

A Proposed Algorithm Toward Uniform-distribution Monotone DNF Learning

A Thesis

Presented to the Faculty

of the Mathematics and Computer Science Department

McAnulty College and Graduate School of Liberal Arts

Duquesne University

in partial fulfillment of

the requirements for the degree of

Master of Science in Computational Mathematics

by

Wenzhu Bi

07/30/2004

Wenzhu Bi

**A Proposed Algorithm Toward Uniform-distribution Monotone
DNF Learning**

Master of Science in Computational Mathematics

07/30/2004

APPROVED _____
Jeffrey Jackson, Ph.D., Associate Professor of Computer Science

APPROVED _____
Donald L. Simon, Ph.D., Associate Professor of Computer Science

APPROVED _____
Frank D'Amico, Ph.D., Professor of Mathematics

APPROVED _____
Kathleen Taylor, Ph.D., Graduate Director
Computational Mathematics Department

APPROVED _____
Constance D. Ramirez, Ph.D., Dean
McAnulty College and Graduate School of Liberal Arts

1 Introduction

1.1 Proposed Problem

In 1984 Valiant[1] introduced the distribution-independent model of Probably Approximately Correct (PAC) learning from random examples and brought up the problem of whether polynomial-size DNF functions are PAC learnable in polynomial time. It has been about twenty years that the DNF learning problem has been widely regarded as one of the most important —and challenging — open questions in Computational Learning Theory.

It is well known that learning monotone DNF in the distribution-independent setting is equivalent to learning general DNF[2], so a lot of work has been done in learning monotone DNF. Because of the difficulty of learning monotone DNF in the distribution-independent setting, the simpler case — the Monotone DNF with uniformly-distributed examples — is studied.

Here we develop an algorithm that learns a threshold function in polynomial time from the truth table generated by a monotone DNF function. We use the whole truth table instead of uniform-distributed examples. We can calculate the Fourier coefficient by using the whole truth table; the Fourier coefficients can also be estimated with high accuracy by using the uniformly-distributed examples. So if our algorithm performs well by using the whole truth table, we may also get good test results by using the uniformly-distributed examples. The reason that we use the whole truth table is that we can eliminate the variability added by estimating the Fourier coefficients.

The terminology used above will be explained in the following sections.

1.2 Definitions and Notations

For the purpose of easily understanding the definition of *Disjunctive Normal Form*, we will talk about some basic conceptions in *Boolean Logic*.

Boolean Logic is a familiar mathematical notation for expressing compound statements such as the following:

1. Either it is not raining now or the cane is not in the corner.
2. It is raining and the cane is not in the corner.

1.2.1 Literals

In *Boolean Logic* we use *Boolean variables* x_1, x_2, \dots to stand for the individual statements such as “it is raining now” or “the cane is in the corner”. Each variable denotes a statement that can in principle be *true* or *false* independently of the truth value of the others. We then use *Boolean connectives*, such as \wedge and \vee , to combine Boolean variables to form more complicated *Boolean formulae*. *Boolean variables* are also called *Literals*. Let $X = \{x_1, x_2, \dots, x_n\}$ be a finite set of *Boolean variables*, and let $\bar{X} = \{\bar{x}_1, \bar{x}_2, \dots, \bar{x}_n\}$, where the $\bar{x}_1, \bar{x}_2, \dots, \bar{x}_n$ are new symbols standing for the *negations*. We call the elements of $X \cup \bar{X}$ *literals*.

1.2.2 Conjunctions

Let p and q be literals. The formula “ p and q ”, denoted by $p \wedge q$ or pq , is true when both p and q are *true* and is *false* otherwise. The formula “ $p \wedge q$ ” is called the *conjunction* of p and q .

1.2.3 Disjunctions

Let p and q be literals. The formula “ p or q ”, denoted by $p \vee q$ or $p + q$, is false when both p and q are *false* and is *true* otherwise. The formula “ $p \vee q$ ” is called the *disjunction* of p and q .

1.2.4 Disjunctive Normal Form

A Disjunctive Normal Form (DNF) formula is a disjunction of conjunctions of Boolean literals. Conjunctions in a DNF are called “terms”. The size of a DNF formula is defined as the number of terms that it has. For example,

$$f = x_1\bar{x}_2x_3 + x_4\bar{x}_5x_6$$

is a DNF function with size 2.

Then if we use x_1 to denote “it is raining now” and use x_2 to denote “the cane is in the corner”, then

1. The negation of x_2 — \bar{x}_2 denotes “the cane is not in the corner”.
2. “Either it is not raining now or the cane is not in the corner” can be expressed as “ $\bar{x}_1 \vee \bar{x}_2$ ”.
3. “It is raining and the cane is not in the corner” can be expressed as “ $x_1 \wedge \bar{x}_2$ ”.

1.2.5 Monotone DNF

A *monotone* DNF is a DNF with no negated variables. For example,

$$g = x_1 + x_2x_3 + x_4x_5 + x_6x_7x_8$$

f is a monotone DNF function with size 4.

1.3 PAC-Learning in DNF-Learning problem

Before defining the PAC-Learning, we will define the supporting concept *example oracle*. An *example oracle* $EX(f, D)$ for a target function f with respect to $D(EX(f, D))$ is an oracle that on request draws an instance x at random according to probability distribution D and returns the example $\langle x, f(x) \rangle$.

We say that the concept class DNF is **PAC learnable** if there exists an algorithm L with the following property: for every DNF c , for every distribution D of the examples over the n -bit vector space $\{0, 1\}^n$, and for all $0 < \epsilon < 1/2$ and $0 < \delta < 1/2$, $L(EX(f, D), \epsilon, \delta)$ runs in time polynomial in n , s (s is the number of terms in c), $1/\epsilon$ and $1/\delta$ to output a *hypothesis concept* h satisfying $Pr(h \neq c) \leq \epsilon$.

The parameter δ is defined as the degree of the confidence of the algorithm. This parameter is not used in our later work because our work is still in the earlier stage to test some specific functions.

2 Present status of the proposed problem

Now let us go over the previous work in the field of learning monotone DNF. Because of the lack of progress on learning monotone DNF in the distribution-independent setting, instead of approaching this result directly, many researchers study the restricted versions, such as learning monotone DNF with uniformly-distributed examples. Hancock and Mansour [3] gave a polynomial time algorithm for learning monotone read- k DNF (DNF in which every variable appears at most k times in which k is a constant) under constant-bounded product distributions. Then there is a series of research work, done by Verbeurgt [4], Kucera [5], Sakai, Maruoka [6], Bshouty [7] and Tamon [8], to improve monotone DNF learning. The latest best work is done by Servedio [9] who proves that that the class of monotone $2^{O(\sqrt{\log n})}$ -term

DNF formulae can be PAC learned in polynomial time under the uniform distribution from random examples only.

