2005

Towards Optimal Tree Construction of Monotone Functions

Miao Chen

Toward Optimal Tree Construction of Monotone Functions

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

Miao Chen

04/08/2005

Miao Chen

Toward Optimal Tree Construction of Monotone Functions

Master of Science in Computational Mathematics

04/08/2005

APPROVED____
Jeffrey Jackson, Ph.D., Professor of Computer Science

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

APPROVED____
Patrick Juola, Ph.D., Associate Professor of Computer Science

APPROVED____
Kathleen Taylor, Ph.D., Graduate Director and Professor of Mathematics

APPROVED____
Francesco C. Cesareo, Ph.D., Dean
McAnulty College and Graduate School of Liberal Arts

Toward Optimal Tree Construction of Monotone Functions

1. Introduction

A simple way to represent a Boolean function *f* which is assumed to be a mapping from the space $\{0,1\}^n \rightarrow \{-1,1\}$ with the exponent n representing the total number of Boolean variables in the function is through the usage of the disjunctive normal form (DNF), in which a conjunction of disjunctions of Boolean literals (Boolean variables and their negations) are used. For instance, the expression $x_1x_2+x_2x_3+x_4x_5$ is a DNF. While a monotone function *f* is a function for which the result will increase or remain the same if one of its variables is flipped from 0 to 1, a monotone DNF simply contains no negation of a Boolean variable. An expression like $x_1x_2+x_1\bar{x}_2$ is not a monotone DNF but is a monotone function, since although it includes the negation of the Boolean variable x_2, it can be further simplified to the minimized expression x_1.

While the question of whether DNF is PAC learnable (defined later) in the distribution-free setting has remained open during the past few decades, scientists have been trying to solve the reduced problem of monotone decision tree learning from a uniformly distributed sample [1]. Any Boolean function can be represented by a decision tree. For example, a decision tree representation of the above DNF $x_1x_2+x_2x_3+x_4x_5$ is given in Figure 1.

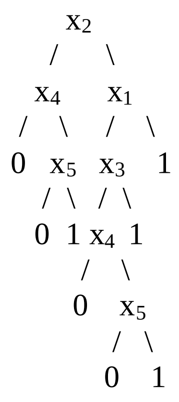

Figure (1) Decision tree representation of $x_1x_2+x_2x_3+x_4x_5$

Since most decision tree representations of a monotone function (MDT) are larger than the DNF representations of the same function (MDNF), i.e., learning a MDT is expected to have less complexity than learning a MDNF and is an interesting problem. A problem related to this is the construction of optimal tree. In this paper, I introduce a series of experiments to test the conjecture that

In a MDT, if a Boolean variable v_b is relevant in only one sub-tree when a Boolean variable v_r is the root, but v_r is relevant in both sub-trees when v_b is the root, then constructing an optimal tree having v_r as its root, produces at least as small as a tree having v_b as its root.

In the above statement, the size of a decision tree is measured using the total number of the leaves. Variables v_b and v_r are any two variables within all the n variables, which satisfy the above conditions. For the purpose of examining the effectiveness of this conjecture, a comparison between optimal trees with the different roots needs to be conducted. The criteria of optimal tree construction will be presented later including the idea of influence-based optimal tree construction first suggested by O'Donnell and Servedio [2]. Notations involved to this point are further explained as below.

Monotone function

A monotone function can be defined more formally in the following way:

Let e_i be a string over $\{0,1\}^n$ with 1 only on the ith position and everywhere else 0; a monotone function shall satisfy the inequality:

$$f(x) \leq f(x \oplus e_i), \text{ for } \forall \text{ i and } \forall \ x,$$

where the x's ith variable x_i must satisfy $x_i=0$. In the above inequality, the symbol $\oplus$ is the XOR operator.

The importance of monotone DNF comes from the key role it plays in computer learning as illustrated in a paper by Michael J. Kearns et.[4] that if the representation class of monotone DNF formulae is efficiently PAC learnable, the representation class of general DNF formulae is also efficiently PAC learnable.

Monotone Boolean decision tree

Any function can be represented by a decision tree. A monotone Boolean decision tree is a binary tree that computes a monotone function. At the beginning, a variable is chosen as the root of the tree, and then a node splitting procedure will continue as a root variable is set to 0 or 1 in each of the iterations. The procedure will stop when no variable is left for dividing and a leaf is generated to represent the expected value of the function.

For a fixed monotone function, its decision tree representation is not unique, and the size of a decision tree depends on the choice of a variable as the root and the consequent roots of all the sub-trees. To make it clear at this point, let us look at the decision trees of the function described by above DNF, $x_1x_2+x_2x_3+x_4x_5$.

As shown in Figure 2, these two trees have different sizes. The tree in (a) has a size of 8, and the tree in (b) has a size of 11. This degeneracy of decision trees for a given monotone function *f* will naturally lead us to find the minimum sized tree for the given *f*.

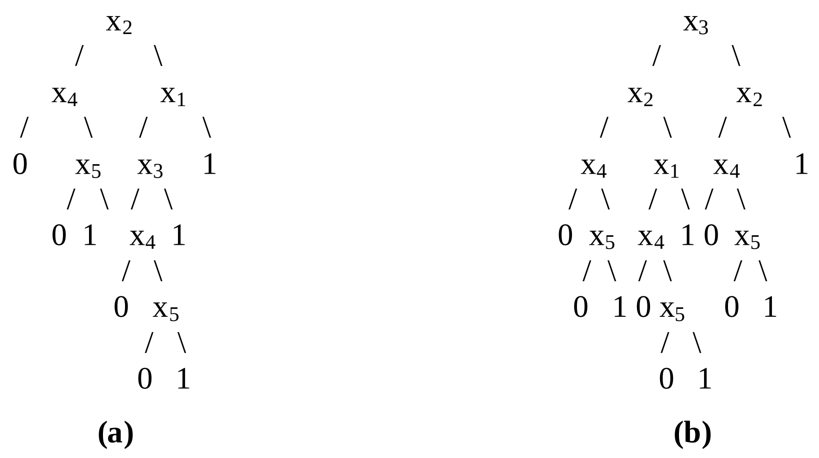

Figure (2) Two different trees for $x_1x_2+x_2x_3+x_4x_5$.

