

Exact Learning Boolean Functions via the Monotone Theory

By: Nader Bshouty

Published in INFORMATION AND COMPUTATION
Vol. 123, No. 1, November 15, 1995

Presented at the Seminar on Computational
Learning Theory

by Genady Beryozkin, genadyb@cs

Technion, 29 March, 2000

1 Introduction

This is a summary of a presentation of a paper “Exact Learning Boolean Functions via the Monotone Theory” by Nader Bshouty in the Computation Learning Seminar, held at the Technion.

The monotone theory describes a way to learn any boolean function $f : \{0, 1\}^n \rightarrow \{0, 1\}$ in a time polynomial in the size of its minimal DNF representation, the size of its minimal CNF representation and the number of variables, n . From the above follows that decision trees can be learned in a time polynomial in the number of leaves and the number of variables, a question that has been open until then.

The main idea of the monotone theory is to find many monotone approximations (with respect to some partial order on the assignments) to the target function f . These approximations are later used to find f .

All the results presented in this summary are in the model of exact learning with equivalence and membership queries. Being able to learn a class of functions in this model implies the ability to learn that class in both the PAC and the PAC_D models (with membership queries).

The learning models and the query types were presented in the class in previous lectures and are not described here. However, the algorithms for learning monotone functions and simple monotone approximations appear in this summary as a background, and to provide an intuition and simplify many proofs of the monotone theory.

2 Learning monotone DNF functions

The subclass of monotone DNF functions (MDNF) is a class of functions that have a DNF representation with no negative literals. This section describes the algorithm for learning the MDNF class, and analyzes its complexity.

2.1 MDNF functions

First we define a partial order over the set of assignments, $\{0, 1\}^n$:

Definition 2.1 For two assignments a, b we say $a \leq b$ if for every i , we have $a_i \leq b_i$, where a_i is the value of the i -th variable in the assignment a .

The partial order we have just introduced, defines a lattice over the set $\{0, 1\}^n$.

Definition 2.2 $f(x)$ is monotone if for every $a \leq b$ we have $f(a) \leq f(b)$.

It is clear that for any DNF formula with no negative literals, the above definition holds, and we will see below that a monotone function has a DNF representation that does not have any negative literals.

Definition 2.2 implies that there exists a finite set of assignments A such that for any $a \in A$:

$$f(a) = 1, \quad \text{and for any } b < a \text{ we have } f(b) = 0.$$

Definition 2.3 Such assignments are called *minterms*, and we denote by $T(a)$ the term consisting of the variables satisfied by a . In the special case of $a = 0$, we say $T(a) = 1$.

The term $T(a)$ is monotone since it has no negative literals. Also note that if $f \equiv 1$, $A = \{0^n\}$ and if $f \equiv 0$, $A = \emptyset$.

Lemma 2.1 *If ‘ f ’ is a monotone function and A is the set of all minterms of ‘ f ’, then*

$$f = \bigvee_{a \in A} T(a)$$

Proof: Suppose there exists a term for which $T(a)(x) = 1$. Since $T(a)$ is monotone, x must have ones in all positions that are ‘one’ in a . Therefore, $x \geq a$, and because f is monotone and $f(a) = 1$, $f(x) = 1$ as well. Now suppose $f(x) = 1$ for some x . Then there exists a minterm $a \in A$, $a \leq x$, with a corresponding term $T(a)$. Such minterm does exist because even if for all $b \leq a$, $f(b) = 1$, then 0^n is a minterm. Since the term $T(a)$ is monotone, $T(a)(x) = 1$. ■

From the proof follows that for any term $T(a)$, $T(a) \implies f$, i.e., if $T(a)(x) = 1$, then $f(x) = 1$ as well. The above is also true for a disjunction of terms.

Definition 2.4 The *DNF size* of a boolean function f , is the number of terms in a DNF representation of f which has the minimal number of terms. We will refer to the DNF size of f by $\text{size}_{\text{DNF}}(f)$.

Definition 2.5 The *CNF size* of a boolean function f , is the number of clauses in a CNF representation of f which has the minimal number of clauses. We will refer to the CNF size of f by $\text{size}_{\text{CNF}}(f)$.

Lemma 2.2 *The number of minterms of a monotone function ‘ f ’, is exactly $\text{size}_{\text{DNF}}(f)$.*

Proof: Suppose f ’s minimal DNF representation is $f = \bigvee T_j$. We say a minterm a_i is *associated* with a term T_j if it is a minterm of a function $T_j(x)$.

First we show that every minterm is associated with some term. Suppose a_i is a minterm of f . Then $f(a_i) = 1$ and there exists a term T_j , such that $T_j(a_i) = 1$. We also know that for any $b < a_i$, $f(b) = 0$, and therefore $T_j(b) = 0$. Thus, a_i is associated with T_j . Each term T_j has exactly one minterm a_i associated with it, it is the assignment that has ones in all positions of positive literals in T_j , and zeros in all other positions.