3 Suggested methodology

3.1 The Proposed Algorithm

3.1.1 Concepts and Definitions used in the Algorithm

- The threshold function used in this algorithm is defined particularly as

$$T(\vec{x}) = \text{sign}[F(\vec{x})] = \begin{cases} +1 & \text{if } F(\vec{x}) > 0 \\ -1 & \text{if } F(\vec{x}) < 0 \end{cases}$$

We will guarantee that $F(\vec{x})$ is not equal to 0 in this algorithm.

- The parity function $\chi_{\vec{a}}(\vec{x})$ is defined as:

$$\chi_{\vec{a}}(\vec{x}) = (-1)^{\vec{a} \cdot \vec{x}}$$

That is, $\chi_{\vec{a}}(\vec{x})$ is the Boolean function that is 1 when the parity of the number of 1's in \vec{x} indexed by \vec{a} is even and is -1 otherwise. If the number of 1's in \vec{a} is even, we call \vec{a} “even”; otherwise, we call it “odd”.

The number of 1's in the parity is also called *degree* of the parity. Assume that we have two parity functions A and B , if the degree of A is greater than the degree of B , we can say that A has a *higher degree* than B or B has a *lower degree* than A . If the number of 1's is 0, we call the parity function *constant parity*.

- The threshold function that we used in the algorithm is defined as the following:

$$T(\vec{x}) = \text{sign}[F(\vec{x})]$$

in which

$$F(\vec{x}) = \sum_{\vec{a} \in S} \hat{F}(\vec{a}) \chi_{\vec{a}}(\vec{x})$$

where

1. S is a set of n -bit vectors and for each n -bit vector $\vec{a} \in S$ there is one integral coefficient $\hat{F}(\vec{a})$.

$$2. \text{sign}[\hat{F}(\vec{a})] = (-1)^{|\chi_{\vec{a}}|}$$

It means that if the number of 1's in \vec{a} is even, the sign of the Fourier coefficient of $\hat{F}(\vec{a})$ is positive; otherwise it is negative.

$$3. \forall \vec{a}' (\text{except the vector } 0^n) \text{ such that } \vec{a} \subseteq \vec{a}', |\hat{F}(\vec{a})| \geq |\hat{F}(\vec{a}')|$$

4. Each parity in the set S in the hypothesis should be a subset of a term in the target function (Please notice that this subsets rule is not programmed in the algorithm. But when we check the test results, if this rule is broken, it may be a sign that the algorithm will not work)

3.1.2 How the Hypothesis is Related to the Target Function

We suppose that if the value of a Boolean function is *false*, the function has a value “-1” and if the value of a Boolean function is *true*, the function has a value “1”. We know that a conjunction can be expressed as a sum of some parity functions. In the following examples, suppose $n = 4$:

- $x_1x_2 = -\frac{1}{2}\chi_{0000} - \frac{1}{2}\chi_{0001} - \frac{1}{2}\chi_{0010} + \frac{1}{2}\chi_{0011}$
- $x_2x_3x_4 = -\frac{3}{4}\chi_{0000} - \frac{1}{4}\chi_{0010} - \frac{1}{4}\chi_{0100} - \frac{1}{4}\chi_{1000} + \frac{1}{4}\chi_{0110} + \frac{1}{4}\chi_{1010} + \frac{1}{4}\chi_{1100} - \frac{1}{4}\chi_{1110}$
- $x_1x_2x_3x_4 = -\frac{7}{8}\chi_{0000} - \frac{1}{8}\chi_{0001} - \frac{1}{8}\chi_{0010} - \frac{1}{8}\chi_{0100} - \frac{1}{8}\chi_{1000} + \frac{1}{8}\chi_{0011} + \frac{1}{8}\chi_{0101} + \frac{1}{8}\chi_{0110} + \frac{1}{8}\chi_{1001} + \frac{1}{8}\chi_{1010} + \frac{1}{8}\chi_{1100} - \frac{1}{8}\chi_{0111} - \frac{1}{8}\chi_{1011} - \frac{1}{8}\chi_{1110} + \frac{1}{8}\chi_{1111}$

Notice that except for the constant parity, the odd degree parities all have negative coefficients $-\frac{1}{2^{l-1}}$ (l is the number of literals in the conjunction) while the even degree parities all have positive coefficients $\frac{1}{2^{l-1}}$. Since the denominators of the coefficients are all powers of 2, it is easy to convert the coefficients to integers without changing sign.

We can add conjunctions up. For example:

$$\begin{aligned} f &= x_1x_2 + x_2x_3x_4 \\ &= \left(-\frac{1}{2}\chi_{0000} - \frac{1}{2}\chi_{0001} - \frac{1}{2}\chi_{0010} + \frac{1}{2}\chi_{0011}\right) \\ &+ \left(-\frac{3}{4}\chi_{0000} - \frac{1}{4}\chi_{0010} - \frac{1}{4}\chi_{0100} - \frac{1}{4}\chi_{1000} + \frac{1}{4}\chi_{0110} + \frac{1}{4}\chi_{1010} + \frac{1}{4}\chi_{1100} - \frac{1}{4}\chi_{1110}\right) \\ &= -\frac{5}{4}\chi_{0000} - \frac{1}{2}\chi_{0001} - \frac{3}{4}\chi_{0010} - \frac{1}{4}\chi_{0100} - \frac{1}{4}\chi_{1000} + \frac{1}{2}\chi_{0011} + \frac{1}{4}\chi_{0110} + \frac{1}{4}\chi_{1010} + \frac{1}{4}\chi_{1100} - \frac{1}{4}\chi_{1110} \end{aligned}$$

Notice that the following property: Except for the constant parity, the odd degree parities all have negative coefficients while the even degree parities all have positive coefficients. All the denominators of the coefficients are still powers of 2, so it is easy to convert the coefficients to integers. It can also be noticed that the sum will produce larger magnitude coefficients on lower degree parities.

A monotone DNF is an OR of AND's, so for a monotone DNF with t terms it is easy to see that:

1. If all the t terms are *false*, then all the terms (each term is a conjunction) have the value -1 and the sum of the terms has a value of $-t$. The *DNF* function is *false*. If we add $(t-1)$ to the sum of the terms, it becomes -1 . If we set that $DNF = \text{sign}[(\text{the sum of the terms}) + (t-1)]$, then when DNF is *false*, DNF has a value of -1 .
2. If at least one term is true, then the *DNF* function is true and the sum of terms may have a value in the array with t elements $\{-t+2, -t+4, \dots, t\}$. If we add $(t-1)$ to the sum of the terms, the possible values of the sum all become positive. If we set that $DNF = \text{sign}[(\text{the sum of the terms}) + (t-1)]$, then when DNF is *true*, DNF has a value of $+1$.

Following the above analysis, it is not hard to see that we can assume that $DNF = \text{sign}[(\text{the sum of the terms}) + (t-1)]$. This is the reason why we developed the algorithm to find the threshold function hypothesis (the variable of the threshold function is a sum of some parity functions).

