Implementation Issues in the Fourier Transform Algorithm*

YISHAY MANSOUR

SIGAL SAHAR

Computer Science Department, Tel-Aviv University, Tel-Aviv, Israel

mansour@cs.tau.ac.il

gales@cs.tau.ac.il

Editor: Jude Shavlik

Abstract. The Fourier transform of Boolean functions has received considerable attention in the last few years in the computational learning theory community, and has come to play an important role in proving many important learnability results. The aim of this work is to demonstrate that the Fourier transform techniques are also a useful and practical algorithm, in addition to having many interesting theoretical properties. In fact, this work was prompted by a genuine problem that was brought to our attention; researchers at a company were trying to come by a method to reverse-engineer a state-free controller. They had the capability of querying the controller on any input, thus setting them in the membership query model, in which the Fourier transform algorithm is set.

In order to keep the algorithm run-time reasonable and still produce accurate hypotheses, we had to perform many optimizations. In the paper we discuss the more prominent optimizations, ones that were crucial and without which the performance of the algorithm would severely deteriorate. One of the benefits we present is the confidence level the algorithm produces in addition to the predictions. The confidence level measures the likelihood that the prediction is correct.

Keywords: computational learning theory, Fourier transform, learning algorithms, implementation issues

1. Introduction

The importance of using the “right” representation of a function has been widely recognized in the artificial intelligence community. The Fourier transform representation of a function is a classic representation which is widely used to approximate real functions (i.e., functions whose inputs are real numbers). Over the last few years the Fourier transform representation of Boolean functions has been an instrumental tool in the computational learning theory community. It has been used mainly to demonstrate the learnability of various classes of functions with respect to the uniform distribution.

The first connection between the Fourier representation and the learnability of Boolean functions was established by Linial, Mansour, and Nisan (1993). The main application of the Fourier representation in that work was to learn the class AC^0 (constant depth, polynomial size circuits) in quasi-polynomial time (that is, $O(n^{\text{poly} - \log(n)})$) with respect to the uniform distribution. The work of Kushilevitz and Mansour (1993) has developed a very powerful algorithmic procedure; given a function and a threshold parameter it finds in polynomial

*A preliminary version was presented in Advances in Neural Information Processing Systems (NIPS) 1995.

time all the Fourier coefficients of the function that are greater than the threshold. Originally the procedure was used to learn decision trees (Kushilevitz & Mansour, 1993), and later it was used for learning polynomial size DNF (Mansour, 1995; Blum et al., 1994; Jackson, 1994). The Fourier transform technique applies naturally to the uniform distribution, and, indeed, most of the learnability results based on the Fourier transform are with respect to the uniform distribution. Some of the results were extended to the product distribution (Bellare, 1992; Furst, Jackson, & Smith, 1991).

A great advantage of the Fourier transform algorithm is that it needs not make any assumptions regarding the function it is learning. Thus, we can apply it to any function and hope to obtain some “large” Fourier coefficients which can then be easily used in order to make a prediction. The prediction function simply computes the sum of the coefficients with the corresponding basis functions and compares the sum to a threshold. (Essentially it can be viewed as a perceptron, where the Fourier coefficients are the weights and the inputs are the values of the appropriate basis functions.) The procedure is also immune to a certain rate of noise, and it will be able to operate even if a fraction of the examples are maliciously misclassified. A drawback of the procedure is that it requires querying the target function on various randomly selected inputs. Fortunately, in some real-life cases the target function *is* available in the form of a black-box. Indeed, we were motivated to implement the Fourier transform algorithm in order to solve a particular problem of this type: reverse-engineering.

When faced with the task of reverse-engineering a certain chip, for example, all queries can be performed on the chip by setting the appropriate input lines and observing the output. In other words, membership queries are available, and they are the only available source of information. In many cases, the chip does not maintain information from one invocation to the other; in such cases it can be viewed as a fixed function. These cases are practically tailored for the application of the Fourier transform algorithm. Our work was originally motivated by a particular, real-world reverse-engineering problem that we brought to us and prompted us to convert the Fourier transform algorithm into a practical tool.

The aim of this work is to implement this procedure in an efficient way so that, as we shall demonstrate, it is not only a useful theoretical tool but also a practical one. The procedure scales linearly with the number of input variables, conveniently allowing us to run it on a fairly large number of input variables. The experiments are based on learning of various functions such as decision trees, DNF and majority, and “real” testbeds with the number of input variables ranging from a few dozen to a few hundred. The results demonstrate that the procedure can approximate a wide variety of different function classes.

In the process of implementing the Fourier algorithm we made many important optimizations and enhancements. The optimizations were essential in order to make the algorithm run in a reasonable amount of time. The enhancements significantly improve the accuracy of the hypothesis we generate. Choosing the “right” parameters proved to be imperative in order to achieve desirable running time while maintaining a reasonable error rate. In the process of tuning the algorithm we have come across many interesting issues: the confidence level that we can output with our prediction and the error distribution which clearly supports our assumptions.

When making predictions, it is extremely beneficial to have the prediction algorithm supply an indicator that provides the confidence level the algorithm has in the prediction

it just made. Some learning algorithms, such as neural nets (Golden, 1996), contain a confidence level indicator (Nix & Weigend, 1995).

Our algorithm provides us naturally with such an indicator. Suppose that in the prediction process we are computing a sum and then comparing it to some threshold, say 0. Intuitively we can use the value of this sum to indicate how confident we are of our prediction. If the sum is near 0 then we are probably not doing much better than guessing, while if it is far from 0 we can claim that we are confident of our prediction. Though this simple logic thus far has not been supported, so far, by any theoretical result, our experimental results provide overwhelming evidence that this is indeed the case.

One of the most important enhancements we have added is the ability to detect “on the fly” inaccurate approximations (that the algorithm computes throughout its search) and to immediately correct them. This feature considerably improves the performance of the algorithm with a minimal run time cost.