We have seen that if a is a minterm, it is associated with some term, and that no other minterms are associated with the same term. Thus, the number of minterms is at most $\text{size}_{\text{DNF}}(f)$, and since $\text{size}_{\text{DNF}}(f)$ is the size of the smallest DNF representation, the number of minterms is exactly $\text{size}_{\text{DNF}}(f)$ (or we could get a smaller representation, by lemma 2.1). ■

2.2 Learning MDNF functions

In order to learn an MDNF function we can use Lemma 2.1. If we just collect all the terms of f , their disjunction will give us f . The algorithm we shall see below was first introduced by Dana Angluin in '88.

Our task is to collect all the minterms. Since testing all possible assignments (using algorithm 2) is inefficient, we take the following approach: We start with an initial hypothesis $h = f = 0$. Now we ask our oracle whether $h = f$. It is likely that the answer will be 'No', and we will also get a counterexample b , such that $f(b) = 1$. Then we find the minterm $a \leq b$ by turning to zero as much bits of b as possible, while keeping $f(b) = 1$. Eventually we will reach the minterm. We add the minterm to our collection of minterms, update our hypothesis, and start over. Since $\bigvee T(a) \implies f$, we will continue receiving only positive counterexamples, i.e., assignments not covered by any minterm we have found so far.

Below is the algorithm `LearnMonotone()` that can learn any monotone function (if `EQ(h)` returns 'No', it updates a global variable `counterexample`):

```

Algorithm LearnMonotone():

h = 0;
while (EQ(h) != TRUE)
  a = counterexample;
  for i=1,2,...n
    if (ai=1)
      b=a; bi=0;
      if (f(b)=1) a = b;          // MQ()
  h = h ∨ T(a);                  // 'a' is a minterm
return h;

```

Figure 1: The `LearnMonotone()` algorithm

Lemma 2.3 *Algorithm 1 returns a hypothesis $h = f$.*

Proof: First we prove that at the time we update h , a is a minterm. If we denote by $a^{(i)}$ the value of a at the beginning of i -th iteration we see that $a^{(i+1)} \leq a^{(i)}$ ($a^{(0)} = \text{counterexample}$). Now, suppose a is not a minterm, i.e., there exists $c < a$, such that $f(c) = 1$. Let i be the first bit which is 1 in a and is 0 in c . During the i -th iteration we had $a^{(i)} \geq a$ and $b = a^{(i)}|b_i = 0^1$, so all bits that were cleared in b are also cleared in c . Thus $c \leq b$. The reason we didn't clear the i -th bit is that $f(b) = 0$. However for our c , $f(c) = 1$, a contradiction to our assumption that f is monotone. Therefore, such c does not exist. Since $\bigvee T(a) \implies f$ we will receive only positive counterexamples, i.e., assignments that are not above any of the minterms we have seen so far. Now suppose we have reached ' a ', a minterm below the counterexample. Since the counterexample is not above any existent minterm, a is a new minterm. Since the number of minterms is finite and equals to $\text{size}_{\text{DNF}}(f)$, the algorithm will terminate after $\text{size}_{\text{DNF}}(f)$ iterations, after collecting all minterms. ■

¹ b is $a^{(i)}$ with i -th bit cleared

Complexity: The algorithm uses exactly $\text{size}_{\text{DNF}}(f)$ equivalence queries and at most $n \cdot \text{size}_{\text{DNF}}(f)$ membership queries.

As we have already seen, at every moment during the algorithm, our hypothesis $h \implies f$.

3 The Monotone Theory

As we have seen earlier, there is a simple and efficient way to learn a monotone boolean function. In this section we will see how to find monotone approximations to a function which is not monotone. Later we will see how to combine these approximations and find the function.

3.1 Monotone approximation of a DNF function

In this section we describe a way to find a monotone approximation $\mathcal{M}(f)$ of a boolean function f .

Definition 3.1 A monotone approximation $\mathcal{M}(f)$ of a boolean function f is the minimal monotone function such that $f \implies \mathcal{M}(f)$. By *minimal* we mean that there exists no $g \neq f$ such that $f \implies g \implies \mathcal{M}(f)$.

Lemma 3.1 $\mathcal{M}(f)(x) = 1$ if, and only if, $\exists y \leq x$ s.t. $f(y) = 1$.

Proof: Suppose $\mathcal{M}(f)(x) = 1$. Now suppose there is no $y \leq x$, such that $f(y) = 1$. Then we can find a smaller approximation to f . We pick up a minterm of $\mathcal{M}(f)$, $a \leq x$. Since $a \leq x$, there is also no $b \leq a$, such that $f(b) = 1$. Because a is a minterm and $f(a) = 0$, we can just throw it away, and replace it by all $c > a$, that differ from a by a single bit. The new minterms cover all assignments covered by a , except a itself. This new monotone approximation is smaller than $\mathcal{M}(f)$, which is a contradiction to the fact that $\mathcal{M}(f)$ is the minimal approximation of f .

Now suppose $\mathcal{M}(f)(x) = 0$. It is clear that there exists no $y \leq x$ such that $f(y) = 1$. Otherwise we must have had $\mathcal{M}(f)(y) = 1$, and since $\mathcal{M}(f)$ is monotone, $\mathcal{M}(f)(x) = 1$, a contradiction to our assumption. ■

Lemma 3.2 An assignment ' a ' is a minterm of $\mathcal{M}(f)$ iff ' a ' is a minterm of f .