3.1.3 How the proposed algorithm works

Let the monotone DNF function $f : \{0, 1\}^n \rightarrow \{+1, -1\}$ be the function which we are trying to learn. f is also called the *target function*. For example, $f = x_1x_2x_3$ can be a monotone DNF function which we are trying to learn. We are allowed to see the whole truth table of the target function.

For example, if the target function is $f = x_1x_2x_3$, the truth table is as the following:

index	bitvector($x_3x_2x_1$)	x_3	x_2	x_1	f-value
0	000	0	0	0	-1
1	001	0	0	1	-1
2	010	0	1	0	-1
3	011	0	1	1	-1
4	100	1	0	0	-1
5	101	1	0	1	-1
6	110	1	1	0	-1
7	111	1	1	1	1

Our goal is to learn a threshold function $T(\vec{x})$ from the truth table such that this threshold function $T(\vec{x})$ approximates $f(\vec{x})$ with a high probability over all the examples in the truth table. It can be expressed like this:

$$Pr[f(\vec{x}) \neq T(\vec{x})] \leq \epsilon$$

in which

$$T(\vec{x}) = \text{sign}[F(\vec{x})]$$

and

$$F(\vec{x}) = \sum_{\vec{a} \in S} \hat{F}(\vec{a}) \chi_{\vec{a}}(\vec{x})$$

where S is a set of n -bit vectors and for each n -bit vector $\vec{a} \in S$ there is one integral coefficient $\hat{F}(\vec{a})$.

For example, by running the program we get the following threshold function for the target function $f = x_1x_2x_3$,

$$T(\vec{x}) = \text{sign}[1\chi_{000} - 2\chi_{001} - 2\chi_{010} + 2\chi_{011} - 2\chi_{100} + 2\chi_{101} + 2\chi_{110} - 2\chi_{111}]$$

At first, before we start explaining the algorithm, we may need to explain the method measuring how well a threshold function approximates the target function $f(\vec{x})$. For easier understanding, suppose that somehow we have found a perfect $F(\vec{x})$ — that is, have found a set S and weights $\{\hat{F}(\vec{a})\}_{\vec{a} \in S}$ — such that for all n -bit vectors \vec{x} , $f(\vec{x}) = \text{sign}[F(\vec{x})]$.

Then we will calculate the expected value of $|F(\vec{x})|$ over all n -bit vectors \vec{x} . Because $f(\vec{x}) = \text{sign}[F(\vec{x})]$, we have

$$E_{\vec{x}}[|F(\vec{x})|] = E_{\vec{x}}[f(\vec{x})F(\vec{x})]$$

By a Fourier theorem called Parseval's Identity, it follows that

$$E_{\vec{x}}[f(\vec{x})F(\vec{x})] = \sum_{\vec{a} \in \{0,1\}^n} \hat{f}(\vec{a})\hat{F}(\vec{a})$$

And because $\hat{F}(\vec{a})=0$ for all $\vec{a} \notin S$,

$$\sum_{\vec{a} \in \{0,1\}^n} \hat{f}(\vec{a})\hat{F}(\vec{a}) = \sum_{\vec{a} \in S} \hat{f}(\vec{a})\hat{F}(\vec{a})$$

Then finally we reach a result for the perfect threshold function $F(\vec{x})$:

$$E_{\vec{x}}[|F(\vec{x})|] = \sum_{\vec{a} \in S} \hat{f}(\vec{a})\hat{F}(\vec{a})$$

On the other hand, if we have a function F' such that there are many \vec{x}' s such that $f(\vec{x}') \neq \text{sign}(F'(\vec{x}'))$, then $f(\vec{x}')F'(\vec{x}')$ will be negative (assuming F' is nonzero, which is guaranteed in the algorithm) for these \vec{x}' s, so

$$E_{\vec{x}}[|F'(\vec{x}')|] > E_{\vec{x}}[f(\vec{x}')F'(\vec{x}')] = \sum_{\vec{a} \in S'} \hat{f}(\vec{a})\hat{F}'(\vec{a})$$

Since we know for any fixed n -bit vector \vec{a} ,

$$\hat{f}(\vec{a}) = \frac{1}{2^n} \sum_{\vec{x}} f(\vec{x})\chi_{\vec{a}}(\vec{x})$$

where the sum is over all possible 2^n n -bit vectors \vec{x} . We can calculate how much larger $E_{\vec{x}}[|F'(\vec{x}')|]$ is than $\sum_{\vec{a} \in S'} \hat{f}(\vec{a})\hat{F}'(\vec{a})$. The smaller the difference, the better the threshold function F approximates the target function f . Our goal is to find an F for which this difference is very small.

Here is how the proposed algorithm works:

Step 1. The initialized set S_0 is $\{\vec{0}\}$ and $\hat{F}(\vec{0})$ is 1. Then the function is:

$$F_0 = 1$$

Then we check over the truth table if

$$Pr[f(\vec{x}) \neq \text{sign}[F(\vec{x})] \leq \epsilon.$$

If it is, the algorithm ends here and the best hypothesis is

$$T[\vec{x}] = \text{sign}[F(\vec{x})] = 1;$$

Otherwise, the algorithm will continue to execute next step.

Step 2. Suppose currently we have a function F_{i-1} (The initial value of i is 1), and we would like to modify it to obtain a new function F_i that is closer to the F we are seeking. We will take the following steps:

Step 2.a Create a set N which includes all of the “neighbors” of S_{i-1} . S_{i-1} is the set of parity functions used to define F_{i-1} . (An n -bit vector \vec{b} is a neighbor of an n -bit vector \vec{a} if \vec{a} and \vec{b} differ in only one bit. The *Immediate down neighbor* of \vec{a} is a neighbor of \vec{a} in which the number of 1's is just one fewer than the number of 1's in \vec{a} .) Then for every vector \vec{a} , check if all of its immediate down neighbors are already included in the set S_{i-1} : if so, keep it in the set N ; otherwise delete it from the set N .

Step 2.b For each n -bit vector \vec{a}_j in the union of N and S_{i-1} , create a new function F_{i-1}^j by starting with a copy of F_{i-1} and modifying this copy as following:

Step 2.b.1 If $\vec{a}_j \in S_{i-1}$, if \vec{a}_j is “odd”, subtract 2 from the weight $\hat{F}_{i-1}^j(\vec{a}_j)$; if \vec{a}_j is “even”, add 2 to the weight $\hat{F}_{i-1}^j(\vec{a}_j)$. Then to guarantee

$$\forall a' \text{ such that } a \subseteq a', |\hat{F}(\vec{a})| \geq |\hat{F}(\vec{a}')|,$$

we need to check if all the absolute values of the weights of all the down neighbors of \vec{a}_j are greater than or equal to the absolute value of the newly increased weight of \vec{a}_j . If they are, add F_{i-1}^j to the pool of the potential hypotheses; if not, F_{i-1}^j is not a potential hypothesis.