Optimal tree

We noted that the trees considered here are those monotone decision trees that compute monotone functions. An optimal tree is a representation of a tree with size, which equals to the total number of leaves, at least as small as any other representation. A variable is called an optimal root of the tree if the optimal size of the tree can be achieved by choosing this variable as the root.

Influence

Conceptually, the term "influence" introduced here can be treated as it would function in real life. If a variable doesn't provide any positive change to the result, that variable would have zero influence. A mathematical definition of the influence in the uniformly distributed setting is given as:

$$\mathrm{Inf}_i(f) \equiv \Pr_{x\sim U_n}[f(x) \neq f(x \oplus e_i)].$$

This denotes that the influence of the i^{th} variable is equal to the probability of change of function's value when the i^{th} variable is flipped from 0 to 1 or vice versa. For a monotone function, the calculation of influence can be further simplified by the lemma below:

Lemma: Let f:$\{0,1\}^n \rightarrow \{-1,1\}$. In a monotone function, $\mathrm{Inf}_i(f) = -\mathbf{E}[f(x)\cdot \chi_{e_i}] = -\mathbf{E}[f(x)\cdot(-1)^{x_i}]$, where **E** denotes the expectation value with respect to the uniform distribution over x and χ_{e_i} is defined as $\chi_{e_i}(x) \equiv (-1)^{e_i \cdot x}$.

Relevant variable in a tree

A variable v is relevant in a tree if the variable has non-zero influence in the tree.

PAC learnable

A set of Boolean functions are efficiently PAC learnable if there exists an algorithm, which, when given a teaching procedure (a procedure allowing creation of functions from training data) to learn the target function, can output a hypothesis satisfying error $\leq \varepsilon$ with probability at least $1-\delta$ in time polynomial to $1/\varepsilon$, $1/\delta$, n and the minimum size representation of the function.

This famous model of learning is PAC learning, which stands for the Probably Approximately Correct learning. The concept was first introduced by L.G. Valiant [3].

The main effort here is motivated by somewhat different criteria from an actual learning procedure of a monotone function, since the learning program is not fed by a fraction of the data but rather the entire truth table of a specific monotone function. The problem becomes more like an optimization procedure of tree construction as we are trying to find evidence for or against the thesis that optimal-sized tree can be constructed using the idea of influence.

2. Present status of the proposed problem

It has been shown in previous studies that if monotone DNF is PAC learnable with any distribution for a given sample, then this DNF is also PAC learnable, independent of sample distributions; nevertheless, this does not necessarily hold if monotone DNF is PAC learnable with respect to uniform distribution [4]. In 1994, Jackson discovered that a DNF can be efficiently PAC learned under uniform distribution by applying a membership-query algorithm [5]. By assuming a constant accuracy, O'Donnell and Servedio showed that the class of monotone functions under uniform distribution is PAC learnable in time polynomial in the size of decision tree representing the function [2]. If it is not feasible to construct an optimal tree efficiently by feeding the program the entire truth table of the function, then by the Chernoff bound, $\Pr[|\hat{p} - p| \geq \varepsilon] \leq 2e^{-2m\varepsilon^2}$, a number of samples m polynomial in $1/\varepsilon$ is sufficient to ensure (with high probability) that the hypothesis is within an error ε. Therefore, finding an optimal tree representation by perfect data can lead to optimization of the learning algorithm presented by O'Donnell and Servedio.

3. Suggested methodologies

1) Conjectures:

In order to remind the reader of the primary concern raised in the beginning of this proposal, it is repeated here as the primary conjecture.

Primary conjecture:

In a decision tree representation of a monotone function *f*, if a Boolean variable v_b is relevant in only one sub-tree when a Boolean variable v_r is the root, but v_r is relevant in both sub-trees when v_b is the root, then constructing an optimal tree having v_r as its root produces at least as small as a tree having v_b as its root.

Secondary conjecture:

If f is a monotone function and $\mathrm{Inf}_i(f) \geq \mathrm{Inf}_j(f)$ for all $v_j \neq v_i$, then an optimal-sized tree for f can be constructed having v_i as its root.

The algorithm of influence-based optimal tree construction explained below is built upon this conjecture. The two conjectures are related: we prove below that for the situation given in the primary conjecture, $\mathrm{Inf}_r(f) \geq \mathrm{Inf}_b(f)$.

In the secondary conjecture, the inference from the condition is quite liberal since it does not necessarily say that only a variable with the largest influence can be a monotone function's optimal root. Examples clarifying this point are quite easy to find. Let's compare the optimal trees with x_2 and x_3 as the root, respectively, in the monotone function $x_1x_2+x_1x_3x_4+x_2x_3$:

```
        x2                          x3
      /     \                     /     \
    x3       x3                 x2       x2
   /   \    /   \              /   \    /   \
  0    x1  x1    1            0    x1  x1    1
       /  \ / \                    /  \ / \
     0  x4  0  1                  0  1  0 x4
        / \                                / \
       0   1                              0   1
```

Figure (3)

From the above graphs, both optimal trees have 7 leaves and therefore have the same size, but $\mathrm{Inf}(x_2)=5/8$ while $\mathrm{Inf}(x_3)=3/8$.

Lemma: If r and b are two variables in a monotone function f and b is relevant in only one sub-tree when r is the root, but r is relevant in both sub-trees when b is the root, then $\mathrm{Inf}_r(f) \geq \mathrm{Inf}_b(f)$.

Proof: Assume without loss of generality that b is relevant only in the right sub-tree when r is the root. Let $\boldsymbol{a}$ be an assignment such that $f(\boldsymbol{a}\,|b=0,r=1)=0$ and $f(\boldsymbol{a}\,|b=1,r=1)=1$, then $f(\boldsymbol{a}\,|r=0)=0$, because if $f(\boldsymbol{a}\,|r=0)=1$ then the fact that $f(\boldsymbol{a}\,|b=0,r=1)=0$ violates the condition of f being monotone.