Proof: If a is a minterm in $\mathcal{M}(f)$, then $f(a) = 1$, or we could get rid of it, as we have seen in the proof above. For any $b < a$, $\mathcal{M}(f)(b) = 0$, and by lemma 3.1 we have $f(b) = 0$. This means that a is a minterm of f .

If a is a minterm of f , then $f(a) = 1$ and therefore $\mathcal{M}(f)(a) = 1$. For any $b < a$ we know that $f(b) = 0$, i.e., for any $b < a$ there exists no $c \leq b$, such that $f(c) = 1$. By lemma 3.1 we have $\mathcal{M}(f)(b) = 0$. Thus, a is a minterm of $\mathcal{M}(f)$. ■

Definition 3.2 A *local minterm* of a boolean function f is an assignment a , such that $f(a) = 1$, and for any $b < a$ that differ from a by a single bit, $f(b) = 0$.

This is a relaxed definition of a minterm, since it does not require that the condition will hold for all $b < a$, but only for those b 's that differ from a by a single bit.

Of course all local minterms satisfy $\mathcal{M}(f)$, because they satisfy f . Local minterms of a monotone function are its minterms, and it is shown in the proof of lemma 3.4.

Lemma 3.3 *The number of local minterms of a boolean function f , is at most $size_{\text{DNF}}(f)$.*

Proof: The proof is very similar to the proof of lemma 2.2. Suppose f 's minimal DNF representation is $f = \bigvee T_j$. We say a local minterm a_i is *associated* with a term T_j if it is a local minterm of a function $T_j(x)$.

First, we show that every local minterm is associated with some term. Suppose a_i is a local minterm of f . Then $f(a_i) = 1$ and there exists a term T_j , such that $T_j(a_i) = 1$. We also know that for any $b < a_i$, that differ from a_i by a single bit, $f(b) = 0$, and therefore $T_j(b) = 0$. Thus, a_i is associated with T_j .

If the term T_j has a local minterm associated with it, it is the assignment that has ones in all positions of positive literals in T_j , and zeros in all other positions. Therefore only one local minterm can be associated with the same term, i.e., the function from the set of local minterms to the set $\{T_j\}$ is one-to-one, but not onto.

From the above follows that the number of local minterms is at most $size_{\text{DNF}}(f)$. ■

We can learn a monotone approximation to a (not monotone) boolean function f using an algorithm similar to algorithm 1. We would like to collect all minterms of $\mathcal{M}(f)$ (by collecting f 's minterms), but we have a problem: Since the function f is not monotone, we might get negative counterexamples, which our algorithm does not know how to deal with. Negative counterexample means that our hypothesis is too general. However, negative counterexamples are not required to learn $\mathcal{M}(f)$, so at this moment we assume we have an oracle that returns only positive counterexamples (PEQ).

Next we have a problem of telling whether an assignment is a minterm or not. In the algorithm that learns MDNF, we could prove that after the n -th `for` iteration, we have found a minterm. In the case of a general DNF function the proof no longer holds, and for a good reason. We could also do well, if it would be a local minterm, but that's not the case either. The solution is to loop over the `for` loop, until we find a local minterm. There are two reasons why we check only for local minterms. First, their number is at most $size_{\text{DNF}}(f)$, and the real minterms are among them. Second, the complexity of the check is $O(n)$, compared to $O(2^n)$ for a real minterm.

So, first we want to know whether an assignment a is a local minterm of f . Function `IsLocalMinterm(f, a)` on figure 2 gives us the answer.

Lemma 3.4 *Algorithm 2 returns TRUE iff ' a ' is a local minterm of ' f '.*

Proof: The algorithm merely follows the definition. ■

Complexity: The `IsLocalMinterm` function uses at most n membership queries.

```

boolean IsLocalMinterm(f,a):

if (f(a)=0) return FALSE;
for i from 1 to n do
  if (ai=1)
    b = a; bi=0;
    if (f(b)=1) return FALSE;      // MQ()
return TRUE;

```

Figure 2: The IsLocalMinterm(f, a) algorithm

Figure 3 shows the algorithm to learn $\mathcal{M}(f)$.

```

Algorithm LearnMf():

h = 0;
while (PEQ(h) != TRUE)
  a = counterexample;
  do
    for i=1,2,...n
      if (ai = 1)
        b = a; bi=0;
        if (f(b)=1) a = b;      // MQ()
  until IsLocalMinterm(a);
  h = h  $\vee$  T(a);
return h;

```

Figure 3: The LearnMf() algorithm

Lemma 3.5 *The LearnMf() algorithm on figure 3 returns $h = \mathcal{M}(f)$.*

Proof: First, we prove that the `do` loop terminates. If during the `for` iterations we didn't turn any bit to zero, then the definition of a local minterm holds for a , and the `do` loop terminates. Since a is n bits wide, we can turn to zero at most n bits, which means that the `do` loop terminates after at most n iterations.

Now, we were interested only in “real minterms” of f , but in the algorithm we collect local minterms as well. This is ok, since local minterms are always above some “real minterm”, so they don't add any erroneous information to our hypothesis regarding $\mathcal{M}(f)$.