Step 2.b.2 If $\vec{a}_j \notin S_{i-1}$ but $\vec{a}_j \in N$, add \vec{a}_j to S_{i-1}^j and set the value of $\hat{F}_{i-1}^j(\vec{a}_j)$ to 2 if \vec{a}_j is “even”; otherwise if \vec{a}_j is “odd”, set the value of $\hat{F}_{i-1}^j(\vec{a}_j)$ to -2 .

Step 2.c Define F_i to be the function F_{i-1}^j which minimizes the difference between $E_x[|F_{i-1}^j|]$ and $\sum_{\vec{a} \in S_{i-1}^j} \hat{f}(\vec{a}) \hat{F}_{i-1}^j(\vec{a})$.

Step 2.d Check over the truth table if

$$\Pr[f(\vec{x}) \neq \text{sign}[F_i(\vec{x})] \leq \epsilon.$$

If it is, then the algorithm ends here; otherwise, continue to *Step 3*.

Step 3. By repeating the procedure in *Step 2*, we will finally approach a threshold function F which satisfies the required accuracy.

3.2 Performed Tests and Results

3.2.1 Some Explanation for the Performed Tests

We will explain some concepts before we talk about the test results:

1. When the algorithm is tested, for most of the cases the ϵ value is set to be 0.05 so that the final best hypothesis $F(\vec{x})$ will satisfy

$$Pr[f(\vec{x}) \neq \text{sign}[F(\vec{x})] \leq 0.05.$$

We also set ϵ to be 0.01 for some cases to test if the algorithm will find a more accurate hypothesis. The result is that the algorithm performs well to find the more accurate hypothesis for several cases.

2. The *minimum error* is the minimum value of $Pr[f(\vec{x}) \neq \text{sign}[F(\vec{x})]$ so far when the algorithm is running;
3. The difference value is the difference between $E_x[|F_{i-1}^j|]$ and $\sum_{\vec{a} \in S_{i-1}^j} \hat{f}(\vec{a}) \hat{F}_{i-1}^j(\vec{a})$;
4. The number of loops is the number of times of the *Step 2* has been run.
5. To get valuable test results, we need to choose the test cases. We prefer to choose the target function with a probability of about 0.5 to be +1 and a probability of about 0.5 to be -1. If a target function satisfies the probabilities requirements, it is *balanced*; otherwise, it is *biased*. Suppose the variable β is the probability that the target function yields -1, then the more skewed the target function, the larger the β value is from 0.5.

When the algorithm is running, if the target function is *biased*, for example, if it has a β value of 0.8, then when modifying the hypothesis, it will be easier to achieve the *minimum error* 0.05. But target function which is *balanced* or almost *balanced* is hard to learn. When the target function satisfies this requirement, it may need more time to learn. So if the algorithm works on the more *balanced* functions, the algorithm is more successful.

3.2.2 How to Analyze the Results

When analyzing the results, there are the following things that we need to check:

- Each parity in the hypothesis should be a subset of the terms in the target function. If some parity is not a subset of any term in the target function, this will be a sign to break the algorithm; otherwise, it is working as we expect.
- Check if the error values keep decreasing. The error values may bounce up and down. But the minimum error values are expected to be decreasing. When the minimum error is smaller than ϵ , the algorithm stops and finds the final best hypothesis.

If the result of some case satisfies the above requirements, we say that the algorithm *performs well* on this case.

3.2.3 Performed Tests and Results with Different Cases

1. The following cases do not necessarily have a β value close to 0.5. They are only some simple cases which we used to test the algorithm. They are listed here for reference purpose.

- $f = x_1x_2x_3$
- $f = x_1x_2 + x_3$
- $f = x_1x_2 + x_3x_4$
- $f = x_1x_2 + x_3x_5 + x_4x_1 + x_2x_3$
- $f = x_1x_2x_3x_4 + x_5x_6x_7x_8$
- $f = x_1x_2x_3 + x_4x_5x_6 + x_7x_8x_9$
- $f = x_1x_2x_3 + x_4x_5x_6 + x_7x_8x_9 + x_{10}x_{11}x_{12}$
- $f = x_1x_2x_3 + x_4x_5x_6 + x_7x_8x_9 + x_{10}x_{11}x_{12} + x_{13}x_{14}x_{15}$
- $f = x_1x_2x_3x_4 + x_6x_5x_7x_8 + x_9x_{10}x_{11}x_{12} + x_{13}x_{14}x_{15}x_{16}$
- $f = x_{15}x_2x_3x_4 + x_6x_5x_7x_8 + x_9x_{10}x_{11}x_{12} + x_{13}x_{14}x_{15}x_{16}$
- $f = x_1x_2x_3x_4 + x_5x_6x_7x_8 + x_9x_{10}x_{11}x_{12} + x_{13}x_{14}x_{15} + x_{16}x_{17}x_{18}$
- $f = x_1x_2x_3x_4 + x_5x_6x_7x_8 + x_9x_{10}x_{11}x_{12} + x_{13}x_{14}x_{15} + x_{16}x_{17}x_{18}x_{19}$
- $f = x_1x_2x_3x_4 + x_5x_6x_7x_8 + x_9x_{10}x_{11}x_{12} + x_{13}x_{14}x_{15}x_{16} + x_{17}x_{18}x_{19}x_{20}$

For the case $f = x_1x_2x_3x_4 + x_6x_5x_7x_8 + x_9x_{10}x_{11}x_{12} + x_{13}x_{14}x_{15}x_{16}$, the algorithm found the best hypothesis with *error* 0 as the following:

$$\begin{aligned}
F = & 1\chi_{0000000000000000} - 2\chi_{0000000000000001} - 2\chi_{0000000000000010} + \\
& 2\chi_{0000000000000011} - 2\chi_{0000000000000100} + 2\chi_{0000000000000101} + 2\chi_{0000000000000110} - \\
& 2\chi_{0000000000000111} - 2\chi_{0000000000001000} + 2\chi_{0000000000001001} + 2\chi_{0000000000001010} - \\
& 2\chi_{0000000000001011} + 2\chi_{0000000000001100} - 2\chi_{0000000000001101} - 2\chi_{0000000000001110} + \\
& 2\chi_{0000000000001111} - 2\chi_{0000000000010000} - 2\chi_{0000000000010000} - 2\chi_{0000000000010000} - \\
& 2\chi_{0000000001000000} - 2\chi_{0000000100000000} - 2\chi_{0000001000000000} - 2\chi_{0000010000000000} - \\
& 2\chi_{0000100000000000} - 2\chi_{0001000000000000} - 2\chi_{0010000000000000} - 2\chi_{0100000000000000} - \\
& 2\chi_{1000000000000000} + 2\chi_{00000000000110000} + 2\chi_{00000000001010000} + 2\chi_{00000000010010000} + \\
& 2\chi_{0000000001100000} - 2\chi_{0000000001110000} + 2\chi_{0000000010100000} - 2\chi_{0000000010110000} + \\
& 2\chi_{0000000011000000} - 2\chi_{0000000011010000} - 2\chi_{0000000011100000} + 2\chi_{0000000011110000} + \\
& 2\chi_{0000000110000000} + 2\chi_{0000001010000000} + 2\chi_{0000100100000000} + 2\chi_{0000011000000000} + \\
& 2\chi_{0000101000000000} - 2\chi_{0000011100000000} - 2\chi_{0000101100000000} + 2\chi_{0000110000000000} - \\
& 2\chi_{0000110100000000} - 2\chi_{0000111000000000} + 2\chi_{0000111100000000}
\end{aligned}$$

2. The case is the read-once monotone DNF, which has u terms and each term with $\log(u)$ distinct variables. For example, when $u = 4$, there is a read-once monotone DNF as

$$x_1x_2 + x_3x_4 + x_5x_7 + x_6x_8$$

This case finished with the expected hypothesis.