The paper is organized as follows: Section 2 defines the Fourier transform and provides a description of the algorithm. Section 3 provides a description of the experiments and their results. Section 4 describes various aspects of our implementation and the major enhancements we have incorporated into the algorithm. In Section 5 we present our motivation and the results of the Fourier transform algorithm when run on scenarios obtained from a “real” source. We then show a comparison between the performance of the Fourier transform algorithm and Quinlan’s C4.5 algorithm (Quinlan, 1993) in Section 6. We end with our conclusions and a short description of future work in Section 7.

2. Preliminaries—Fourier transform theoretical background

In this section we sketch the ideas behind Fourier transform, its connection to learning and the algorithm that finds the large coefficients. A comprehensive survey of the theoretical results and proofs can be found in Mansour (1994).

2.1. The Fourier transform

Boolean functions of n variables are considered as real valued functions, that is, $f : \{0, 1\}^n \rightarrow \{-1, +1\}$. Our aim is to learn the class of Boolean functions. The learning will take place in the *Membership Query Model* (Angluin, 1987). In this model, we presuppose the existence of a procedure that for every input $x \in \{0, 1\}^n$ provides the correct response $f(x)$. We will often refer to this procedure as an *oracle* or a *black-box*.

2.1.1. The Fourier transform basis. We define an orthonormal basis for the class of Boolean functions as follows: for each $z = \langle z_1, \dots, z_n \rangle \in \{0, 1\}^n$, define the *basis function* $\chi_z(x)$:

$$\chi_z(x_1, \dots, x_n) \stackrel{\text{Def}}{=} (-1)^{\sum_{i=1}^n x_i z_i}.$$

Alternatively, the basis functions can be defined using a subset $S \subseteq \{1, \dots, n\}$. The set S

defines the set of inputs on which the function χ_S is defined as follows,

$$\chi_S(x_1, \dots, x_n) \stackrel{\text{Def}}{=} \prod_{i \in S} (-1)^{x_i} = \begin{cases} +1 & \text{if } \sum_{i \in S} x_i \text{ is even} \\ -1 & \text{if } \sum_{i \in S} x_i \text{ is odd.} \end{cases}$$

We then get:

$$\chi_S \equiv \chi_z$$

where $S \subseteq \{1, \dots, n\}$ and $i \in S \iff z_i = 1$. Thus, χ_z is defined for $\|z\| \leq n$, where $\|z\|$ is the size of, or number of bits in, z .

The inner product of two functions is defined by:

$$\langle g, f \rangle \stackrel{\text{Def}}{=} E_{x \in \mathcal{U}}[g(x) \cdot f(x)] = \frac{1}{2^n} \sum_{x \in \{0,1\}^n} f(x)g(x), \quad (1)$$

where E is the expected value with respect to the uniform distribution, $\mathcal{U}$.

2.1.2. The Fourier representation. Any function f of n Boolean inputs can be uniquely expressed as a linear combination of the Fourier basis functions. We call this linear combination the *Fourier Transform* or *Fourier Representation* of the function:

$$\forall f : \{0, 1\}^n \rightarrow \mathbb{R}, \quad f(x) = \sum_{\alpha \in \{0,1\}^n} \hat{f}(\alpha) \chi_\alpha(x),$$

where $\hat{f}(z)$ is a real number that denotes the z th *Fourier coefficient* of f . We call z the *coefficient-name* of $\hat{f}(z)$. The Fourier coefficients are computed by

$$\hat{f}(z) = \langle f, \chi_z \rangle. \quad (2)$$

In order to see why the equality in Eq. (2) holds, we use the Fourier representation of $f(x)$ as follows:

$$\langle f, \chi_z \rangle = \left\langle \left(\sum_y \hat{f}(y) \chi_y(x) \right), \chi_z \right\rangle = \sum_y \hat{f}(y) \langle \chi_y, \chi_z \rangle = \hat{f}(z),$$

where the last identity is due to the orthonormality of the Fourier basis.

The orthonormality of the Fourier basis also implies Parseval's Identity:

$$\forall f : \{0, 1\}^n \rightarrow \mathbb{R}, \quad \sum_{\alpha \in \{0,1\}^n} \hat{f}^2(\alpha) = E[f^2], \quad (3)$$

since

$$\begin{aligned}
 E[f^2] &= E\left[\left(\sum_{\alpha} \hat{f}(\alpha)\chi_{\alpha}(x)\right) \cdot \left(\sum_{\beta} \hat{f}(\beta)\chi_{\beta}(x)\right)\right] \\
 &= \sum_{\alpha} \sum_{\beta} \hat{f}(\alpha)\hat{f}(\beta)E[\chi_{\alpha} \cdot \chi_{\beta}] \\
 &= \sum_{\alpha} \hat{f}^2(\alpha).
 \end{aligned}$$

2.1.3. Goal of the Fourier transform algorithm. Our aim is to create a learning algorithm that given a target function, f , as a black-box or oracle, will output a “good” hypothesis, h , of f . A good hypothesis is one that with high probability will have little error with respect to the target function. We define the *error* between two functions as:

$$\text{error}(f, h) \stackrel{\text{Def}}{=} \Pr_{x \in \mathcal{D}}[f(x) \neq h(x)],$$

where $\mathcal{D}$ is any arbitrary distribution. A hypothesis is then considered to be “good” if

$$\forall \varepsilon, \delta > 0, \quad \Pr_{x \in \mathcal{D}}[\text{error}(f(x), h(x)) \leq \varepsilon] \geq 1 - \delta$$

is satisfied. That is, if with arbitrarily high confidence, $1 - \delta$, we can be certain of an arbitrarily small error, ε .