All we have to show now, is that we collect all f 's minterms. Since the minterm itself is not above any other minterm, PEQ will return FALSE, until we find it. However it is possible we won't find all local minterms, since some are covered by other local minterms we have found earlier.

We won't find any local minterm twice, since the positive counterexample is not above any local minterm we have seen so far. Therefore the `while` loop terminates after at most $\text{size}_{\text{DNF}}(f)$ iterations. ■

Complexity: There are at most $\text{size}_{\text{DNF}}(f)$ `while` iterations, which means there are $\text{size}_{\text{DNF}}(f)$ positive equivalence queries. Now there are at most n `do` loop iterations, containing n iterations of the `for` loop, with at most n membership queries. Also the `IsLocalMinterm()` function has n `for` iterations, and at most n membership queries.

The total time complexity is $O(n^2 \cdot \text{size}_{\text{DNF}}(f))$. The algorithm uses at most $\text{size}_{\text{DNF}}(f)$ positive equivalence queries and at most $2n^2 \cdot \text{size}_{\text{DNF}}(f)$ membership queries.

3.2 Other monotone approximations

Our discussion in the previous section was based on the definition of the partial order “ \leq ”. Now we shall see how to find monotone approximations based on other partial orders.

Definition 3.3 For three assignments $a, b, c \in \{0, 1\}^n$, we say $a \leq_c b$, if $a \oplus c \leq b \oplus c$.

The new partial order is a variation on the partial order we have defined earlier, using a parameter c . We can see it as a result of a function that operates on the elements of the ‘ \leq ’ relation, and transfers each pair $a \leq b$ to a pair $a \oplus c \leq_c b \oplus c$.

When $c = 0$, we are back to our old partial order (\leq), with all corresponding definitions.

Definition 3.4 A boolean function f is c -monotone, if for any $a \leq_c b$, $f(a) \leq f(b)$. Minterms and local minterms are defined analogously regarding the new partial order \leq_c .

Lemma 3.6 A boolean function f is c -monotone iff $f(x \oplus c)$ is monotone.

Proof: f is c -monotone iff for any $a \leq_c b$, $f(a) \leq f(b)$. This is true iff for any $a' \leq b'$, $f(a' \oplus c) \leq f(b' \oplus c)$, i.e., iff $f(x \oplus c)$ is monotone. ■

A c -monotone function must have all variables appearing either positive or negative, but not both. Then c has ones in all positions corresponding to literals that are negative. In $f(x \oplus c)$, all literals appear positive, and therefore it is monotone.

Definition 3.5 A c -monotone approximation $\mathcal{M}_c(f)$ of a boolean function f is the minimal c -monotone function such that $f \implies \mathcal{M}_c(f)$. By *minimal* we mean that there exists no $g \neq f$ such that $f \implies g \implies \mathcal{M}_c(f)$. For $c = 0$, we get $\mathcal{M}_{(0)}(f) = \mathcal{M}(f)$.

Lemma 3.7 Lemmas 3.1, 3.2 and 3.3 hold for the new partial order:

3.1 $\mathcal{M}_c(f)(x) = 1$ if, and only if, $\exists y \leq_c x$ s.t. $f(y) = 1$.

3.2 An assignment ‘ a ’ is a c -minterm of $\mathcal{M}_c(f)$ iff ‘ a ’ is a c -minterm of f .

3.3 The number of local c -minterms of a boolean function f , is at most $\text{size}_{\text{DNF}}(f)$.

Proof: The proofs of lemmas 3.1 and 3.2 hold for the new partial order, and the new proof of lemma 3.3 is brought here once again:

Suppose f 's minimal DNF representation is $f = \bigvee T_j$. We say a local c -minterm a_i is *associated* with a term T_j if it is a local c -minterm of a function $T_j(x)$.

First, we show that every local c -minterm is associated with some term. Suppose a_i is a local c -minterm of f . Then $f(a_i) = 1$ and there exists a term T_j , such that $T_j(a_i) = 1$. We also know that for any $b <_c a_i$, that differ from a_i by a single bit, $f(b) = 0$, and therefore $T_j(b) = 0$. Thus, a_i is associated with T_j .

If the term T_j has a local c -minterm associated with it, it is the assignment that has ones in all positions of positive literals in T_j , zeros in in all positions of negative literals in T_j , and all other bits have values of the corresponding bits in c . Therefore only one local minterm can be associated with the same term.