3. The random monotone DNF. This case of monotone DNF will be generated by the following method: Every variable of each term is randomly drawn from all the variables without replacements. For example, at first we draw x_1 and put it into the first term. The second variable will be drawn randomly from all the variables except x_1 , for example x_3 . Then the third variable will be drawn randomly from all the other variables except x_1 and x_3 . After the first term is drawn, the second term will be drawn by the same method as we draw the first term: the first variable of the second term is randomly drawn from all the variables, so on and so forth. Here is an example of the case of monotone DNF,

$$x_1x_3x_7 + x_2x_5x_{12} + x_4x_8x_9 + \dots$$

Drs. Jackson and Servedio recently gave the proof that the n -term random monotone DNF with $\log_2 n$ literals in each term is learnable in

polynomial time from uniform random examples with high probability, so this is an interesting case with which to test the algorithm. Furthermore, we need to test for the case of the n^2 -term random monotone DNF with $2\log(n)$ literals in each term because Jackson and Servedio's result does not extend to this case.

We have tried to test 2 cases with $n = 8$ and $n = 16$:

- The random monotone DNF with 8 variables, 64 terms and 6 literals in each term. This case was finished with expected results. This result is really encouraging since this random monotone DNF is really a big function. The following is the target function we have tried to learn.

$$\begin{aligned}
f = & x_1x_6x_3x_5x_7x_2 + x_8x_6x_7x_4x_1x_5 + x_4x_2x_8x_6x_5x_7 + x_6x_4x_5x_3x_7x_1 + \\
& x_4x_1x_2x_3x_5x_8 + x_8x_3x_5x_6x_7x_2 + x_1x_3x_6x_5x_2x_8 + x_1x_4x_8x_6x_7x_3 + \\
& x_4x_7x_5x_2x_6x_1 + x_3x_2x_6x_1x_8x_5 + x_8x_2x_3x_5x_1x_4 + x_2x_8x_7x_4x_3x_1 + \\
& x_4x_2x_6x_5x_8x_1 + x_6x_1x_8x_7x_2x_3 + x_1x_4x_6x_3x_8x_7 + x_2x_1x_4x_3x_7x_5 + \\
& x_8x_3x_5x_2x_7x_4 + x_1x_5x_3x_4x_7x_8 + x_3x_1x_4x_5x_8x_2 + x_1x_3x_4x_2x_7x_5 + \\
& x_3x_6x_5x_4x_8x_2 + x_3x_5x_2x_6x_1x_7 + x_3x_7x_5x_2x_6x_8 + x_5x_6x_7x_3x_2x_8 + \\
& x_2x_6x_3x_5x_8x_1 + x_1x_4x_5x_2x_3x_8 + x_1x_7x_4x_2x_6x_3 + x_2x_6x_1x_4x_8x_7 + \\
& x_8x_7x_3x_5x_4x_1 + x_5x_4x_2x_1x_3x_8 + x_1x_8x_6x_7x_2x_5 + x_6x_3x_2x_5x_8x_1 + \\
& x_7x_1x_3x_8x_6x_5 + x_8x_3x_6x_5x_2x_1 + x_3x_6x_7x_8x_4x_1 + x_7x_5x_1x_2x_3x_4 + \\
& x_8x_3x_1x_5x_6x_4 + x_8x_5x_1x_4x_7x_3 + x_5x_7x_2x_4x_6x_8 + x_3x_4x_5x_8x_2x_7 + \\
& x_2x_4x_6x_3x_5x_7 + x_5x_6x_8x_3x_4x_7 + x_4x_7x_2x_8x_3x_5 + x_5x_8x_1x_2x_4x_6 + \\
& x_4x_8x_5x_3x_6x_1 + x_6x_5x_2x_8x_3x_4 + x_8x_5x_2x_3x_4x_7 + x_6x_7x_3x_4x_5x_1 + \\
& x_5x_4x_1x_2x_7x_8 + x_8x_5x_1x_6x_3x_2 + x_5x_6x_3x_1x_4x_8 + x_4x_6x_1x_3x_5x_7 + \\
& x_4x_1x_6x_2x_5x_8 + x_8x_4x_1x_6x_5x_3 + x_8x_3x_4x_5x_7x_6 + x_2x_4x_8x_3x_1x_5 + \\
& x_4x_8x_7x_2x_1x_3 + x_4x_2x_3x_8x_6x_7 + x_3x_7x_5x_4x_2x_6 + x_3x_2x_6x_1x_7x_4 + \\
& x_3x_7x_4x_1x_5x_6 + x_2x_6x_7x_5x_8x_4 + x_4x_5x_3x_2x_8x_6 + x_6x_8x_2x_7x_5x_3
\end{aligned}$$

The final best hypothesis approximates the target function with an error value 0.046875. The hypothesis is:

$$\begin{aligned}
F = & 1\chi_{00000000} - 6\chi_{01000000} - 6\chi_{00001000} + 6\chi_{01001000} - 6\chi_{00010000} + \\
& 6\chi_{00011000} + 6\chi_{01010000} - 6\chi_{01011000} - 6\chi_{00100000} + 6\chi_{01100000} + \\
& 6\chi_{00110000} - 6\chi_{01110000} + 4\chi_{00101000} - 4\chi_{01101000} - 4\chi_{00111000} + \\
& 4\chi_{01111000} - 6\chi_{10000000} - 4\chi_{00000001} - 4\chi_{00000010} - 4\chi_{00000100} + \\
& 4\chi_{01000001} + 4\chi_{01000010} + 4\chi_{01000100} + 6\chi_{11000000} + 6\chi_{10001000} -
\end{aligned}$$

$$\begin{aligned}
& 4\chi_{11001000} + 2\chi_{00001001} - 2\chi_{01001001} + 2\chi_{00001010} - 2\chi_{01001010} + \\
& 2\chi_{00001100} - 2\chi_{01001100} + 6\chi_{10010000} - 4\chi_{10011000} - 6\chi_{11010000} + \\
& 4\chi_{11011000} + 6\chi_{10100000} - 4\chi_{11100000} - 4\chi_{10101000} + 4\chi_{11101000} - \\
& 4\chi_{10110000} + 2\chi_{10111000} + 4\chi_{11110000} - 2\chi_{11111000} + 2\chi_{00010001} + \\
& 2\chi_{00100001} + 2\chi_{10000001} + 2\chi_{00010010} + 2\chi_{00100010} + 2\chi_{10000010} + \\
& 2\chi_{00010100} + 2\chi_{00100100} + 2\chi_{10000100} - 2\chi_{01100001} - 2\chi_{01100010} - \\
& 2\chi_{01100100} - 2\chi_{01010001} - 2\chi_{01010010} - 2\chi_{01010100} - 2\chi_{11000001} - \\
& 2\chi_{11000010} - 2\chi_{11000100}
\end{aligned}$$

