t_{\bar{z}})$. However at least one of these trees must not contain $t_{\bar{x}}$ (without loss of generality assume $t_{\bar{x}} \notin tree(t_{\bar{z}})$). Lemma 13 now implies that t_x is not a minterm of f, thus contradicting our assumption that f is a reduced DNF formula.

Lemma 13 *If a is a positive sensitive assignment for biform literal x, $y \in t_x - \{x\}$, $t_{\bar{y}} \in G^*(a,x)$, $tree(t_{\bar{y}})$ is the subtree of $G^*(a,x)$ rooted at $t_{\bar{y}}$, $t_{\bar{x}} \notin tree(t_{\bar{y}})$, and there are no nice nodes in $tree(t_{\bar{y}})$, then t_x is not a minterm of f.*

Proof: We first describe the consensus operation. Let r and s be two terms, not necessarily from f. If there is exactly one literal x in r such that $\bar{x}$ is in s, then the consensus of r and s (denoted by $r \odot s$) is defined to be the term $r \cup s - \{x, \bar{x}\}$. When there is not exactly one literal x in r such that $\bar{x}$ is in s, then $r \odot s$ is undefined. It is easily shown that if r and s are both implicants of a function f and $r \odot s$ is defined, then $r \odot s$ is also an implicant of f.

To show that t_x is not a minterm of f we show that there exists another implicant of f which contains a proper subset of the literals in t_x. This is done by first finding an implicant t of f such that t contains exactly one literal whose negation is in t_x. Furthermore, all other literals of t are also in t_x; thus $t_x \odot t$ derives a new implicant of f which is a proper subset of t_x. In particular, we will show that by iteratively taking consensus of the terms in $tree(t_{\bar{y}})$ it is possible to derive an implicant t of f such that

$$\{\bar{y}\} \subseteq t \subseteq (t_x - \{y\}) \cup \{\bar{y}\}. \qquad (1)$$

Since $y \in t_x$ by the hypothesis of this lemma, $t_x \odot t$ is defined and is a proper subset of t_x as desired.

We derive t by iteratively consenting the terms from $tree(t_{\bar{y}})$ onto an "accumulated term". After consenting all the terms from $tree(t_{\bar{y}})$, this accumulated term will be t. In the formal statement of this consenting procedure below, T^i_j refers to the accumulated term. Since we consent the terms from $tree(t_{\bar{y}})$ in a breadth first fashion, it makes sense to refer to the different accumulated terms with respect to how far down the breadth first traversal of $tree(t_{\bar{y}})$ they have consented; thus T^i_j refers to the accumulated term after consenting terms through the jth term on the ith level of $t_{\bar{y}}$.

We now describe more formally the definition of iterated consensus of terms in $tree(t_{\bar{y}})$ to find t. Below, let t^i_j be the jth term on the ith level of $tree(t_y)$, k_i be the number of terms on the ith level of $tree(t_{\bar{y}})$, and d be the depth of $t_{\bar{y}}$. The terms T^i_j, and the final term t constructed, are defined recursively as follows:

$T^1_1 = t_{\bar{y}}$ (the root of the tree)
$T^i_j = T^i_{j-1} \odot t^i_j$ for $1 \leq i \leq d$ and $2 \leq j \leq k_i$
$T^{i+1}_1 = T^i_{k_i} \odot t^{i+1}_1$
$t = T^d_{k_d}$

The breadth first procedure for finding t is implied by the above recursive definition of T^i_j. The breadth first traversal of $tree(t_{\bar{y}})$ terminates with the last node on the deepest level, $t^d_{k_d}$, which will be the last term consented to derive $T^d_{k_d} = t$.

Recall that $y \in t_x$, thus since $\bar{y} \in t_{\bar{y}}$, y must be biform. Further, since the indegree of $t_x = 0$, it is clear that t_x is not in $tree(t_{\bar{y}})$; so y does not appear in $tree(t_{\bar{y}})$. By the definition of consensus, for all terms r and s, $r \odot s \subset r \cup s$. Thus, since y does not occur in $tree(t_{\bar{y}})$, y will not be in t, so to show equation (1) we only need to show that $\{\bar{y}\} \subseteq t \subseteq t_x \cup \{\bar{y}\}$. To see that $\{\bar{y}\} \subseteq t$, note that $\bar{y}$ is in the root of $tree(t_{\bar{y}})$ and y does not occur in $tree(t_{\bar{y}})$, so $\bar{y}$ will never be removed during any of the consensus operations in the derivation of t.