Also we can get $f(\boldsymbol{a}\,|r=0,b=1)=0$ and $f(\boldsymbol{a}\,|r=1,b=1)=1$. So r is relevant in $\boldsymbol{a}\,|b=1$. This means r is at least as relevant as b, that is $\mathrm{Inf}_r(f)\geq\mathrm{Inf}_b(f)$. QED

2) The general algorithm for finding counterexamples of the secondary conjecture

For the purpose of examining the effectiveness of the secondary conjecture, we have designed a preliminary program to achieve the goal of efficiently generating an influence-based optimal size tree and searching for examples against this method of tree construction. If under the current capability of computational resources, a counter-example could not be found for a certain interesting set of monotone functions, our confidence in this conjecture will be increased. The algorithm is explained below:

i) Function of Optimal tree construction-Opt(f):

```
1.  If $f$ is constant then
        return 1
2.  else
        for every relevant variable $v$ in $f$
3.          call opt($f|v\leftarrow 0$),opt($f|v\leftarrow 1$) with $v$ being set to 0 and 1:
            s0= opt($f|v\leftarrow 0$), s1= opt($f|v\leftarrow 1$)
4.          sum=s0+s1
5.      return minimum sum
6.  end if
```

ii) Function of influence-based optimal tree construction-inf_opt(f):

```
1.  if $f$ is constant return 1
2.  else
      determine $v$, any one among the most influential variables
3.    call inf_opt($f|v\leftarrow 0$),inf_opt($f|v\leftarrow 1$):
      s0= inf_opt($f|v\leftarrow 0$), s1= inf_opt($f|v\leftarrow 1$)
```

```
4.      return sum=s0+s1+1
5.  end if
```

iii) Function of finding counter example for the primary conjecture:

```
1.  for every monotone function f of n bits
2.      If inf_opt(f)≠opt(f) then
3.          Output f  (the counter example)
4.      end if
5.  end for
```

3) Influence Calculation

The most intuitive way to calculate influence of variables in a monotone function is based on how influence is defined. For a monotone function of n variables, we need to list its truth table with 2^n entries, and each entry consists of an input state and an output state. Count each time the variable of interest is flipped from 0 to 1 and the function value increases. Then by dividing this count by 2^{n-1}, we get the influence for that variable.

Another way to calculate influence is through the application of the Fourier transform, which is defined as:

$$\hat{f}(\boldsymbol{a}) = \mathbf{E}[f(x)\cdot \chi_a(x)] = 1/2^n * \boldsymbol{\Sigma}\, f(x)\cdot \chi_a(x),$$

where $\boldsymbol{a} \in \{0,1\}^n$, $\chi_a(x) = (-1)^{\boldsymbol{a}\cdot \boldsymbol{x}}$, f:$\{0,1\}^n \rightarrow \{-1,1\}$.

For any monotone function f,

$$\mathrm{Inf}_i(f) = -(\hat{f}(e_i)),$$

where $\mathrm{Inf}_i(f)$ is the influence of the ith variable in f, and e_i is a sequence of 0s and 1 with 1 appearing only on the ith bit.

The usage of Fourier coefficients may seem difficult to comprehend at first sight. Compared with the original influence calculation method, it has no computational advantage. But it is more convenient for hand calculation, especially when the number of variables n in the function gets large, because otherwise we would need to list the 2^n entries for the truth table first. Let's look at an example. A monotone Boolean decision tree can always be built for a monotone function *f* with our favorite variable as root disregarding optimality of that tree, and the tree structure supports a visual method for calculating the influence as a Fourier coefficient. Suppose

```
         x2
        /   \
      x3     x3
     /  \   /  \
   -1  x1   x1  1
      /  \ /  \
    -1  x4 -1  1
        / \
      -1   1
```

Figure (4)

is the tree representing monotone function f and x_2 is the variable for which we want to calculate influence. There are 4 variables in the tree, so the denominator of Fourier coefficient is -2^4. For each branch from root to leaf, let m be the number of missing variables in that branch, and then multiply 2^m with the leaf's value (the leaf's negative value) for all branches in the left (right) sub-tree. In this case, the leftmost branch does not have x_1 and x_4, so $2^2 * (-1)$ is the term we need; the rightmost path also misses two variables, so $2^2 * (-1)$ is the term for that path. Sum up all terms for all branches while ignoring any pair of branches having leaves that can be canceled out by each other, and then by dividing the sum by the denominator, we get $Inf_{x2}(f)= [2^2*(-1) +2*(-1)+ 2^2*(-1)]/(-2^4)=5/8$.

As introduced above, both of the methods for influence calculation have their own advantage and are alternately used to achieve better performance in this study.

4) Optimal size calculation for tree representation of monotone function of n variables.

In order to verify the optimal tree construction based on influence, we need to compare the optimal tree size of a function having a root obtained from the influence method with the actual optimal tree size. The computational complexity is also doubly exponential if we try to build the optimal tree by putting each variable iteratively at each level of the tree. We can compute the optimal size for function of n variables with a certain variable as root by simply adding up the optimal size of the sub-trees for that particular

variable plus 1; in the case when two sub-trees are identical, the optimal size of the whole tree is just the optimal size of one sub-tree plus 1.

5) Interesting Cases that might be the counterexamples to the primary conjecture

In general, there are 2^{2^n} possible Boolean functions of n variables. As illustrated below for the case of two variables:

x_1:	0	0	1	1
x_2:	0	1	0	1
F_1:	0	0	0	0
F_2:	0	0	0	1
F_3:	0	0	1	0
F_4:	0	0	1	1
F_5:	0	1	0	0
F_6:	0	1	0	1
F_7:	0	1	1	0
F_8:	0	1	1	1
F_9:	1	0	0	0
F_{10}:	1	0	0	1
F_{11}:	1	0	1	0
F_{12}:	1	0	1	1
F_{13}:	1	1	0	0
F_{14}:	1	1	0	1
F_{15}:	1	1	1	0
F_{16}:	1	1	1	1

Table (1)

There are two Boolean variables x_1 and x_2 in each function. Therefore, there will be 2^2 input states; the value of each function is an output state specified by each input state. So there are 2^{2^2} Boolean functions of 2 variables in this example.