- The random monotone DNF with 16 variables, 256 terms and 8 literals in each term. This case was not finished. The result so far we have is that: the minimum error is 0.111252 in Loop 2349 with a difference value 6.38239. Before that, the minimum error stayed 0.112366 from Loop 1363 to Loop 2238 and then it was changed to 0.112183, then to 0.111252. Since the difference value is still small and the minimum value is still getting smaller, it might be reasonable to say that for this case it may be finished with expected results. (This case had run for several days and then was stopped because the computer was restarted by other people. We will try to test this case again if we have time.)
4. The monotone DNF in which there is significant sharing of variables between terms. We have tested a case like this: Whenever x_1 is selected for a term, x_2 and x_3 will also be in the term with probability 0.5 each. That is, whenever x_1 is selected for a term, there is a probability 0.25 that both x_2 and x_3 will be in the term, a probability 0.25 that only x_2 will be in the term, a probability 0.25 that only x_3 will be in the term and a probability 0.25 that neither x_2 nor x_3 will be in the term. This means that x_2 and x_3 have the equal probability appearing in the function. This might confuse the algorithm when facing the problem of whether x_2 or x_3 should be chosen.

We tested the case with 2 examples:

- (a) One example has 8 variables, 64 terms and 6 literals in each term;
- (b) The other example has 10 variables, 64 terms and 6 literals in each term.

These two cases both performed well and got the expected results.

5. The monotone DNF that we can get by making an AND-OR tree of depth $2\log\log(n)$, i.e. a binary tree where odd layers all have AND and

even layers all have OR. This case was suggested by Dr. Rocco Servedio. This function has $\log^2(n)$ many relevant variables, and has both monotone DNFs and CNFs of polynomial size (which is polynomial respect to n). The function is actually a polynomial size monotone CNF.

We tried to test a case like this:

$$\begin{aligned} &(((x_1 \wedge x_2) \vee (x_3 \wedge x_4)) \wedge ((x_5 \wedge x_6) \vee (x_7 \wedge x_8))) \\ &\vee(((x_9 \wedge x_{10}) \vee (x_{11} \wedge x_{12})) \wedge ((x_{13} \wedge x_{14}) \vee (x_{15} \wedge x_{16}))) \end{aligned}$$

Which can be also expressed as:

$$(x_1x_2 + x_3x_4)(x_5x_6 + x_7x_8) + (x_9x_{10} + x_{11}x_{12})(x_{13}x_{14} + x_{15}x_{16})$$

If it is converted to a Monotone DNF function, it becomes:

$$\begin{aligned} &x_1x_2x_5x_6 + x_1x_2x_7x_8 + x_3x_4x_5x_6 + x_3x_4x_7x_8 + \\ &+ x_9x_{10}x_{13}x_{14} + x_9x_{10}x_{15}x_{16} + x_{11}x_{12}x_{13}x_{14} + x_{11}x_{12}x_{15}x_{16} \end{aligned}$$

For this case the algorithm didn't finish with the expected results. The result is: the minimum error was 0.134216, which began from Loop 135. And this minimum error stayed 0.134216 until Loop 2681. The difference values are not getting large, staying around 1 and 2. For Loop 2681, the ratio is 1.28302. Since it required so many loops without reaching a smaller error value and some parity functions are not subsets of the terms in the target function, it appears that the algorithm will not work out for this case.

Since a DNF expression can also be represented by a CNF, it is not guaranteed that the algorithm will find the terms in the DNF instead of the clauses in the CNF. It was expected that this case of monotone DNF may be hard to learn since the algorithm may find the parities which are the clauses in the CNF.

6. The monotone DNF with dependent terms. Assume there is a variable $\alpha(0 \leq \alpha \leq 1)$, which is used to indicate the level of dependency between the terms in the target function. The larger the value of α , the more terms are dependent. The target function is generated in the

following way:

Suppose the target function has t terms, with a total n variables and l literals in each term. Then all the literals in the first term are chosen randomly without replacement (the same way that we used to generate each term in Case 3). Then for the literals in all the other $t - 1$ terms, each literal has the probability α of choosing the literal which appears at the same place in the last term and the probability $1 - \alpha$ of choosing randomly from all the remaining variables except the variables that have already appeared in this term.

For example, if we want to generate a function with 8 variables, 8 terms, and 3 literals in each term with $\alpha = 0.3$, the first term can be generated as $x_1x_7x_4$. Then the first variable in the second term will have probability 0.3 to be x_1 again and probability 0.7 to be chosen from all the remaining literals, $x_2, x_3, x_4, x_5, x_6, x_7$, and x_8 . Each literal has equal probability $\frac{1}{7}$ to be chosen. Suppose we end up choosing x_2 here. Then for the second literal in the second term, it will have probability 0.3 to be x_7 again and probability 0.7 to be chosen from all the remaining literals except x_2 and x_7 , that is chosen randomly from x_1, x_3, x_4, x_5, x_6 , and x_8 . Each literal has equal probability $\frac{1}{6}$ to be chosen. Suppose we happen to still choose x_7 to be the second literal in the second term. Then until now we have x_2 and x_7 to be the first and the second literal respectively in the second term. We still need the third literal to be generated. Then for the third literal in the second term, it will have probability 0.3 to be x_4 again and probability 0.7 to be chosen from all the remaining literals except x_2, x_7 and x_4 , that is chosen randomly from x_1, x_3, x_5, x_6 , and x_8 . Each literal has equal probability $\frac{1}{5}$ to be chosen. Suppose we get x_5 to be the third literal in the second term. Then we get the second term as $x_2x_7x_5$. Then starting from the second term, we may need to find the third term in the same way. For example, for the first literal in the third term, it has probability 0.3 to be x_2 again and probability 0.7 to be chosen from all the remaining literals except x_2 and x_7 , that is, chosen randomly from x_1, x_3, x_4, x_5, x_6 , and x_8 . Then by the similar way we will generate the other literals in the second term and the other terms.

For this case, the value of α can be changed continuously from 0 to 1. If it is 0, it means that all the terms are generated independently of

each other; if it is a value greater than 0 but smaller than 1, it means that the terms in the target function are dependent with a coefficient α . The larger the α is, the more dependent the terms in the target function are on each other.

To get valuable test results, we tested the algorithm with some more balanced target functions. As we said earlier, we prefer to choose the target function with a probability of about 0.5 to be +1 and a probability of about 0.5 to be -1. If a target function satisfies with the probabilities requirements, it is *balanced*; otherwise, it is *biased*. Suppose the variable β is the probability of the target function to be -1, then the more skewed the target function, the larger the β value is away from 0.5.