We complete the proof of the lemma by showing that $t \subseteq t_x \cup \{\bar{y}\}$. Let $E(i, i+1)$ denote the set of labels for directed edges in $tree(t_{\bar{y}})$ from level i to level $i+1$. We prove by induction on i that all the consensus operations performed to derive $T^i_{k_i}$ are valid and that $E(i,i+1) \subseteq T^i_{k_i} \subseteq t_x \cup \{\bar{y}\} \cup E(i,i+1)$. Since there are no outgoing edges on the last level of $tree(t_{\bar{y}})$, we have that $t = T^d_{k_d} \subseteq t_x \cup \{\bar{y}\}$, as desired.

Base case: Since $t_{\bar{y}}$ is the only term on the first level of $t_{\bar{y}}$, $T^1_{k_1} = T^1_1 = t_{\bar{y}}$ and $E(1,2) =$ the labels of the edges emanating from $t_{\bar{y}}$. Since no consensus operations were performed to derive $t_{\bar{y}}$ it is trivially true that all the consensus operations performed so far are valid. Recall from the definition of edge labels that if l is the label of an edge from r to s, then the literal l must be in r; Thus, $E(1,2) \subseteq t_{\bar{y}}$. An unstated technical lemma shows that $t_{\bar{y}} \subseteq t_x \cup \{\bar{y}\} \cup E(1,2)$.

Inductive step: Assume inductively that the consensus operations in the definition of $T^{i-1}_{k_{i-1}}$ are all valid, and that $E(i-1,i) \subseteq T^{i-1}_{k_{i-1}} \subseteq t_x \cup \{\bar{y}\} \cup E(i-1,i)$. Let $t_{\bar{z}_1}, t_{\bar{z}_2}, \ldots, t_{\bar{z}_k}$ be the set of terms on the ith level of $tree(t_{\bar{y}})$ with labels $\bar{z}_1, \bar{z}_2, \ldots, \bar{z}_k$ respectively. The edges leading to these terms have labels $z_1, z_2, \ldots, z_k$ respectively. Without loss of generality, assume that the order of the terms on the ith level of the tree is such that $T^i_{k_i} = (((T^{i-1}_{k_{i-1}} \odot t_{\bar{z}_1}) \odot t_{\bar{z}_2}) \ldots \odot t_{\bar{z}_k})$. To finish the induction we show that all the consensus operations in this expression are valid and that $E(i,i+1) \subseteq T^i_{k_i} \subseteq t_x \cup \{\bar{y}\} \cup E(i,i+1)$.

For notational convenience let $T^i_0 = T^{i-1}_{k_{i-1}}$. To show that the jth consensus operation at the ith level $(T^i_{j-1} \odot t_{\bar{z}_j})$ is valid for $1 \leq j \leq k$ we must show that there is exactly one literal z in T^i_{j-1} such that $\bar{z}$ is in $t_{\bar{z}_j}$. First we show that there is at least one such literal (namely $\bar{z}_j$, the label of the edge leading into $t_{\bar{z}_j}$). Since $E(i-1,i) \subseteq T^{i-1}_{k_{i-1}}$, $T^{i-1}_{k_{i-1}}$ must contain z_j. Further, z_j must be biform, thus $\bar{z}_j$ occurs in only one term on the ith level of $tree(t_{\bar{y}})$. Since $\bar{z}_j$ occurs in $t_{\bar{z}_j}$, no other term on level i that is consented onto $T^{i-1}_{k_{i-1}}$ contains $\bar{z}_j$; thus z_j cannot be removed by consensus on level i prior to consensus with $t_{\bar{z}_j}$.

Next we show that there is at most one literal z in T^i_{j-1} such that $\bar{z}$ is in $t_{\bar{z}_j}$. Assume to the contrary that there are two literals u and v in T^i_{j-1} such that $\bar{u}$ and $\bar{v}$ are in $t_{\bar{z}_j}$. Therefore u and v are biform and the terms containing u and v (i.e., t_u and t_v) were involved in the iterated consensus that derived T^i_{j-1}. Thus t_u and t_v are either on the same level or above $t_{\bar{z}_j}$ in $tree(t_{\bar{y}})$. Note by the definition of E, since $u \in t_u$ and $\bar{u} \in t_{\bar{z}_j}$, that there is either an edge from t_u to $t_{\bar{z}_j}$ or from $t_{\bar{z}_j}$ to t_u. Similarly, there is either an edge from t_v to $t_{\bar{z}_j}$ or from $t_{\bar{z}_j}$ to t_v. But since there are no edges between nodes on the same level of the tree, t_u and t_v must both be above $t_{\bar{z}_j}$ in $tree(t_{\bar{y}})$, and since all edges in the tree lead down, the two edges must be from t_u to $t_{\bar{z}_j}$ and from t_v to $t_{\bar{z}_j}$, with labels u and v, respectively. This contradicts the earlier observation that the indegree of every node in $G^*(a,x)$ is at most 1. Therefore all of the consensus operations in the definition of $T^i_{k_i}$ are valid, and z_j is the unique literal in T^i_{j-1} whose negation is in $t_{\bar{z}_j}$.