Since the computational complexity of generating all Boolean functions of n variables is doubly exponential in the number of variables in a Boolean function, we could not make a computer automatically generate all Boolean functions. Although the set of monotone functions is just a subset of all Boolean functions, the number of monotone functions of more than 6 variables is still very large as seen from Table (2). A program was written to test all monotone functions up to 6 variables. For the purpose of examining functions of more than 6 variables, another program was written so that individual interesting functions of that many variables can be tested. The method of specifying and categorizing interesting functions is still in exploration. Three cases are listed that have been examined:

I. All monotone functions with 6 variables.

We introduce here another algorithm used to generate all distinct monotone functions for the given number of input variables.

The problem of enumerating distinct monotone functions was first studied by Dedekind in 1897. The following values are known since 1991:

n	M(n)
0	2
1	3
2	6
3	20
4	168
5	7581
6	7828354
7	2414682040998
8	56130437228687557907788

Table (2)

Here M(n) in the second column is the number of distinct monotone functions of n variables. Although the number gets huge when it reaches 8 variables, the sequence increases slowly compared with double exponential growth. Therefore, this allows us to write a computer program to generate all monotone functions of 6 variables. Beyond 6 variables, it requires not only more computational power, which a normal PC may be barely able to handle especially for the case of 7 variables, but also lots of memory, which even the most powerful computer can not afford. For example, the case of 7 variables will require at least 10 GB memory to store all the monotone functions. For this reason, we only tested a maximum total number of variables up to 6.

An elementary way of constructing all distinct monotone functions of a certain number of variables is through the usage of the truth table representation of the monotone functions. As a consequence, each n variable function is specified by 2^n input states and the corresponding output states. To construct the monotone functions of n variables, we can simply take a join of any two monotone functions F_i and F_j of n-1 variables to see if F_i will be absorbed by F_j. Here the subscripts i and j are just a pair of integer numbers, which could traverse all the monotone functions of n-1 variables. The output of the join operation of F_i and F_j will lead to a potential candidate of the n variable monotone function. Depending on the actual value of F_i U F_j, if the result still equals F_j, the adjoinment of F_i and F_j, F_iF_j is a new monotone function of n variables. For a concrete view, the procedure of generating monotone functions of 3 variables from monotone functions of 2 variables is shown below:

Input states with 2 variables

	0011
	0101
Function	
1	0000
2	0001
3	0011
4	0101
5	0111
6	1111

Input states with 3 variables

	00001111
	00110011
	01010101
Function	
1	00000000
2	00000001
3	00000011
4	00000101
5	00000111
6	00001111
7	00010001
8	00010011
9	00010101
10	00010111
11	00011111
12	00110011
13	00110111
14	00111111
15	01010101
16	01010111
17	01011111
18	01110111
19	01111111
20	11111111

Figure (5)

The right hand side lists the 20 monotone functions of 3 variables, which are constructed from the 6 monotone functions of 2 variables shown on the left hand side.

Notice the input states are positioned in an ascending order and a function such as "00010000" is not an entry of the right list since "0001" U "0000" $\neq$ "0000". Any monotone function adjoined with itself would make an entry of the right list. It is not hard to see why this method will generate only monotone functions of n variables. Whenever one more variable is added to the function of n-1 variables, each input state splits into two states, such as the states "000" and "001" from "00" in the previous example. Then certainly the functions of n variables contain twice the number of output states as functions of n-1 variables. To be a monotone function, the function value can not decrease whenever a variable is flipped from 0 to 1.This requirement is already satisfied by the initial case where 0 and 1 are the only two monotone functions of 0 variable, so at each step of the constructing procedure, all possible permutations of 0s and 1s in each output state of n variables can be exhaustively tested by simply adjoining any two functions of n-1 variables and checking the absorbability.

The algorithm for generating all monotone functions of n variables is constructed as below. The program for searching for a counter example over all monotone functions of 0 up to 6 variables is attached in the Appendix.

```
1. For every monotone function $f_i(n)$ of n variables
2.                  $f_i(n) \leftarrow 0$
3. For every monotone function $f_j$ (n-1) of n-1 variables
4.     For every monotone function $f_k$ (j) of n-1 variables
5.         If $f_j$ (n-1) OR $f_k$ (n-1) equals $f_k$ (n-1)  then
6.                 $f_j$ (n-1) shifts $2^{n-1}$ to the left;
7.                 $f_i(n) = f_j$ (n-1) OR  $f_k$ (n-1)
8.         End if
```

II. Monotone functions with decision tree representation with sub-trees having no common optimal roots at the level of the tree next to the root level when a variable only relevant in one sub-tree is the root.