An algorithm will be considered efficient if it is polynomial in n , $1/\varepsilon$ and $\log \frac{1}{\delta}$. Our goal is to arrive at an efficient algorithm that will output a good hypothesis for a given target function. We describe such an algorithm in the next section. Note that although the above definitions hold for any distribution, in this work we concern ourselves with the uniform distribution.

2.2. The algorithm: background

The Fourier transform algorithm constructs a hypothesis that contains the t largest Fourier coefficients of the target function. In this section we explain why such a hypothesis is a good one and in the following section we explain how to find and calculate these large coefficients.

2.2.1. Motivation. Calculating all the Fourier coefficients of the target function yields the accurate Fourier representation of the function. This is seemingly the simplest and best hypothesis available. However, since a target function of n Boolean input variables has an exponential number of Fourier coefficients, 2^n , calculating this hypothesis would take exponential time, while our goal is to arrive at a polynomial runtime algorithm. We choose to construct a hypothesis based on only *some* of the coefficients. That is,

$$h(x) = \sum_{z \in \Lambda} \hat{f}(z)\chi_z(x),$$

where Λ is a subset of the Fourier coefficients.

We define a *t-sparse function* to be a function whose Fourier transform has at most t non-zero Fourier coefficients. Our learning algorithm will efficiently construct a *t-sparse hypothesis* of the target function.

If we had to construct a hypothesis that was 1-sparse, the natural choice would be:

$$h_1(x) = \hat{f}(\alpha_1)\chi_{\alpha_1}(x),$$

where $\hat{f}(\alpha_1)$ is the largest Fourier coefficient, that is, $\hat{f}(\alpha_1) \geq \hat{f}(\alpha_2) \geq \dots \geq \hat{f}(\alpha_{2^n})$. If we had to construct a hypothesis that was 2-sparse, the natural choice would be

$$\begin{aligned} h_2(x) &= \hat{f}(\alpha_1)\chi_{\alpha_1}(x) + \hat{f}(\alpha_2)\chi_{\alpha_2}(x) \\ &= h_1(x) + \hat{f}(\alpha_2)\chi_{\alpha_2}(x). \end{aligned}$$

A natural choice for a 3-sparse hypothesis would then be

$$h_3(x) = h_2(x) + \hat{f}(\alpha_3)\chi_{\alpha_3}(x),$$

and so on.

Intuitively, we approximate the target function, f , by a *t-sparse function*, h , that simply includes the t largest coefficients of f as follows. Let $\Lambda = \{\alpha_1, \dots, \alpha_t\}$ be the t coefficient names that correspond to the largest Fourier coefficients of the target function, f . The hypothesis h can then be expressed as:

$$h(x) = \sum_{i=1}^t \hat{f}(\alpha_i)\chi_{\alpha_i}(x).$$

Since calculating the exact value of a Fourier coefficient takes exponential time (see Eq. (1)), we will use an approximation, a_i , of $\hat{f}(\alpha_i)$. Our *t-sparse hypothesis* will then be:

$$h(x) = \sum_{i=1}^t a_i \chi_{\alpha_i}(x). \tag{4}$$

2.2.2. Prediction. The hypothesis in Eq. (4) is not necessarily a Boolean function. In order to get a Boolean output, we will return $+1$ if $h(x) \geq 0$ and -1 otherwise. This is equivalent to superposing the *Sign* function onto the hypothesis function. The *Sign* function returns the sign of its parameter. It is formally defined in Eq. (5).

$$\text{Sign}(x) \stackrel{\text{Def}}{=} \begin{cases} +1 & \text{if } x \geq 0 \\ -1 & \text{if } x < 0 \end{cases}. \tag{5}$$

Our hypothesis is therefore:

$$h(x) = \text{Sign}\left[\sum_{i=1}^t a_i \chi_{\alpha_i}(x)\right],$$

where a_i is the approximation of the i th largest Fourier coefficient of f .

We would like to bound the error we get from using the hypothesis, h , to approximate the target function, f , using the expected error squared, i.e., $E[(f - h)^2]$.

Claim 1. $\forall f : \{0, 1\}^n \rightarrow \{+1, -1\}, \forall h : \{0, 1\}^n \rightarrow \mathbb{R},$

$$Pr[f(x) \neq \text{Sign}(h(x))] \leq E[(f - h)^2].$$

Proof: Let I be the indicator function, i.e.,

$$I[f(x) \neq \text{Sign}(h(x))] \stackrel{\text{Def}}{=} \begin{cases} 1 & \text{if } f(x) \neq \text{Sign}(h(x)) \\ 0 & \text{if } f(x) = \text{Sign}(h(x)) \end{cases}$$

The probability of error is:

$$Pr[f(x) \neq \text{Sign}(h(x))] = \frac{1}{2^n} \sum_x I[f(x) \neq \text{Sign}(h(x))].$$

We show that for every $x \in \{0, 1\}^n$, $I[f(x) \neq \text{Sign}(h(x))] \leq (f(x) - h(x))^2$, which implies the claim. We consider the following two cases.

- If $f(x) = \text{Sign}(h(x))$ then $I[f(x) \neq \text{Sign}(h(x))] = 0$, so clearly,

$$I[f(x) \neq \text{Sign}(h(x))] = 0 \leq (f(x) - h(x))^2.$$

- If $f(x) \neq \text{Sign}(h(x))$ then we have $|f(x) - h(x)| \geq 1$. Therefore,

$$I[f(x) \neq \text{Sign}(h(x))] = 1 \leq (f(x) - h(x))^2,$$

completing the proof. □

Thus, bounding the expected error squared bounds the Boolean prediction error. For a given t , the hypothesis h which is t -sparse and minimizes $E[(f - h)^2]$ simply includes the t largest coefficients of f . Note that the more coefficients we include in our approximation and the better we approximate their values, the smaller $E[(f - h)^2]$ is going to be. This provides us with the motivation to find the “large” coefficients. An efficient algorithm to find the large coefficients is described in Section 2.3.