By induction, we have $E(i-1,i) \subseteq T^{i-1}_{k_{i-1}} \subseteq t_x \cup \{\bar{y}\} \cup E(i-1,i)$. We show that in deriving $T^i_{k_i} = (((T^{i-1}_{k_{i-1}} \odot t_{\bar{z}_1}) \odot t_{\bar{z}_2}) \ldots \odot t_{\bar{z}_k})$, (1) all literals of $E(i-1,i)$ are eliminated, (2) all literals of $E(i,i+1)$ are added, and (3) any other literal added must be in $t_x - \{y\}$. Thus, after the ith level consensus, we will have $E(i,i+1) \subseteq T^i_{k_i} \subseteq t_x \cup \{\bar{y}\} \cup E(i,i+1)$, completing the inductive step.

To see that (1) holds, note that by the definition of $T^i_{k_i}$, all of the terms from level i are consented onto $T^{i-1}_{k_{i-1}}$, so every label z_j in $E(i-1,i)$ will be removed from $T^{i-1}_{k_{i-1}}$ by consenting the term $t_{\bar{z}_j}$.

To see that (2) holds, let z be the label of an edge from level i to $i+1$. Then, by definition, z appears in some term at level i, and $\bar{z}$ in some term at level $i+1$. But each term on the ith level, in particular t_z, must be consented onto $T^{i-1}_{k_{i-1}}$ to derive $T^i_{k_i}$; thus z will appear in $T^i_{k_i}$. (Note that z cannot be removed from $T^i_{k_i}$ until consensus is done with the unique occurrence of $\bar{z}$ in $t_{\bar{z}}$ at the $(i+1)$st level of the tree.) (3) follows from an unstated technical lemma. □

References

[1] D. Angluin. Queries and concept learning. *Machine Learning*, 2:319–342, 1988.

[2] D. Angluin, M. Frazier, and L. Pitt. Learning conjunctions of Horn clauses. In *Proceedings of the 31st Annual IEEE Symposium on Foundations of Computer Science*, pages 186–192, IEEE Computer Society Press, Washington, D.C., October 1990.

[3] D. Angluin, L. Hellerstein, and M. Karpinski. *Learning Read-Once Formulas with Queries.* Technical Report, University of California at Berkeley, Report No. 89/528, 1989. (Also, International Computer Science Institute Technical Report TR-89-050.).

[4] D. Angluin and M. Kharitonov. When won't membership queries help? In *Proceedings of the 23rd Annual ACM Symposium on Theory of Computing*, ACM, New York, May 1991.

[5] D. Angluin and L. Pitt. Private communication. 1990.

[6] T. Hancock. Learning 2μdnf formulas and $k\mu$ decision trees. In *Proceedings of the 1991 Workshop on Computational Learning Theory*, Morgan Kaufmann, San Mateo, CA, August 1991.

[7] L. Kucera, A. Marchetti-Spaccamela, and M. Protasi. On the learnability of DNF formulae. In *Proceedings of the 15th International Colloquium on Automata, Languages, and Programming*, Springer-Verlag, Heidelberg, July 1988.

[8] N. Linial, Y. Mansour, and N. Nisan. Constant depth circuits, Fourier transform, and learnability. In *Proceedings of the 30th Annual IEEE Symposium on Foundations of Computer Science*, pages 574–579, IEEE Computer Society Press, Washington, D.C., October 1989.

[9] L. G. Valiant. Learning disjunctions of conjunctions. In *Proceedings of the 9th International Joint Conference on Artificial Intelligence, Vol. 1*, pages 560–566, Los Angeles, California, August 1985.

[10] L. G. Valiant. A theory of the learnable. *Communications of the ACM*, 27(11):1134–1142, 1984.

[11] K. Verbeurgt. Learning DNF under the uniform distribution in quasi-polynomial time. In *Proceedings of the 1990 Workshop on Computational Learning Theory*, pages 314–326, Morgan Kaufmann, San Mateo, CA, August 1990.