Jackson had tried to prove the primary conjecture by an induction on the level of a monotone Boolean decision tree. He assumed that in a tree, if its both sub-trees are not constant, then they share a common optimal root. But counterexamples against the presumption of common optimal root were found. For example, in a monotone function F represented by $x_1x_2+x_3x_4+x_3x_5$, if we choose x_3 as the root to build a decision tree for this function, then A is the left sub-tree represented by x_1x_2 with x_3 set to 0 while B is the right sub-tree represented by $x_1x_2+x_4+x_5$ with x_3 set to 1. Clearly all input states that satisfy A will satisfy B. But A's optimal root is either x_1 or x_2 while B's optimal root is x_4 or x_5. Monotone decision trees of A and B with respect to root x_1,x_2,x_4 and x_5 are given in figure(6).

```
          x1                              x2
         /  \                            /  \
        0    x2                         0    x1
            /  \                            /  \
           0    1                          0    1
      Sx1(A)=3(leaves)               Sx2(A)=3(leaves)          A: x1+x2
        x1               x2               x4               x5
       /  \             /  \             /  \             /  \
     x4    x2         x4    x1         x5    1          x4    1
    /  \   /  \      /  \   /  \      /  \             /  \
  x5   1  x4   1   x5   1  x4   1    x1   1           x1   1
 / \      / \     / \      / \      / \               / \
0   1   x5   1   0   1   x5   1    0   x2            0   x2
        / \              / \           / \               / \
       0   1            0   1         0   1             0   1
Sx1(B)=6(leaves) Sx2(B)=6(leaves) Sx4(B)=5(leaves) Sx5(B)=5(leaves)
```

B: $x_1x_2+x_4+x_5$

Figure (6)

Therefore, testing monotone Boolean decision trees without common optimal roots on at least one level of the tree becomes interesting when a pair of variables having the relation as defined in the primary conjecture exists in the function and the one that is only relevant in one sub-tree is the root. Since a lack of common optimal root on any level of a tree having the above property is where the induction breaks down, we choose 10 monotone functions of 7 and 10 monotone functions of 8 variables without common optimal root at the level right next to the root level for testing the effectiveness of the primary conjecture. They are represented here as DNF.

For 7 variables:

1. $x_1x_2x_3x_4x_5 + x_1x_2x_3x_4x_6 + x_1x_3x_4x_5x_6 + x_7x_2$
2. $x_1x_2x_3x_4 + x_1x_5x_6 + x_3x_7$
3. $x_1x_2 + x_3x_4x_5 + x_6x_7 + x_1x_5$
4. $x_1x_2x_3x_4x_6 + x_6x_7 + x_1x_5$
5. $x_1x_2 + x_2x_3x_4 + x_4x_5 + x_4x_6 + x_4x_7$
6. $x_{1+}\ x_2 + x_3 + x_4x_5x_7 + x_6x_7$
7. $x_1x_6x_7 + x_2x_3x_4x_5 + x_2x_6x_7 + x_1x_4x_7$
8. $x_{1+}\ x_2 + x_3x_4x_5 + x_4x_5x_7 + x_6x_7$
9. $x_1x_2x_3x_4 + x_1x_4x_5x_6 + x_3x_7$
10. $x_1x_4x_7 + x_1x_3x_5x_7 + x_1x_6x_7 + x_2x_3x_4x_5 + x_2x_6x_7$

For monotone functions of 8 variables:

1. $x_1x_2x_3x_4x_5x_6 + x_1x_2x_3x_4x_5x_7 + x_1x_3x_4x_5x_6x_7+x_8x_2$
2. $x_1x_2x_3x_4 + x_1x_5x_6 + x_3x_7x_8$
3. $x_1x_2 + x_3x_4x_5 + x_6x_7x_8+ x_1x_5$
4. $x_1x_2x_3x_4x_6 +x_6x_7x_8 + x_1x_5$
5. $x_1x_2 + x_2x_3x_4x_8 + x_4x_5 + x_4x_6 + x_4x_7$
6. $x_{1+}\ x_2 + x_3 + x_4x_5x_7x_8 + x_6x_7$
7. $x_1x_6x_7 + x_2x_3x_4x_5 + x_2x_6x_7x_8 + x_1x_4x_7$
8. $x_{1+}\ x_2 + x_3x_4x_5x_8 + x_4x_5x_7 + x_6x_7$
9. $x_1x_2x_3x_4 + x_1x_4x_5x_6x_8 + x_3x_7$
10. $x_1x_4x_7 + x_1x_3x_5x_7 + x_1x_6x_7 + x_2x_3x_4x_5 + x_2x_6x_7x_8$

These monotone functions were tested in a program where a particular DNF is the input with the number of variables in the function not limited to 8, and the output of the program gives a counterexample if the DNF violates the primary conjecture or secondary conjecture. The result for each function was given instantaneously and for these functions, there is no counterexample against the primary conjecture.

III. Monotone functions with a sub-tree having no v_r and v_b (as stated in the primary conjecture) larger than a one node sub-tree.

It has been shown by Jackson (unpublished) that if ***C*** is a sub-tree in which neither v_r nor v_b is relevant, and ***C*** is simply a leaf or a one node sub-tree, then decision trees with ***C*** as a sub-tree also satisfy the primary conjecture and therefore are not the counterexamples we are looking for. But once ***C*** gets larger than a one node sub-tree, there has not been a good way to prove for this case that the primary conjecture still holds. Therefore, we are searching for counterexamples in the functions with this property. Below are 10 monotone functions of 7 and 10 monotone functions of 8 variables with the property represented here as DNF and there is counterexample among them.

For 7 variables: $x_1x_2x_3x_4x_5x_6 + x_1x_2x_3x_4x_5x_7 + x_1x_3x_4x_5x_6x_7+x_8x_2$

1. $x_1x_3 + x_1x_5 + x_2x_4x_6x_7$
2. $x_1x_6x_7 + x_3x_4x_5 + x_2x_6 + x_4x_6$
3. $x_1x_3x_4 + x_5x_6 + x_2x_7 + x_3x_4\ x_7$
4. $x_1x_2x_3x_4 + x_2x_3x_4x_5 + x_3x_4x_5x_6 + x_4x_5x_6x_7$
5. $x_1x_2 + x_2x_3x_4 + x_3x_4x_5+x_6x_7 + x_4x_5x_6$
6. $x_1x_2x_3 + x_4x_5x_6 + x_7x_1$
7. $x_1x_2x_3x_4 + x_5x_6x_7 + x_3x_4x_5x_6$
8. $x_1x_2x_4 + x_3x_4x_6 + x_5x_7$
9. $x_1x_4 + x_3x_6 + x_2x_4x_5x_7$
10. $x_1x_3x_7 + x_3x_5x_6 + x_2x_3x_4 + x_4x_5x_6x_7$