From the above follows that the number of local minterms is at most $\text{size}_{\text{DNF}}(f)$. ■

Now we will see how to learn a c -monotone approximation to a boolean function f , using an algorithm similar to algorithm 3. All we have to do is to modify the `for` loop and the `IsLocalMinterm()` function, and prove its correctness once again. The algorithm appears on figure 4.

```

Algorithm LearnMcf():
    h = 0;
    while (PEQ(h) != TRUE)
        a = counterexample;
        do
            for i=1,2,...n
                if (ai ≠ ci) // greaterc than
                    b = a; bi = ci;
                    if (f(b)=1) a = b; // MQ()
            until IsLocalMintermc(a);
        h = h ∨ T(a ⊕ c)(x ⊕ c);
    return h;

```

Figure 4: The LearnM_cf() algorithm

Lemma 3.8 *The IsLocalMinterm_c(f, a) function returns TRUE if, and only if, 'a' is a local minterm of f regarding the partial order ' \leq_c '.*

Proof: If the i -th bit in a differs from the i -th bit in c , then b that equals to a with i -th bit equal to c_i , is less than a . The above is true, since after xoring with c , the i -th bit of $a \oplus c$ will be 1, and the i -th bit of $b \oplus c$ will be 0. Therefore, we check all b 's that are less than a , and differ from a , by a single bit. If for one of them $f(b) = 1$, it is not a minterm and we return FALSE. If for all b 's we have checked $f(b) = 0$, it is a minterm, and the function returns TRUE. ■

```

boolean IsLocalMintermc(f, a):

if (f(a)=0) return FALSE;
for i from 1 to n do
  if (ai ≠ ci)
    b = a; bi = ci;
    if (f(b)=1) return FALSE;      // MQ()
return TRUE;

```

Figure 5: The IsLocalMinterm_c(f, a) algorithm

Lemma 3.9 *The LearnM_cf() algorithm returns a hypothesis $h = \mathcal{M}_c(f)$.*

Proof: First we show that the `do` loop terminates. In the `for` loop we set f 's bits to match as much bits of c as possible while keeping $f(a) = 1$. If we didn't change any bit during the `for` loop iterations then we have a local minterm, and the call to IsLocalMinterm_c() returns TRUE. Notice that the if we didn't change any bit, IsLocalMinterm_c() function will check the same bit values as the `for` loop, and will make the same decisions in both `if` statements. Since in our `for` loop we never reached the statement `a=b`; in the IsLocalMinterm_c() function we will never reach the `return FALSE`; statement. Since we can change at most n bits, and we change them only in one direction (to match c_i 's bits) the `do` loop will terminate after at most n iterations.

Now we show that $T(a \oplus c)(x \oplus c)$ is the desired term, i.e., a is the minterm of $T(a \oplus c)(x \oplus c)$. The variable v_i appears in the term only if $a_i \neq c_i$ ($a \oplus c = 1$). Now, v_i appears positive only if $c_i = 0$, and therefore $a_i = 1$. In such case a_i satisfies v_i . If $c_i = 1$, then v_i appears negative, i.e., as \bar{v}_i . However, now $a_i = 0$, and again satisfies v_i 's literal. We have seen that a satisfies the term, and it is clear that if we change any bit $a_i \neq c_i$, it will no longer satisfy v_i , and the value of the term will be 0. This means that a is a local minterm, but since a term has only one minterm, a is the one.

We also want to be sure that we do not collect any unnecessary information. Just as before, all local minterms are above(\geq_c) some real term. All we have to show, is that we collect all minterms. The proof is very similar to what we have done for the LearnMf() algorithm. A real minterm is not above any other minterm, so if we do not collect it, our hypothesis will return FALSE for it, and then PEQ can tell us about it. Thus for PEQ to return TRUE, we must collect all minterms. The `while` loop will terminate since we never collect the same minterm twice (recall that a counterexample is not above any minterm we have seen so far), and the number of local minterms is finite, and is at most $\text{size}_{\text{DNF}}(f)$. ■

Complexity: Just like in algorithm LearnMf(), the algorithm uses at most $\text{size}_{\text{DNF}}(f)$ positive equivalence queries and at most $2n^2 \cdot \text{size}_{\text{DNF}}(f)$ membership queries.

Lemma 3.10 *$\mathcal{M}_c(f)$'s properties:*

1. If $f \implies g$, then $\mathcal{M}_c(f) \implies \mathcal{M}_c(g)$.

2. If C is a clause and $C(c) = 0$, then $\mathcal{M}_c(C) = C$.

Proof:

1. Suppose $\mathcal{M}_c(f)(x) = 1$. Then, there exists a $a \leq_c x$, such that $f(a) = 1$. Since $f \implies g$, we also have $g(a) = 1$, and because $g \implies \mathcal{M}_c(g)$, we have $\mathcal{M}_c(g)(a) = 1$. $\mathcal{M}_c(g)$ is c -monotone and therefore $\mathcal{M}_c(g)(x) = 1$.
2. We show that C is already c -monotone. By lemma 3.6 it is enough to show that $M = C(x \oplus c)$ is monotone. We know that $M(c \oplus c) = M(0) = 0$. This means M , which is still a clause, has no negative literals, and therefore is monotone. By the definition of $\mathcal{M}_c(C)$ it is the minimal c -monotone function such that $C \implies \mathcal{M}_c(C)$. Since C is already c -monotone, $\mathcal{M}_c(C) = C$.

■

3.3 The monotone basis

Why do we need these c -monotone approximations? The following definitions reveal part of the answer.

Definition 3.6 A set of assignments $A = \{a_1, \dots, a_t\}$ is an M -basis for a class of boolean functions, if for any function from the class:

$$f \equiv \bigwedge_{a_i \in A} \mathcal{M}_{a_i}(f)$$

Definition 3.7 The *monotone dimension*, or $Mdim$ of a class of boolean functions, is the size of its smallest M-basis.