2.3. The algorithm: finding the large coefficients

In Section 2.2 we saw that the t -sparse hypothesis function that includes the t largest Fourier coefficients of the target function is a good hypothesis. We now show how to efficiently find those large coefficients. In Section 2.3.4 we will explain how to efficiently approximate the value of a coefficient.

2.3.1. Example. In this section we present an informal overview of the algorithm that finds the large coefficients by following an example. The example will be for $n = 4$, of a target function that has four large coefficients:

$$\begin{aligned}\hat{f}(0000) &= \frac{1}{2}, \\ \hat{f}(0001) &= \frac{1}{2}, \\ \hat{f}(0100) &= \frac{1}{2}, \text{ and} \\ \hat{f}(0101) &= -\frac{1}{2}.\end{aligned}$$

The Fourier coefficients listed above are all the non-zero Fourier coefficients of $f_{x_1 \wedge x_3}$ defined in Eq. (6).

$$f_{x_1 \wedge x_3} = f(\vec{x}) = f(x_1, x_2, x_3, x_4) \stackrel{\text{Def}}{=} x_1 \wedge x_3. \quad (6)$$

Our goal is to find all the large coefficients of $f_{x_1 \wedge x_3}$. We assume the existence of an oracle, *EXIST*, that given a prefix, determines whether a large coefficient with that prefix exists. For example, for the prefix “1”, $EXIST(1) = \text{FALSE}$ since f has no large coefficient with the prefix “1”. Over the prefix “000”, $EXIST(000) = \text{TRUE}$, since both coefficient-names, “0000” and “0001” have the prefix “000”. As we will see in Section 2.3.3, a variation of this oracle can be naturally implemented in the membership query learning model.

We start the algorithm with the null prefix: $\alpha = \emptyset$. Since large coefficients exist, we get $EXIST(\emptyset) = \text{TRUE}$. We concatenate “0” to the prefix, resulting in $\alpha_0 = 0$. We call on the oracle with the new prefix and get: $EXIST(0) = \text{TRUE}$. Hence, we know that large coefficients whose coefficient-name starts with a “0” exists. We also concatenate “1” to the prefix, α , resulting in $\alpha_1 = 1$. Once again, we call on the oracle, this time with the prefix α_1 and get: $EXIST(1) = \text{FALSE}$. This signifies that *no* large coefficients with the prefix “1” exist. This single bit of knowledge narrows down the search in half; knowing this, we need not examine all the coefficients starting with “1” which means we need not investigate half the possible coefficients.

Since the prefix $\alpha_0 = 0$ yielded a positive response, we continue recursively with it, setting our new prefix, α to α_0 . We concatenate both “0” and “1” to the new prefix, α , yielding, $\alpha_0 = 00$ and $\alpha_1 = 01$ respectively. We call the oracle with each new prefix, and receive a TRUE response in both cases: $EXIST(00) = \text{TRUE}$ and $EXIST(01) = \text{TRUE}$. This indicates that the search for large coefficients must continue in both directions: large coefficients with both prefixes, “00” and “01”, exist.

The entire search process is depicted in figure 1, where a shaded nodes indicates that the *EXIST* oracle has been called on the prefix that marks the node and a positive (TRUE) response has been returned. White nodes indicate that a negative response (FALSE) will be returned by the *EXIST* oracle if it is called on those nodes. The nodes outlined with a heavy line indicate that the *EXIST* oracle has, indeed, been called on during the search process, and

Figure 1. Example search. A shaded node indicates that the *EXIST* oracle has returned a TRUE response over the prefix that marks the node, signifying that it is a prefix of a large coefficient. The boldly outlined white nodes indicate that the *EXIST* oracle has been called on the prefix that marks the node, and a FALSE response has been returned. White nodes that are not heavily outlined indicate that the *EXIST* oracle was not called on that node, but if it were, a negative response would have been returned.

that it returned a FALSE response, indicating no large coefficients with the coefficient-name that marks the node. White nodes that are not outlined with a bold line are ones that we did not have to query during the search process; if we would have queried them, a negative response would have been received from the *EXIST* oracle.

The search for large coefficients in any direction terminates upon receiving a negative response from the oracle, or upon encountering a prefix of size 4. As we can see in figure 1, such nodes are leaves. Leaves that received a positive response from the oracle are the large coefficients we are looking for.

In Section 2.3.3 we will formally present the search process. However, the crucial point of the algorithm, one we did not address so far, is the implementation of the *EXIST* oracle. We find this implementation to be infeasible (in polynomial time). We therefore define a somewhat different oracle by relaxing our requirements of the *EXIST* oracle or predicate. The performance of this new predicate, *TEST*, will satisfy us if $TEST(\alpha) = \text{FALSE}$ implies that no large coefficients with prefix α exist, just as we expect of *EXIST*. Unlike the predicate *EXIST*, we do *not* require a guarantee that $TEST(\alpha) = \text{TRUE}$ will necessarily signify that a large coefficient with prefix α exists.

The predicate *TEST* is obviously weaker than *EXIST*, yet it is sufficient for our needs: we can bound the number of false TRUEs¹ the oracle has so as to ensure a satisfactory polynomial runtime. Since *TEST* can be estimated in polynomial time, this new oracle adequately fulfills our requirements and we use it instead of *EXIST* one. We describe this new oracle in Section 2.3.2.

2.3.2. The *TEST* predicate. The purpose of the *TEST* predicate is to determine whether or not any of the coefficient-names of f with prefix α have an absolute value larger than θ . We will therefore refer to *TEST* as $TEST[f, \alpha, \theta]$. In this section we will show that this predicate can be efficiently estimated *and* that it is sufficiently strong to be our oracle in the efficient search for large coefficients. Note that if the *TEST* predicate is very permissive we might reach all the coefficients, in which case our running time will not be polynomial. We will first define the predicate, and show that it is sufficiently strong. We will then show how to estimate it efficiently.