For monotone functions of 8 variables:

1. $x_1x_3x_8 + x_1x_5 + x_2x_4x_6x_7$
2. $x_1x_6x_7 + x_3x_4x_5 + x_2x_6x_8 + x_4x_6$
3. $x_1x_3x_4 + x_5x_6 + x_2x_7x_8+x_3x_4x_7$

4. $x_1x_2x_3x_4\ x_5 + x_2x_3x_4x_5x_6 + x_3x_4x_5x_6x_7 + x_4x_5x_6x_7x_8$
5. $x_1x_2 + x_2x_3x_4 + x_3x_4x_5 + x_6x_7x_8 + x_4x_5x_6$
6. $x_1x_2x_3 + x_4x_5x_6 + x_7x_1x_8$
7. $x_1x_2x_3x_4 + x_5x_6x_7 + x_3x_4x_5x_6x_8$
8. $x_1x_2x_4 + x_3x_4x_6 + x_5x_7x_8$
9. $x_1x_4 + x_3x_6 + x_2x_4x_5x_7x_8$
10. $x_1x_3x_7 + x_3x_5x_6 + x_2x_3x_4 + x_4x_5x_6x_7x_8$

4. Future work

I. By the way of constructing a DNF representation for a monotone function, it does not seem like two variables v_r and v_b with the relation defined in the primary conjecture could have the same influence. The proof of $\mathrm{Inf}_r(f) \geq \mathrm{Inf}_b(f)$ does not take into account that v_r is relevant in both sub-trees of v_b , so it does not exclude a case like x_1x_2 . Although the bound with equality is sufficient for applying influence-based optimal tree construction, the proof can be revised to prove $\mathrm{Inf}_r(f) > \mathrm{Inf}_b(f)$.

II. The two programs written for the purpose of finding counterexamples against the primary conjecture can be further revised so that they could accommodate more variables in functions. That will increase the probability of finding a counterexample if there is any.

5. Conclusion

It has been shown in this thesis that this conjecture holds for monotone functions of 6 variables and less. Because the doubly exponential computational complexity and the limitation of the common computer, we only construct all monotone functions of up to 6 variables, and there are 7828354 monotone functions of 6 variables are generated and tested for finding counterexamples against the conjecture. Additional efforts are made to test some interesting functions of 7 and 8 variables chosen based on certain methods. These methods and the methods including building optimal tree with and without influence are also further introduced in the thesis. In a conclusion, about 8 million monotone functions as described in this thesis are tested and no counterexample has been found in this sample. This increases our confidence in favor of the conjecture.

Bibliography

1. J. Jackson, and R. Servedio, *Learning Random log-depth Decision Trees under the Uniform Distribution*, Proceedings of the 16th Annual Workshop on Computational Learning Theory, 2003.
2. R. O'Donnell and R. Servedio. *On decision trees, influence, and learning monotone decision trees*. (unpublished)
3. L.G. Valiant. A theory of the learnable. *Communications of the Association for Computing Machinery*, 27(11):1134-1142, 1984.
4. M. Kearns , M. Li , L. Pitt , L. Valiant, *On the learnability of Boolean formulae*, Proceedings of the nineteenth annual ACM conference on Theory of computing, 285-295, 1987.
5. J. Jackson, *An Efficient Membership-Query Algorithm for Learning DNF with Respect to the Uniform Distribution*, Journal of Computer and System Sciences 55(3), 1997.