Now we want to know whether any class has an M-basis.

Lemma 3.11 For any boolean function f :

$$f \equiv \bigwedge_{c \in \{0,1\}^n} \mathcal{M}_c(f)$$

Proof: First, by its definition, for any c , $f \implies \mathcal{M}_c(f)$, so

$$f \implies \bigwedge_{c \in \{0,1\}^n} \mathcal{M}_c(f)$$

Now we have to prove the opposite direction. Suppose $\bigwedge_{c \in \{0,1\}^n} \mathcal{M}_c(f)(x) = 1$. Then for any c , $\mathcal{M}_c(f)(x) = 1$. We focus on $c = x$, and recall that c is a minterm of $\mathcal{M}_c(f)$. By lemma 3.2 restated for the new partial order (3.7), we get that $f(c = x) = 1$. Hence, $\bigwedge_{c \in \{0,1\}^n} \mathcal{M}_c(f) \implies f$. ■

Lemma 3.12 A set of assignments A is an M -basis for a class of boolean functions \mathcal{C} , if and only if, every boolean function $f \in \mathcal{C}$ can be represented as a CNF formula $f = C_1 \wedge \dots \wedge C_r$, such that every clause in this CNF is falsified by some assignment in A .

Proof: For a CNF formula, $f \implies C_i$, and by item 1 in lemma 3.10 we have $\mathcal{M}_a(f) \implies \mathcal{M}_a(C_i)$. By item 2, we know that there exists $a \in A$, such that $C_i(a) = 0$ and therefore $\mathcal{M}_a(C_i) = C_i$. Thus, $\mathcal{M}_a(f) \implies C_i$.

Now if we take the conjunction of all clauses we get that $\bigwedge \mathcal{M}_a(f) \implies \bigwedge_{i=1}^r C_i$, or in other words $\bigwedge \mathcal{M}_a(f) \implies f$. We know that $f \implies \mathcal{M}_a(f)$, and therefore

$$\bigwedge \mathcal{M}_a(f) = f$$

Some a 's may not appear in the conjunction above, because they don't falsify any clause (we can throw them from A). But, we can add them to the conjunction because they don't change its value ($f \implies \mathcal{M}_c(f)$ for any c). Hence,

$$\bigwedge_{a \in A} \mathcal{M}_a(f) = f$$

and A is an M-basis. ■

Lemma 3.13 For two classes of boolean functions \mathcal{C}_1 and \mathcal{C}_2 and M-bases A_1 and A_2 , the class $\mathcal{C}_1 \wedge \mathcal{C}_2$ (conjunctions of a function from \mathcal{C}_1 and \mathcal{C}_2) has an M-basis $A_1 \cup A_2$.

Proof: A (possible) CNF representation of a conjunction of functions from \mathcal{C}_1 and \mathcal{C}_2 is the conjunction of their CNF representations. Then each clause is falsified by an assignment from A_1 or A_2 . Hence, $A_1 \cup A_2$ is an M-basis for $\mathcal{C}_1 \wedge \mathcal{C}_2$. ■

Definition 3.8 For a decision tree T , the *decision tree size* of T is the number of leaves in T , and denote it by $\text{size}_{\text{DT}}(T)$. For a boolean function f , the *decision tree size* of f is

$$\text{size}_{\text{DT}}(f) = \min_{T \equiv f} \text{size}_{\text{DT}}(T)$$

Lemma 3.14 For a boolean function f , we have

1. $\text{size}_{\text{DT}}(f) \geq \text{size}_{\text{DNF}}(f) + \text{size}_{\text{CNF}}(f)$.
2. $Mdim(\{f\}) \leq \text{size}_{\text{CNF}}(f)$.

Proof:

1. We can represent a decision tree as a DNF, where each leaf marked with 1 corresponds to a DNF term, and as a CNF, where each leaf marked with 0 corresponds to a CNF clause.
2. Follows directly from lemma 3.12. ■

4 Learning boolean functions

4.1 Learning A Class With Known M-Basis

Now to learn a class of functions \mathcal{C} , with a known M-basis A , we can simply run the $\text{LearnM}_c f()$ algorithm for each element of the basis (it can be done in parallel) and taking the conjunction of the outputs of all algorithms.

Yet, we still have the problem that we use the positive equivalence queries oracle. We need an algorithm that will use only equivalence and membership queries, like the Λ -Algorithm on figure 6.

```

The  $\Lambda$ -Algorithm(A):
A = {a1, ..., at}

for i=1,2,...,t
  Start running algorithm LearnMaif()
  until it asks for PEQ;
  hi = LearnMaif()'s current hypothesis, h
H =  $\bigwedge_{i=0}^t h_i$ ; // hypothesis of f

while (EQ(H) != TRUE)
  v = counterexample;
  for i=1,2,...,t
    if (hi(v)=0) // positive counterexample
      provide LearnMaif() with PEQ=FALSE and
      counterexample = v;
      continue LearnMaif() until next PEQ;
      hi = LearnMaif()'s current hypothesis;

  H =  $\bigwedge_{i=0}^t h_i$ ; // update our hypothesis of f

kill all algorithms LearnMaif();
return H;