Since we may want to have α continuously changing in the interval from 0 to 1, we hope that when α is 0, β is about 0.45, which is close to 0.5 but a little less than it; when α is 0.95, β is around 0.65.

We tried to vary the number of terms, the number of literals in each term and the number of variables in the target function to get valuable test cases. Then we found a series of test cases: each function has 16 variables, 20 terms and 4 literals in each term. The cases are:

- (a) Case 1. When $\alpha = 0$, $\beta = 0.465515$;
- (b) Case 2. When $\alpha = 0.25$, $\beta = 0.462097$;
- (c) Case 3. When $\alpha = 0.50$, $\beta = 0.516663$;
- (d) Case 4. When $\alpha = 0.75$, $\beta = 0.648193$;
- (e) Case 5. When $\alpha = 0.95$, $\beta = 0.875$ (Notice that the β value is 0.875 instead of 0.65. It is because that in fact it is hard to find the idea series to satisfy: when α is 0.95, β is around 0.65. This is the best series of test cases that I have found to test.)

The algorithm performs very well in finding the hypothesis on Case 1, Case 4 and Case 5 with $\epsilon = 0.05$. But it seems that it may fail for Case 2 and Case 3. The following is the data we got in the results:

- (a) For alpha=0.25, from Loop 82 to Loop 5451, the minimum error stays 0.269104; The difference values keep getting larger and become 409.171 in Loop 5451.

- (b) For $\alpha=0.5$, from Loop 51 to Loop 7046, the minimum error stays 0.209656; The difference values keep getting larger and become 360.967 in Loop 7046.

For these two cases, the difference values are getting so large without reaching a smaller error value. Since the goal of our algorithm is to find an F for which this difference is nearly 0, the difference values getting so large is not reasonable: If it they are getting so large, then it is hard to guarantee that the selected hypothesis is getting closer to the perfect hypothesis we are seeking. So we begin to think that maybe the current algorithm does not work correctly for some of the monotone DNF functions with dependent terms. We will talk later in the section *Future Work* about some modifications to the current algorithm to make it work for these cases.

Another series of the similar target functions are: each function has 16 variables, 256 terms and 6 literals in each term. The cases are:

- (a) Case 1. When $\alpha = 0$, $\beta = 0.45665$;
- (b) Case 2. When $\alpha = 0.1$, $\beta = 0.454147$;
- (c) Case 3. When $\alpha = 0.2$, $\beta = 0.457825$;
- (d) Case 4. When $\alpha = 0.3$, $\beta = 0.45816$;
- (e) Case 5. When $\alpha = 0.4$, $\beta = 0.460587$;
- (f) Case 6. When $\alpha = 0.5$, $\beta = 0.460587$;
- (g) Case 7. When $\alpha = 0.6$, $\beta = 0.472488$;
- (h) Case 8. When $\alpha = 0.7$, $\beta = 0.497726$;
- (i) Case 9. When $\alpha = 0.8$, $\beta = 0.520004$;
- (j) Case 10. When $\alpha = 0.85$, $\beta = 0.551468$;
- (k) Case 11. When $\alpha = 0.9$, $\beta = 0.62793$;
- (l) Case 12. When $\alpha = 0.95$, $\beta = 0.719345$.

The results and some analysis of the results are explained in the following:

- (a) Case 1. When $\alpha = 0$, $\beta = 0.45665$; The algorithm reaches a minimum error 0.144882 in Loop 33 and keeps this minimum error until Loop 5185. From the result file, we can see that there are no new parities added in the hypothesis after Loop 47. Without

adding in new parities, the weights of the old parities are added up. In Loop 5185, the selected hypothesis is:

$$\begin{aligned}
F = & 651\chi_{0000000000000000} - 316\chi_{0000000000001000} - 314\chi_{0100000000000000} \\
& + 2\chi_{0100000000001000} - 314\chi_{0000000100000000} + 2\chi_{0000000100001000} \\
& + 2\chi_{0100000100000000} - 2\chi_{0100000100001000} - 326\chi_{0000001000000000} \\
& - 650\chi_{0000000010000000} + 326\chi_{0000001010000000} - 326\chi_{0000100000000000} \\
& - 326\chi_{0000010000000000} - 326\chi_{0000000000000010} - 324\chi_{0000000000000100} \\
& - 324\chi_{0000000000000001} - 16\chi_{0001000000000000} - 324\chi_{0000000000100000} \\
& - 324\chi_{0000000000010000} - 324\chi_{0010000000000000} - 322\chi_{1000000000000000} \\
& - 322\chi_{0000000001000000} + 2\chi_{0101000000000000} + 2\chi_{0001000000001000} \\
& + 2\chi_{0001000100000000} - 2\chi_{0101000100000000} - 2\chi_{0101000000001000} \\
& - 2\chi_{0001000100001000} + 2\chi_{0101000100001000} + 312\chi_{0000000110000000} \\
& + 324\chi_{0000000010000001} + 324\chi_{0010000010000000} + 324\chi_{0000000010010000} \\
& + 312\chi_{0100000010000000} + 14\chi_{0001000010000000} + 314\chi_{0000000010001000} \\
& + 322\chi_{1000000010000000} + 326\chi_{0000000010000010} + 326\chi_{0000100010000000} \\
& + 326\chi_{0000010010000000} + 324\chi_{0000000010100000} + 322\chi_{0000000011000000} \\
& + 324\chi_{0000000010000100}
\end{aligned}$$

The difference value for this selected hypothesis is 99.7091.

For this case, since $\alpha = 0$, the terms are supposed to be independent of each other. That is, it is a random monotone DNF function. But given the above results, the algorithm may fail to find the expected hypothesis. As for now, we suppose that although we thought it was random Monotone DNF, the terms in the function might be somehow dependent each other. We still need to do some analysis on this case.