Appendix

```
/*
 * This is the program written to calculate the opt size of the monotone
 * function up to 6 variables.
*/
# include <stdio.h>
# include <math.h>
# include <stdlib.h>
# include <fstream.h>
# include <iostream.h>

#define Nterm 4
#define SFT3 8 // 2^3
#define SFT4 16 // 2^4
#define SFT5 32 // 2^5
#define N_3 20 //the number of monotone functions for 3 vars
#define N_4 168 //the number of monotone functions for 4 vars
#define N_5 7581 //the number of monotone functions for 5 vars
#define N_6 7828354 //the number of monotone functions for 5 vars
#define MAX 10000 //just a large number



void nrerror(char error_text[])
        /*Numerical Recipes standard error handler*/
{
        int i;
        printf("\n\n");
        fprintf(stderr,"Numerical Recipes run-time error...\n");
        fprintf(stderr,"%s\n",error_text);
        fprintf(stderr,"...now exiting to system...\n");
        exit(1);
```

```
}



int * inf_cal(unsigned long mfv,int n,int t2n,int * mindex)

    //mfv: monotone function's value; n: number of variables in the function; t2n: number
    //equal to 2^n.
{
        //to calculate the inf using the naive method, which defines the inf as a ratio of the
        //numb of changed f(v)|v:0->1,
        //with the total 2^n numbers. And here the inf is defined without normalized by 2^n
        //since one only cares the relative inf
        int i,j;
        unsigned long tmp;
        int indxf;
        long  max=-1;
                                        //for monotone function value from bit 0 to bit 15
                                        /*
                                         * x0 00000000 | 11111111
                                         * x1 00001111 | 00001111
                                         * x2 00110011 | 00110011
                                         * x3 01010101 | 01010101
                                         * -----------------------
                                         * f  00000001 | 00000011 this is just one of 168 functions of
                                         *4 vars
                                         * bit0---------------->bit 15
                                         */



        //cal the inf for each varible
        for(j=0;j<n;j++)//for each var
```

```
{
        vinf[j]=0; //initialize
        for(i=0;i<t2n;i++)//go through the whole list
                       //namely 0,1,.....,2^n-1
        {
         if(!((i>>(n-1-j))&1))//if the var_j is 0
                          //this definition of j is consistent with elsewhere in this
                          //program
         {
                 if((t2n-1-i)<32){
                         tmp=1&(mfv>>(t2n-1-i)); //get the fvalue of bit n-1-i
                 }
                 else{ tmp=1&(mfv>>16>>16>>(t2n-1-i-32));}
                 indxf=(i|(1<<(n-1-j))); //flip the var_j from 0 to 1

                 if((t2n-1-indxf)<32){
                         if(tmp!=(1&(mfv>>(t2n-1-indxf))))//if the DNF changes
                                 vinf[j]++;
                 }
                 else{
                         if(tmp!=(1&(mfv>>16>>16>>(t2n-1-32-indxf))))//if the
                          //DNF changes
                                 vinf[j]++;

                 }
         }
        }//end of the list
} //end of each var loop

for(j=0;j<n;j++) //for each var
{
```

```
                        if(max <= vinf[j])
                {
                        mindex[j]=j;
                        max=vinf[j];
                }
        }

        //now return the index of the max influce var
        return mindex;
}

//the following method search an array of m var function lish and returns its
//optsize of that monotone function
int srch_opts(unsigned long *flist, int *optlist, unsigned long  fvalue, int length)
        //the flist is an array of monotone functions
        //optlist is an array of optsize of the monotone functions
        //fvalue is the searching key and length is the size of the array
{
        int L=0, R=length-1;
        int M; //the left index rightmost index and middle index
        int found=0;
        while(L!=R&&!found)
        {
                if((R-L)==1)
                {
                        if(fvalue==flist[L])
                        {
                                M=L;
                                found=1;
                        }
                        else if(fvalue==flist[R])
                        {
                                M=R;
                                found=1;
                        }
                        else
                        { }
                }
                else
                {
                        M=(L+R)/2;

                        if(fvalue>flist[M])
                        {
                                L=M;
                        }
                        else if(fvalue<flist[M])
                        {
                                R=M;
                        }
                        else //equal values
                        {
                                found=1;
                        }
                }
        }
        return optlist[M];
}

int srch_opts2(unsigned long *flist, int *optlist, unsigned long  fvalue, int length)
        //the flist is an array of monotone functions
        //optlist is an array of optsize of the monotone functions
```

```
        //fvalue is the searching key and length is the size of the array
{
        int L=0, R=length-1;
        int i; //the left index rightmost index and middle index
      for(i=0;i<length;i++)
        {
                if(flist[i]==fvalue)
                        break;
        }
        return optlist[i];
}

int  gen_mnt(unsigned long *mntf_pre, unsigned long *mntf_nxt,int N_pre,int N_nxt,int
sft)
        /*
         * this function generate the mnt_function of n var from mnt_function of n-1 var
         * mntf_pre is the list of mnt functions of n-1 var
         * mnt_nxt is the list of mnt functions of n var
         * int N_pre is the number of mntf of n-1 var
         * and the int N_nxt is the number of mntf of n var
         * sft is the shift numbre and equals 2^(n-1)
         */
{
        int i,j;
        //construct the 4 var monotone function
        for(i=0;i<N_nxt;i++)
        {
                mntf_nxt[i]=0;
        }

        int cnt=0; //the counter
        for(i=0;i<N_pre;i++)
        {
                for(j=i;j<N_pre;j++)
                {
                        //generate the 4 var monotone functions from the N_3 3 var
                        //functions
                        if((mntf_pre[i]|mntf_pre[j])==mntf_pre[j])
                        {

        mntf_nxt[cnt]=(mntf_pre[i]<<1<<(sft-1))|mntf_pre[j];//incase sft==32, the << can
                                                                                              //not handle >=32
                                cnt++;
                        }
                }
        }
        return cnt;
}

void left_right(unsigned long &left, unsigned long &right,int NV, int nv,unsigned long
mf,int N1,int N2)
        /*
         * suppose that the N1=2^3 is the number of inputs of 3 var, and the N2=2^4=16 is
         *now the number of inputs of 4 var; then
         * left, right;  each stores the left and right subtree 3 vars monotone function
         *  value, which is the key to find the optsize from the 3 var stored result
         *  The nv is the nth var. For N2=16, nv=0,1,2,or 3.
      *  NV is the # of variables in the MF
      * mf is the MF's value.
         */
{
        int i,j, l,r;
```

```
unsigned long temp;
i=nv;
left=0; //need to double check the position of these two lines
right=0;
l=0; r=0; //the left and right counter are set to zero in the beginning
for(j=0;j<N2;j++)
{
        //for 4 vars for i=0 to 3