We define *TEST* to be TRUE iff $\sum_{\beta} \hat{f}^2(\alpha\beta) \geq \theta^2$. Clearly, if there exists a β such that $\hat{f}(\alpha\beta) \geq \theta$, then $TEST[f, \alpha, \theta] = \text{TRUE}$ and $EXIST[f, \alpha, \theta] = \text{TRUE}$ as well. Unlike *EXIST*, $TEST[f, \alpha, \theta] = \text{TRUE}$ does *not* imply that there exists a large coefficient-name with prefix α . However, it can be shown that the number of prefixes α for which $TEST[f, \alpha, \theta]$ returns TRUE is bounded (a further discussion of the bound will follow below). The bound is such that the number of superfluous calls made is limited to allow for a polynomial runtime.

To aid us in the implementation of the *TEST* predicate, we formally define a “prefix function”, $f_{\alpha}(x)$, as follows,

$$f_{\alpha}(x) \stackrel{\text{Def}}{=} \sum_{\beta \in \{0,1\}^{n-k}} \hat{f}(\alpha\beta) \chi_{\beta}(x). \quad (7)$$

The function $f_{\alpha}(x)$ is a partial Fourier representation of f , including only those coefficients $\hat{f}(\gamma)$ of f whose corresponding coefficient-names starts with α , that is, $\gamma = \alpha\beta$. It can be shown (Kushilevitz & Mansour, 1993) that:

$$E_{x \in \{0,1\}^{n-k}} E_{y \in \{0,1\}^k}^2 [f(yx) \chi_{\alpha}(y)] = E_{x \in \{0,1\}^{n-k}} [f_{\alpha}^2(x)] = \sum_{\beta \in \{0,1\}^{n-k}} \hat{f}^2(\alpha\beta). \quad (8)$$

Using Eq. (8), we can formally state that $TEST[f, \alpha, \theta]$ computes whether

$$E_{x \in \{0,1\}^{n-k}} E_{y \in \{0,1\}^k}^2 [f(yx) \chi_{\alpha}(y)] \geq \theta^2, \quad \text{where } k = \|\alpha\|. \quad (9)$$

This implies that if there exists a coefficient whose absolute value is greater than θ and whose prefix is α , that is, $|\hat{f}(\alpha\beta)| \geq \theta$, then the expected value is at least θ^2 . This condition guarantees the correctness of our algorithm, namely, that when using this oracle, we will reach all the “large” coefficients of the target function. Since it is *not* true that when the expected value in Eq. (9) is greater than θ^2 , we necessarily have a coefficient-name with a prefix α , during the search we will possibly encounter other (not large) coefficients as well.

In order to ensure the algorithm is efficient, we would like to bound the number of recursive calls made to the *TEST* predicate. Claim 2 gives us the means to do so.

Claim 2. Let f be a Boolean function, and $\theta > 0$. Then,

1. At most $1/\theta^2$ values of z satisfy $|\hat{f}(z)| \geq \theta$.
2. For any k , $1 \leq k < n$, at most $1/\theta^2$ functions f_{α} with $\alpha \in \{0, 1\}^k$ satisfy $E[f_{\alpha}^2] \geq \theta^2$.

Proof: Using the assumption that f is a Boolean function and from Parseval's Identity for Boolean functions (Eq. (3)), we get

$$\sum_{z \in \{0,1\}^n} \hat{f}^2(z) = 1. \quad (10)$$

From Eq. (10), part (1) of this claim immediately follows.

Similarly, using the definition of f_α (Eq. (7)) combined with Parseval's Identity (Eq. (3)), we get:

$$E[f_\alpha^2] = \sum_{\beta \in \{0,1\}^{n-k}} \hat{f}^2(\alpha\beta).$$

Thus, if $|\hat{f}(\alpha\beta)| \geq \theta$ for some $\beta \in \{0,1\}^{n-k}$, then $E[f_\alpha^2] \geq \theta^2$. From the definition of f_α :

$$\sum_{\alpha \in \{0,1\}^k} E[f_\alpha^2] = E[f^2] = 1.$$

Therefore, at most $1/\theta^2$ functions f_α have $E[f_\alpha^2] \geq \theta^2$, which completes the proof of part (2) of the claim. $\square$

Claim 2 shows that for any prefix of certain size, i.e., when considering all α such that $\|\alpha\| = k$ for a given k , at most $1/\theta^2$ of these prefixes will satisfy $TEST[f, \alpha, \theta] = \text{TRUE}$. This bounds the number of recursive calls we will make by $O(n/\theta^2)$.

In *TEST* we would like to compute the expected value (see Eq. (9)), but in order to do so efficiently we settle for an approximation of its value, B_α . This can be done, for $\|\alpha\| = k$, as described in Eq. (11):

1. We randomly choose m_1 bit-vectors: $x_i \in \{0,1\}^{n-k}$.
2. For each x_i , we randomly choose m_2 bit-vectors: $y_j \in \{0,1\}^k$.
3. For each i and each j we query f , provided to us as a black-box, on $y_{i,j}x_i$, and receive $f(y_{i,j}x_i)$. This is why membership queries are required in this algorithm: to query f on many points with the *same* x_i .
4. We then compute the estimate, B_α as:

$$B_\alpha = \frac{1}{m_1} \sum_{i=1}^{m_1} \left[\frac{1}{m_2} \sum_{j=1}^{m_2} f(y_{i,j}x_i) \cdot \chi_\alpha(y_{i,j}) \right]^2. \quad (11)$$

To prove that B_α is a “good” approximation of $E_x[f_\alpha^2]$, we need to prove: (1) that B_α is a “good” approximation of $E_x[f_\alpha^2]$ if computed with the real values of $f_\alpha(x_i)$, (2) that $\sum_{j=1}^{m_2} f(y_{i,j}x_i) \cdot \chi_\alpha(y_{i,j})$ is a “good” approximation for $f_\alpha(x_i)$, and, (3) that even if we compute B_α with the approximation for $f_\alpha(x_i)$, it is still a “good” approximation of $E_x[f_\alpha^2]$. We will not show the details of the proofs here, and the interested reader is referred to Kushilevitz and Mansour (1993).