```

Figure 6: The Λ -Algorithm()

Proposition A *If $EQ(H)=\text{FALSE}$, 'v' is a positive counterexample, i.e., $H(v) = 0$.*

Proof: We know that $\bigwedge_{a_i \in A} \mathcal{M}_{a_i}(f) = f$, and that for each algorithm, $h_i \implies \mathcal{M}_{a_i}(f)$. Therefore, we get that $\bigwedge_{a_i \in A} h_i \implies f$, or in other words, $H \implies f$. If $H \implies f$, and $H(v) \neq f(v)$, the only possibility is that $H(v) = 0$ and $f(v) = 1$, which means v is a positive counterexample. ■

Lemma 4.1 *The hypothesis Λ -Algorithm() returns, is the requested function f .*

Proof: The algorithm controls the run of all $\text{Learn}_{\mathcal{M}_{a_i}f}()$ processes, reads their most correct hypotheses and simulates the PEQ oracles. However, we can only answer the PEQ query if the answer is FALSE. We are not sure of the positive answer until we get $EQ(H) = \text{TRUE}$, but then we do not care, since we already have the answer.

Before entering the **while** loop, we do some initializations, and advance all algorithms until their first PEQ request. Our initial hypothesis is $H = 0$, because all $h_i = 0$.

First we show that the **while** loop terminates. Suppose we have entered the **while** loop. Then by proposition A, v is a positive counterexample. If v is a positive counterexample, it must be a positive counterexample for at least one h_i . In such case, $\text{Learn}_{\mathcal{M}_{a_i}f}()$ can be released and given the positive counterexample.

Since each algorithm will reach its final $\mathcal{M}_{a_i}(f)$ after at most $\text{size}_{\text{DNF}}(f)$ PEQs, at some point of the algorithm run, we will have

$$H = \bigwedge_{i=0}^t h_i = \bigwedge_{i=0}^t \mathcal{M}_{a_i}(f) = f$$

During each **while** iteration we answer at least one PEQ. Therefore after at most $t \cdot \text{size}_{\text{DNF}}(f)$ iterations, we will have $H = f$, and the **while** loop will terminate with $H = f$. ■

Complexity: As we have seen in the proof, the algorithm performs at most $|A| \cdot \text{size}_{\text{DNF}}(f)$ equivalence queries, before receiving TRUE. The number of membership queries is at most $2|A|n^2 \cdot \text{size}_{\text{DNF}}(f)$, since the algorithm $\text{Learn}_{\mathcal{M}_c f}()$ performs at most $2n^2 \cdot \text{size}_{\text{DNF}}(f)$ membership queries.

Note that parts of the algorithm can be run in parallel, especially the **for** loops, that do most of the heavy processing. The output of our Λ -Algorithm is the same as the output of the Λ -Algorithm from the paper. However, while the Λ -Algorithm in the paper, does it all at once, we have split the work into small pieces, and proved their correctness separately.

Theorem 1 *For assignments $A = \{a_1, \dots, a_t\}$, let $\Lambda(A)$ be the set of all boolean functions that can be represented as CNF, in which every clause is falsified by some assignment in A . Then any $f \in \Lambda(A)$ is learnable in polynomial time in the number of variables, the DNF size of f and $|A|$.*

Proof: From lemma 3.12 follows that A is an M-basis for $\Lambda(A)$. With known M-basis A , any function $f \in \Lambda(A)$ can be learned using the Λ -Algorithm in a time polynomial in the number of variables (n), the DNF size of f ($\text{size}_{\text{DNF}}(f)$) and $|A|$. ■

4.2 Learning a class with unknown M-basis

In this section we show an algorithm to learn any boolean function f , in a time polynomial in its DNF size, its CNF size and the number of variables.

We know that $A = \{0, 1\}^n$ is an M-basis for any boolean function. However, using it is very inefficient. Apparently we could convert our function to CNF,

find the assignments that falsify its clauses and use it as a base. The problem with that approach is that if we know f , we don't need to learn it, and in most cases we want to learn a function we don't know.

The solution is to learn the function along with its M-basis. First we assume our M-basis is empty, and $f \equiv 1$ (we define an empty conjunction to be 1). If it is the M-basis, the algorithm ends here. Otherwise we receive a negative counterexample. In the Λ -Algorithm we should never receive negative counterexamples. This means we have a problem with our M-basis. We can prove that the negative counterexample we receive is part of the M-basis. We add it to the basis and start over, until we learn our function. Since $\{0,1\}^n$ is a basis, this algorithm will eventually stop. However, we want to show that the algorithm will stop after $\text{size}_{\text{CNF}}(f)$ iterations.

Lemma 4.2 *Let $C_1 \wedge \dots \wedge C_r$ be the CNF formula for f . Suppose our current basis is A , and there is a subset of f 's clauses $S = \{C_{i_1}, \dots, C_{i_m}\}$ that are falsified by some assignment in A . If we get a negative counterexample ' v ', it will falsify a new clause $C \notin S$.*

Proof: If v is a negative counterexample, then $H(v) = 1$. We want to show that for any $C \in S$, $C(v) = 1$. Since $f(v) = 0$, there is another clause that is falsified by v .