- (b) Case 2. When $\alpha = 0.1$, $\beta = 0.454147$; The minimum error is 0.129028 until Loop 9409. Since some files were deleted, I can not find in which loop this minimum error value was reached. It was known from my notes that it was already reached by Loop 6866. In Loop 8117, the selected hypothesis has the difference value 38.94 and in Loop 9409, it has the difference value 44.5602. We also know that after Loop 320, there is no new parity added to the hypothesis.
- (c) Case 3. When $\alpha = 0.2$, $\beta = 0.457825$. In Loop 632, this minimum error value 0.129044 was reached. Then this value stayed until Loop 7627. In Loop 6675, the selected hypothesis has the difference value 31.3 and in Loop 7627, it has the difference value

35.2173. We also know that after Loop 808, there is no new parity added to the hypothesis.

- (d) Case 4. When $\alpha = 0.3$, $\beta = 0.45816$. The minimum error is 0.12883 until Loop 6078. Since some files were deleted, I can not find in which loop this minimum error value was reached. It was known from my notes that it was already reached by Loop 4543. In Loop 5370, the selected hypothesis has the difference value 36.8 and in Loop 6078, it has the difference value 41.387. We also know that after Loop 5928, there is no new parity added to the hypothesis.
- (e) Case 5. When $\alpha = 0.4$, $\beta = 0.460587$. The minimum error is 0.13089 until Loop 5068. Since some files were deleted, I can not find in which loop this minimum error value was reached. It was known from my notes that it was already reached by Loop 2968. In Loop 4130, the selected hypothesis has the difference value 42.3 and in Loop 5068, it has the difference value 52.0493.
- (f) Case 6. When $\alpha = 0.5$, $\beta = 0.460587$. The minimum error 0.129639 was reached in Loop 1976. Then it stayed until Loop 7764. In Loop 6892, the selected hypothesis has the difference value 70 and in Loop 7764, it has the difference value 80.692. We also know that there are new parities added into the hypothesis in Loop 4360, Loop 5577 and Loop 5578.
- (g) Case 7. When $\alpha = 0.6$, $\beta = 0.472488$. The minimum error 0.133835 was reached in Loop 1319. Then it stayed until Loop 9134. In Loop 8075, the selected hypothesis has the difference value 86.93 and in Loop 9134, it has the difference value 98.3617.
- (h) Case 8. When $\alpha = 0.7$, $\beta = 0.497726$. The minimum error 0.137619 was reached in Loop 1386. They it stayed until Loop 4395. In Loop 2458, the selected hypothesis has the difference value 35.7417 and in Loop 4395, it has the difference value 63.94. We also know that after Loop 988, there is no new parity added to the hypothesis.
- (i) Case 9. When $\alpha = 0.8$, $\beta = 0.520004$. The minimum error 0.140915 was reached in Loop 3457. Then it stayed until Loop 8728. In Loop 7776, the selected hypothesis has the difference value 88.6011 and in Loop 8728, it has the difference value 99.9995. We also know that after Loop 3414, there is no new parity added to the hypothesis. Before Loop 3414, the last few

added parities were added in Loop 2980, Loop 3413, Loop 3412, Loop 3411, Loop 3348 and etc.

- (j) Case 10. When $\alpha = 0.85$, $\beta = 0.551468$. The minimum error 0.137848 was reached in Loop 1803. Then it stayed until Loop 8201. In Loop 8201, the selected hypothesis has the difference value 99.9955. We also know that after Loop 2046, there is no new parity added to the hypothesis. Before Loop 2046, the last few added parities were added in Loop 2045, Loop 3413, Loop 3412, Loop 3411, Loop 3348 and etc.
- (k) Case 11. When $\alpha = 0.9$, $\beta = 0.62793$. The minimum error 0.128036 was reached in Loop 1630. Then it stayed until Loop 5510. In Loop 4907, the selected hypothesis has the difference value 40.887 and in Loop 5510, it has the difference value 45.8142. We also know that after Loop 3310, there is no new parity added to the hypothesis. Before Loop 3310, the last few added parities were added in Loop 2045, 2044, 2043, 1830, 1829, 1828, 1827, 1518, 1517, 1516, 1515, 1108, 1107, 1106, 1105, 814, 813, 708, 707, 704, 703, 702, 701, 700, 699, 698, 697, 696, 695, 640 and etc.
- (l) Case 12. When $\alpha = 0.95$, $\beta = 0.719345$. The algorithm works well on this case. It reaches the minimum error 0.053772 in Loop 2357 with the difference value 2.09869.

The results of the above series are similar to the previous series of cases, so we have another example to make us think that maybe the current algorithm does not work correctly for some of the monotone DNF functions with dependent terms.

3.3 Some Attempted Modifications to the Algorithm and the Results

We once tried to use the ratio values instead of the difference values to decide which hypothesis is the temporary best hypothesis from the potential hypotheses pool. The ratio value is defined as the following:

$$ratio = \frac{E_x[|F_{i-1}^j|] - \sum_{\vec{a} \in S_{i-1}^j} \hat{f}(\vec{a}) \hat{F}_{i-1}^j(\vec{a})}{E_x[|F_{i-1}^j|]}.$$

It did not work well to get the expected results, so we switched back to the difference values.

3.4 Future Work

- We start the algorithm by setting the coefficient of the constant parity $\hat{F}(\vec{0})$ as 1. Although it has been proved by Dr. Jackson that “there is a threshold function with a positive constant coefficient that represents any monotone DNF”, it is only proved that a positive constant coefficient will work. We still do not have a proof why we can set the start point as $\hat{F}(\vec{0}) = 1$. We are trying to use this value to make the algorithm work. However, since we have got some test cases, which did not perform well under the algorithm since the algorithm appears to keep adding up the coefficients of constant parity and some lower degree parities without getting a new hypothesis with the smaller *minimum error*, we think that some changes to the constant parity coefficients may help to make the algorithm work better.
- We may need to change the way how we choose the temporary best hypothesis from the potential hypotheses pool if there are more than one hypothesis minimizing difference value. Our current algorithm chooses the first hypothesis in the ties. We want to try 2 ways in future:
 1. Try to choose randomly from the ties;
 2. Try to choose the hypothesis with smallest weight after the parity is added in the hypothesis or the weight of the current parity is added up.
- We should try more cases, such as another case with dependent terms(x_1 dependent on x_2 and x_3 , x_4 dependent on x_5 and x_6 , \dots in the same target function).
- We should use some different data structures programming the algorithm to make the algorithm faster.

4 Bibliography

1. L.G.Valiant. A theory of the learnable, *Comm. ACM* 27(11)(1984),1134-1142
2. M.Kearns, M.Li, L.Pitt, and L.Valiant. On the learnability of Boolean formulae, *in Proc. 19th Ann. ACM Symp. on Theory of Computing*(1987), 285-295.
- 3.T. Hancock and Y. Mansour. Learning Monotone $k - \mu$ DNF formulas on

- product distributions, in “Proc. 4th Ann. Workshop on Comp. Learning Theory” (1991), 179-183.
4. K.Verbeurgt. Learning sub-classes of monotone DNF on the uniform distribution , in Proc. 9th Conf. on Algorithmic Learning Theory(1998), 385-399
 5. L.Kucera, A.Marchetti-Spaccamela and M.Protassi. On learning monotone DNF formulae under uniform distributions, *Inf. And Comput.* 110(1994), 84-95.
 6. Y.Sakai and A.Maruoka. Learning monotone log-term DNF formulas under the uniform distribution, *Theory Comput. Systems* 33(2000),17-33. A preliminary version appeared in Proc. Seventh Conf. on Comp. Learning Theory(1994), 165-172.
 7. N.Bshouty. Exact learning via the monotone theory. *Information and Computation* 123(1) (1995), 146–153
 8. N.Bshouty and C. Tamon. On the Fourier spectrum of monotone functions, *J. ACM* 43(4) (1996),747–770
 9. R.A.Servedio. On Learning Monotone DNF under Product Distributions.