The *TEST* predicate compares the value of $E_{x \in \{0,1\}^{n-k}} E_{y \in \{0,1\}^k}^2 [f(yx) \chi_\alpha(y)]$ to θ^2 as in Eq. (9). When using approximations of the expected values, we can guarantee with high probability (complete discussion and proofs in Kushilevitz and Mansour (1993)) that if we make the comparison $B_\alpha \geq \theta^2/2$, every coefficient-name z satisfying $|\hat{f}(z)| \geq \theta$ will be output. This approximation also guarantees that with high probability, no coefficient whose absolute value is less than $\theta/2$ will be output, which, using Parseval's Identity bounds the number of coefficients the algorithm outputs to at most $4/\theta^2$.

On each invocation of *TEST*, the parameters we provide are θ , m_1 and m_2 , where θ is the largeness threshold and m_1 and m_2 are the parameters that determine how accurately we approximate the *TEST* predicates (as in Eq. (11)). The runtime of each invocation is $O(m_1 \cdot m_2)$.

Now that we have overcome the difficulty in efficiently implementing an estimation for our oracle, we are ready to outline the algorithm we use formally in Section 2.3.3.

2.3.3. Algorithm for finding large coefficients. We now briefly describe the algorithm that finds the “large” coefficients of a function, using the approximation of the *TEST* oracle described in Section 2.3.2. The algorithm receives as its two inputs a function f (a black-box it can query on various inputs, that is, we permit membership queries) and an interest threshold parameter $\theta > 0$. It outputs a list of coefficient-names that

1. includes all the coefficient-names whose corresponding coefficients are “large”, i.e., at least θ , and
2. does not include “too many” coefficient-names (no more than $4/\theta^2$).

The algorithm runs in polynomial time in both $1/\theta$ and n .

The basic idea of the algorithm is to perform a search in the space of the coefficient-names of the input function f . The search is conducted in the same way it was carried out in the example of Section 2.3.1; throughout the search we maintain a prefix of a coefficient-name, α of Section 2.3.1, and try to estimate whether *any* of its extensions can be a coefficient-name whose value is “large”. The search procedure that was introduced in the example of Section 2.3.1 is formally outlined in figure 2, where the initial call to the search procedure is performed with $\alpha = \emptyset$. The main difference between the example of the search procedure of the Section 2.3.1 and the one outlined in figure 2 is in the use of the weaker oracle: the

```

SUBROUTINE search( $\alpha$ )
  IF  $B_\alpha \geq \theta^2/2$  /* approximation of  $TEST[f, \alpha, \theta]$  */
  THEN IF  $||\alpha|| = n$  THEN output( $\alpha$ )
        ELSE search( $\alpha 0$ )
              search( $\alpha 1$ )

```

Figure 2. Subroutine search. The subroutine uses membership queries within the *TEST* predicate, thus requiring the use of the black-box f . The parameter $\theta \in (0, 1)$ is the “largeness” threshold: the subroutine will output coefficients larger than θ .

approximation of the *TEST* predicate described in Section 2.3.2. Using this algorithm we find the list of all large coefficient-names. In Section 2.3.4 we describe how to approximate the value of a known coefficient-name.

2.3.4. Approximating the value of a single coefficient. The algorithm presented in the previous section outputs the list of “large” coefficient-names, $\beta_1, \dots, \beta_s$ corresponding to the target function. By definition,

$$\hat{f}(\beta_i) = E[f \cdot \chi_{\beta_i}],$$

and it can be proven that a “small” sample will give a “good” approximation for $\hat{f}(\beta_i)$ with high probability (details in Kushilevitz and Mansour (1993)). Therefore, in order to approximate a coefficient $\hat{f}(\beta)$ for a given coefficient-name β , we can randomly sample x_i ’s and take the average value. The sampling is done by choosing the x_i ’s from the uniform distribution, and the estimate is:

$$a_\beta = \frac{1}{m} \sum_{i=1}^m f(x_i) \chi_\beta(x_i).$$

3. Experiments description

We have implemented the algorithm described in Section 2.3 and then substantiated the theoretical findings with empirical validation. We ran a series of experiments, each one involving the learning of a function. The input for each experiment consists of, in addition to the target function, f , the parameters θ , m_1 and m_2 . Let us briefly describe these parameters here.

The target function, f , serves as a “black box” into which we feed an input bit vector of length n and receive the result of the function as a response: a classification of $+1$ or -1 .

The parameter θ determines the threshold between “small” and “large” coefficients. θ controls the number of nodes we will reach in our traversal of the coefficient tree (the tree of calls, as in figure 1). It is a crucial parameter in determining how many recursive calls we perform and the number of coefficients we output. Therefore θ controls the accuracy of our hypothesis.

The parameters m_1 and m_2 determine how accurately we approximate the *TEST* predicate. Should we fail to approximate the *TEST* predicate accurately, *TEST* may return faulty, even random, results (as it will, for example, for a ludicrous choice of $m_1 = 1$ and $m_2 = 1$). An inaccurate approximation of the value in *TEST* (due to an inadequate choice of parameters) may cause the algorithm to lose on two counts: on one hand, we may skip interesting subtrees (ones that contain “large” coefficients at the leaves), causing the final hypothesis to be less accurate. On the other hand, it may also cause the algorithm to traverse subtrees that are “uninteresting” (containing no “large” coefficients at the leaves), leading to a substantial, yet superfluous, increase in the run-time. An intelligent choice of m_1 and m_2 is therefore indispensable. This issue is discussed in greater detail in Section 4.3.