For each clause $C \in S$ we know it is falsified by some assignment $a_i \in A$, i.e., $C(a_i) = 0$. From item 2 in lemma 3.10 follows that $\mathcal{M}_{a_i}(C) = C$. Since $f \implies C$, by item 1 we know that $\mathcal{M}_{a_i}(f) \implies \mathcal{M}_{a_i}(C)$. Combining the two results above we get

$$\mathcal{M}_{a_i}(f) \implies C$$

Since $H(v) = 1$, all $h_i(v) = 1$, and because $h_i \implies \mathcal{M}_{a_i}(f)$ we have $\mathcal{M}_{a_i}(f)(v) = 1$. Hence $C(v) = 1$. ■

The algorithm on figure 7 shows the algorithm to learn any boolean function:

The CDNF-Algorithm:

```

A =  $\emptyset$ ;
while (TRUE)
  Run  $\Lambda$ -Algorithm(A) until
    it terminates or receives a negative
    counterexample ' $v$ ';
  If the algorithm terminated
    with EQ(H)=TRUE, return H;
  A = A  $\cup$  { $v$ };
  Kill the  $\Lambda$ -Algorithm and continue;

```

Figure 7: The CDNF-Algorithm

The correctness of this algorithm follows from lemma 4.2. Note that it is not necessary to abort the algorithms when a negative equivalence query is encountered. We can just start running a new $\text{LearnM}_c\text{f}()$ algorithm (with its

initial hypothesis $h = 0$) and ask the equivalence query again. Then we will receive a positive counterexample, since $H = 0$, and continue from there, as if v has been in the basis from the beginning.

Complexity: The `while` loop will run at most $\text{size}_{\text{CNF}}(f)$ times, therefore the complexity is polynomial in $\text{size}_{\text{DNF}}(f)$, $\text{size}_{\text{CNF}}(f)$ and n .

Theorem 2 *Any boolean function is learnable in polynomial time in the number of variables, its DNF size and its CNF size.*

Proof: The `CDNF-Algorithm` learns any boolean function in time polynomial in the number of variables, its DNF size and its CNF size. ■

4.3 Results

In this section we show the results of the monotone theory.

Result 1 *The CDNF class of boolean functions is the class of boolean functions which CNF size is polynomial in its DNF size. Any function in the CDNF class is learnable in time polynomial in its DNF size and the number of variables.*

Proof: Follows directly from theorem 2. ■

Result 2 *The class of k -almost monotone DNF is the class of monotone DNF formulas that has at most a constant number (k) non-monotone terms. Any k -almost monotone DNF is learnable in polynomial time in its DNF size and n . Also the conjunction of any two functions from this class is learnable in time polynomial in its DNF size and n .*

Proof: If we convert the k -almost monotone DNF to CNF, by taking the conjunction of all possible disjunctions of one literal from each term, we note that every clause contains at most k negative literals. If we take a conjunction of k -almost monotone DNF functions, the conjunction of their CNF representations will also have clauses with at most k negated variables.

The following set is the M-basis for this class:

$$A = \{ \text{all vectors of length } n \text{ with at most } k \text{ ones} \}$$

Any clause with at most k negated variables is falsified by some element from A . If C is a clause with $l \leq k$ negative variables, the assignment that falsifies it, has ones in all positions of negative literals and zeros in all positions of positive literals. Since $l \leq k$, the number of ones in this assignment is less than k . Since k is constant the size of A is polynomial in n . Now, by theorem 1, the result follows. ■

Note that the class of k -almost monotone DNF includes the class of k -term DNF and the class of monotone DNF.

Result 3 *Any boolean function is learnable in polynomial time in its decision tree size and n .*

Proof: From theorem 2, we see that the function is learnable in time polynomial in its DNF size, CNF size and n . By lemma 3.14, $\text{size}_{\text{DT}}(f) \geq \text{size}_{\text{DNF}}(f) + \text{size}_{\text{CNF}}(f)$, and therefore the function is learnable in time that is polynomial in its decision tree size and n . ■

Result 4 *Decision trees over terms are decision trees where each node contains a term. Decision trees of constant depth (k) over terms, are learnable in time polynomial in n .*

Proof: We show that decision trees of depth k over terms, have CNF size and DNF size of at most $(2n)^k$. A decision tree of depth k over variables is a k -CNF and a k -DNF. The DNF of a decision tree over variables is a disjunction of terms leading to 1s. Each term is a conjunction of literals on the path to the leaf. When there are terms inside the nodes, the path to a leaf labeled ‘1’, is a conjunction of terms and negated terms. Note that their total number is at most k . We have no problem with a conjunction of terms, it is a term of at most n literals.

Now we look at the conjunction of the negated terms. A conjunction of negated terms is a conjunction of clauses with negated variables. When we open these conjunctions, we get a total of at most n^k terms. Since k is constant, this number is polynomial in n , just as we wanted. So, for each leaf marked with 1, we have at most n^k terms. The number of leaves in a binary tree of constant depth k , is at most 2^k . Therefore the number of terms in a DNF representation is at most $(2n)^k$. Duality implies that the CNF size is at most $(2n)^k$ clauses. Now, by theorem 2, the result follows. ■