        //for monotone function value from bit 0 to bit 15
        /*
         * x0 00000000 | 11111111
         * x1 00001111 | 00001111
         * x2 00110011 | 00110011
         * x3 01010101 | 01010101
         * -----------------------
         * f  00000001 | 00000011 this is just one of 168 functions of 4 vars
         * bit0---------------->bit 15
         */
        if((j>>(NV-i-1))&1)
        {
                //right
        if(j>=32) temp=mf>>N2-j-1;
        else temp=mf>>N2-(j+32)-1>>16>>16;
                if(temp&1)
                {
                        right|=(1<<(N1-1-r)); //mask that bit to 1
                        r++;
                }
                else
                {
                        //right&=(0<<(N1-1-r)); //mask that bit to 0
                        r++;
                    }
                }
                else
                {
                    //left
                if(j>=32) temp=mf>>N2-j-1;
                else temp=mf>>N2-(j+32)-1>>16>>16;
                    if(temp&1)
                    {
                        left|=(1<<(N1-1-l)); //mask that bit to 1
                        l++;
                    }
                    else
                    {
                        //left&=(0<<(N1-1-l)); //mask that bit to 0
                        l++;
                    }
                }
            }
}

 void opt_size_cal(unsigned long* mntnf_pre,unsigned long* mntnf,int* optsz_pre, int*
optsz,int Nf_pre, int Nf, int nv)
        /*
         * This is the function to calculate the opt size of each monotone function for
         * a given number of vars, i.e., 3 vars, 4 vars, 5 vars, 6 vars etc.
         * Nf= 168 monotone functions for nv=4 vars
         * Nf_pre is 20 then since 3 var is used to cal 4 vars
         * nv is the number of variables in the monotone function
       */
```

```
{
        unsigned long left, right;  //each stores the left and right subtree 3 vars monotone
                    //function
                    //value, which is the key to find the optsize from the 3 var stored result
    int k, l, r, i;
        int tree_size, trszl, trszr; //size of trees
        int min;
        int *index_opt=new (int [nv]),min_index_opt;
        int nterm; //the number of total input terms, e.g. when nv=4 nterm=2^4=16
        int hfnt;
        unsigned long  tmp1=0,tmp2=Nf;
        nterm=(int) pow(2.0, nv);
        hfnt=nterm/2;
    for (i=0;i<nv;i++)
        {
                index_opt[i]=nv;
        }


//      for(k=0;k<Nf;k++) //N_4 number of monotone function of 4 vars
    //if(nv==5) printf("Nf_pre is %d\n",Nf_pre);
        for(k=tmp1;k<tmp2;k++) //N_4 number of monotone function of 4 vars
        {
                min=1000;
          // if (nv==6)          printf("k is %d\n",k);
                index_opt=inf_cal(mntnf[k],nv,nterm,index_opt);//get the index of the
                    //variable with the largest influence
         // cout<<"index_opt is %f"<<index_opt<<"\n";
          for(i=0;i<nv;i++)
                {
                if( k%1000==0){
```

```
            cout<<"the ith variable\t"<<i<<"\n";
          cout<<"mntnf["<<k<<"] of 6 variables "<<mntnf[k]<<"\n";}
            left_right(left,right,nv,i,mntnf[k],hfnt,nterm);


            //part I, find min optsize w/o considering the influence
      //after one gets the left and right subtree value for a give var as the
      //root
            //then calculate the optsize tree for each var as a root.
            //Finally one chooses the min optsize tree
            trszl=srch_opts(mntnf_pre,optsz_pre,left,Nf_pre);
            trszr=srch_opts(mntnf_pre,optsz_pre,right,Nf_pre);
            if(left == right)
       tree_size=trszl;   //the tree size is just one sub-tree's size
      else
       tree_size=trszl+trszr+1;
          if(min>=tree_size)
                  {
                        min=tree_size;
              min_index_opt=i;
                  }
            //Part II, find min optsize by considering the influence of a var
      //if the above part I and part II give two different answer, there is a conter
      //example found.
 }
      //send the results to storage
int count=0;
      for(i=0;i<nv;i++)
      {
   if(index_opt[i] < nv && min_index_opt == index_opt[i]) count=1;
      }
         if(count == 0)
```

```
                {
                        printf("\n**********mntnf[%d] is %d\n ",k,mntnf[k]);
                        printf(" nv is %d\n",nv);
                        printf("the counter example is found now\n");
                        printf("the optimal root w/o influence(min_index_opt) is%d\n
",min_index_opt);
                        printf("the true min is %d\n ",min);
                break;
                }
                optsz[k]=min;
        }
        delete [] index_opt;
}

int main(int argv, char *argc[])
{
        int i=0,cnt;
        unsigned long monotf; //the monotone function f
        int opts; //the optsize of the fuction f
        unsigned long * mntnf4= new (unsigned long [N_4]);
        if(mntnf4==NULL) nrerror("some thing is wrong with the allocation of
mntnf4\n");
        int * optsz4= new (int [N_4]);
        if(optsz4==NULL) nrerror("some thing is wrong with the allocation of optsz4\n");
        unsigned long * mntnf5= new (unsigned long [N_5]);
        if(mntnf5==NULL) nrerror("some thing is wrong with the allocation of
mntnf5\n");
        int * optsz5= new (int [N_5]);
        if(optsz5==NULL) nrerror("some thing is wrong with the allocation of optsz5\n");
        unsigned long * mntnf6= new (unsigned long [N_6]);
```

```
            if(mntnf6==NULL) nrerror("some thing is wrong with the allocation of
mntnf6\n");
            int * optsz6= new (int [N_6]);
            if(optsz6==NULL) nrerror("some thing is wrong with the allocation of optsz6\n");

            unsigned long mntnf[N_3];  int optsz[N_3];
            if(argv<2)
            {
              printf("syntax: <inf_cal.out> <txt file> \n");
              exit(0);
            }
            //the following reads a  3 variable monotone function from a file and
            //stores in an array
        ifstream fin;
            fin.open(argc[1],ifstream::in);
            while(!fin.eof())
            {
                        fin >>monotf>>opts;
                        mntnf[i]=monotf;
                        optsz[i]=opts;
                        i++;
                        if(i>=20) break;
            }
            fin.close(); //finishing reading the  file

            //construct the 4 var monotone function and calculate the opt size
            for(i=0;i<N_4;i++){
                        optsz4[i]=0;
            }
        cnt=gen_mnt(mntnf,mntnf4,N_3,N_4,SFT3);
```

```
    //after successfully generate the monotone function of 4 vars
    //lets calculate the optimal size for each function
    opt_size_cal(mntnf,mntnf4,optsz,optsz4,N_3,N_4,4);




    //construct the 5 var monotone function and calculate the opt size
    for(i=0;i<N_5;i++){
            optsz5[i]=0;
    }
cnt=gen_mnt(mntnf4,mntnf5,N_4,N_5,SFT4);

    //after successfully generate the monotone function of 5 vars
    //lets calculate the optimal size for each function
    opt_size_cal(mntnf4,mntnf5,optsz4,optsz5,N_4,N_5,5);

    //construct the 6 var monotone function and calculate the opt size
    for(i=0;i<N_6;i++){
            optsz6[i]=0;
    }
cnt=gen_mnt(mntnf5,mntnf6,N_5,N_6,SFT5);

    //after successfully generate the monotone function of 6 vars
    //lets calculate the optimal size for each function
    opt_size_cal(mntnf5,mntnf6,optsz5,optsz6,N_5,N_6,6);

    //clean the space
delete [] mntnf4;
    delete [] optsz4;

delete [] mntnf5;
```

```
            delete [] optsz5;

        delete [] mntnf6;
            delete [] optsz6;

            return 0;
}
```