Figure 3. Typical frequency plots and typical errors.

In the following sections, we present representative results of our experiments in the form of graphs that evaluate the output hypothesis of the algorithm on randomly chosen test points. We used a different random bit-vector generator to generate the test set in order to ensure that the algorithm will *not* benefit from learning any possible patterns in the training set bit-vectors. The two sample sets we generate for training and testing are different sets. There may be a very small overlap between the two sets, though that is highly unlikely.

The target function, f , returns a Boolean response, ± 1 , while the Fourier transform hypothesis, h , returns a real response. These graphs will therefore include two curves: the frequency of the values of the Fourier transform hypothesis, $h(x)$, when $f(x) = +1$, and the second curve for $f(x) = -1$, as in figure 3.

If the two curves intersect, their intersection represents the inherent error the algorithm makes. Errors occur in two cases: (1) the algorithm predicts a -1 response when the actual response is $+1$ (the lightly shaded area in figure 3), and (2) the algorithm predicts a $+1$ response, while the true response is -1 (the darker shade area in figure 3).

3.1. Experimental results

Representative results of the experiments we ran appear in this section and in the following two sections, Sections 4 and 5. We summarize some of the results of Sections 4 and 5 below. All the target functions in this section as well as in Section 4 have 40 or 41 Boolean input variables while the ones in Section 5 have 535 input variables.

In many cases we also present the actual runtime of the experiments, that is the *elapsed* time, so as to provide one with a general ideal of the “real”-time performance of the algorithm. When one considers these times, one must keep in mind that different experiments were run on multiuser machines, on different platforms: SUNs (on a wide variety of processors, ranging from SPARCstation2 to SPARCstation20) and SGIs (once again, on processors of widely differing capabilities: from MIPS R4000 to MIPS R10000, though most commonly MIPS R10000). Thus, even when running the same experiment with the same parameters, different elapsed times may be noted due to the load average on the

Figure 4. Results of learning decision trees of depth 5 and 3 with 41 variables. The 5-deep (3-deep) decision tree returns -1 about 50% (62.4%) of the time. The results shown above are for values $\theta = 0.03$, $m_1 = 100$ and $m_2 = 5600$ ($\theta = 0.08$, $m_1 = 100$ and $m_2 = 800$). Both graphs are totally disjoint, signifying 0% error. The actual runtime for the 3-deep decision tree was, on average, 5.5 minutes on an SGI R10000. For different seed values the variance in run-times was around 3 seconds and in all the cases 0% error rate was achieved.

machine. When running the same experiment using different seed values (allowing for different training sets, thus ensuring that the results received are not due to any particular training set), only small time differences are noted (usually well below 10%) and they are detailed where appropriate below.

- Figure 4 contains two graphs portraying the results for randomly constructed fully balanced decision trees of depth 5 and 3, respectively. (While the expected error is 50%, 5 leaves of the tree of depth 3 were labeled -1 and three labeled $+1$, accounting for the probability of -1 being 62%.)
- We have also experimented with functions constructed from XORs. For example, the results of learning the following function are in figure 5:

$$s(x) \stackrel{\text{Def}}{=} ((x_{32} \oplus x_8 \oplus x_{34} \oplus x_{13}) \wedge (x_{33} \oplus x_{20} \oplus x_9 \oplus x_1 \oplus x_{15} \oplus x_{39})) \\ \vee (x_{24} \oplus x_{10} \oplus x_{27} \oplus x_6 \oplus x_{34}).$$

Again, it is easy to see that the hypothesis yielded by the Fourier transform algorithm was free of error.

- Figures 10 and 6 include the results for DNF with 8 and 16 terms respectively, each term containing four literals.
- We have considered additional “logically” defined functions, such as the parity function. The parity function is defined as follows:

$$\mathbf{p}(x) \stackrel{\text{Def}}{=} \begin{cases} \text{TRUE} & \text{if } (\sum_{i=1}^n x_i) \bmod 2 = 1 \\ \text{FALSE} & \text{if } (\sum_{i=1}^n x_i) \bmod 2 = 0 \end{cases} \quad (12)$$

We experimented with a parity function of 41 variables, running the experiment with a choice of values of $\theta = 0.5$, $m_1 = 20$ and $m_2 = 20$, and receiving perfect results, that

Figure 5. Results of learning a randomly generated function of XORs combined with AND and OR. This function of 41 variables returns -1 62.84% of the time. The results shown above are for values of $\theta = 0.1$, $m_1 = 100$ and $m_2 = 500$. Eight coefficients are used to form the hypothesis. Note that the results are guaranteed to be error free. The actual runtime of this experiment was approximately 3 minutes on an SGI O_2 R5000 regardless of the seed used (runtime variance was below 5%—a few seconds) and the error stayed at 0% in all the cases.

is with 0% error. Only one coefficient name was found: the “all ones” coefficient with value -1 (the value of a “positive” result). That should not surprise us: the result of the function toggles between $+1$ and -1 as any of its 41 variables changes its value to (from) 1 from (to) 0. Since we count the number of ones modulo two, if that number changes by one, the result changes as well. The result is equally dependent on each of the variables. This also explains why once we have found the right coefficient we get perfect results. Note that despite the fact that the values of the algorithm’s parameters are extremely small, i.e., we sample so very few times, the results are perfectly accurate on all occasions with 100% confidence level in all cases. This choice of parameters resulted in a very short runtime, of approximately 2 seconds on a MIPS R5000. Regardless of the seed value used, the runtime remained practically the same (the variance is measured in tenths of a second) and the “all-ones” coefficient was the only non-zero coefficient detected in all cases.

- Figure 8 depicts the results for the majority function. A majority function of $n = 2m + 1$ variables is defined in Eq. (13) below.

$$\text{maj}(x) \stackrel{\text{Def}}{=} \begin{cases} \text{TRUE} & \text{if } (\sum_{i=1}^n x_i) \geq m + 1 \\ \text{FALSE} & \text{if } (\sum_{i=1}^n x_i) \leq m \end{cases} \quad (13)$$

4. Implementation enhancements

Our purpose was to arrive at an implementation of the Fourier transform algorithm for Boolean functions that produces an accurate hypothesis while still maintaining a reasonable runtime. In order to achieve this dual goal, we had to make many important optimizations and introduce various changes. In this section we discuss the more prominent enhancements, ones that were imperative in order to achieve an applicable algorithm for practical usage; that is, one providing good results while maintaining an acceptable runtime. Furthermore, we introduce the confidence level that the algorithm produces in addition to the predictions. The confidence level measures the likelihood that the prediction is correct.

4.1. Confidence levels

One of our most consistent and interesting empirical findings was that the distribution of the error versus the value of the algorithm's hypothesis closely resembles that of a bell shaped curve. Knowledge of the error distribution permits us to determine with a high confidence level (often 100%) the result for most instances. The confidence level serves as the indicator of how confident we are of our prediction. Thus, we not only have a measure of the overall error of the classifier, but an error estimation for each prediction.

A somewhat similar confidence level indicator can be extracted from a neural network. In the output level of a neural network (for a Boolean function) there is a single sigmoid with a particular threshold value. The closer the weighted sum of the inputs (to that level) is to the threshold value, the lower the certainty in the prediction of the neural net. It should be noted that this threshold cannot serve as a perfect indicator of the confidence level of the neural net due to a possible bias of the prediction.

Let us demonstrate the strength of this technique in our case. Consider the results of the 16-term DNF in figure 6. The graph on the upper side of the page depicts the empirical approximations of the results' distribution in the form of two curves: the frequency of the hypothesis's results when $f(x) = +1$ in the dotted line and of $f(x) = -1$ in the continuous line. From this graph we can obtain the *error distribution* plotted in the graph below. This graph represents how much error we can expect for each prediction of the Fourier transform's hypothesis: how *un-sure* we are of each prediction. It is calculated as the ratio between the amount of erroneous outcomes to the total number of appearances for the value. Note that the output of the hypothesis is a real number (as explained in Section 2.2.1) that is translated into a Boolean prediction by the *Sign* function (Section 2.2.2). For each real value, the ratio between the number of times the translated Boolean prediction was in error and the total number of appearances of that value defines the error rate. In order to approximate the error efficiently, we randomly select, from a uniform distribution, a sample over which we calculate the value of the hypothesis and compare it to the value of the target function as detailed in Section 3.

For each value outputted by the hypothesis, the *confidence level* is the ratio between the number of correct to the total number of prognostics made with that value. For example, if the algorithm's hypothesis outputs 0.3 (translated into "1" in Boolean terms by the *Sign* function), we know with an 83% confidence level that the prediction is, indeed, correct. If

Figure 6. 16 term DNF. This (randomly generated) DNF of 40 variables returns -1 about 61% of the time. The results shown above are for the values of $\theta = 0.02$, $m_2 = 12500$ and $m_1 = 100$. The hypothesis uses 186 non-zero coefficients. A total of 9.6% error was detected.

the algorithm outputs -0.9 as its prediction, we can virtually guarantee that the response is correct. Thus, although the total error level is over 9%, we can tell for each point how confident we are in our prediction and in most cases, that will be close to 100%. This is an indispensable tool for practical usage of the hypothesis.

4.2. Error detection on the fly—retry

During our experimentations we have noticed that at times our estimate B_α for $E[f_\alpha^2]$ may be inaccurate. An error in this estimate, which provides the result to the *TEST* predicate, can manifest itself in one of two ways. It may lead to the abortion of the transversal of a subtree that may contain leaves (i.e., coefficients) that should, in fact, be included in the algorithm’s hypothesis, thus decreasing the hypothesis’s accuracy. An error may also lead us to the traversal of “un-interesting” subtrees, one containing *no* heavy coefficients, thereby needlessly increasing the algorithm’s runtime without gaining any useful information. We have devised a method to detect many of these errors and correct them in an efficient manner.

The properties of the Fourier transform guarantee that $E[f_\alpha^2] = E[f_{\alpha 0}^2] + E[f_{\alpha 1}^2]$, therefore we expect that $B_\alpha \approx B_{\alpha 0} + B_{\alpha 1}$. Whenever this is not true, we conclude that at least one of our approximations is somewhat lacking, which can lead the algorithm to miss important coefficients at the leaves of the corresponding subtrees or to the superfluous traversal of a subtree. We remedy the situation by running the search procedure again on the respective children, i.e., we “*retry*” the node α .

This solution increases the probability of finding all the “large” coefficients. However, a brute force implementation may cost an inordinate amount of time since in case an inconsistency is found at a parent node, the two subtrees stemming at it will be traversed. This procedure may be extremely costly since one, or both, of the subtrees may have been previously visited. We therefore take care to traverse only subtrees that were, until now, unvisited. Since any discrepancies between the parent and its children are likely to be discovered—and corrected—as soon as they appear, there is no need to re-traverse a subtree we have already visited. The highlights of this procedure are briefly described in pseudo-code in figure 7.

In general, whenever we detect an error in the approximation of the *TEST* predicate, and choose to re-approximate its value, we increase the probability of getting the right result. We therefore implemented a multi-retry mechanism, enabling us to re-approximate *TEST* at any level as many times as we wish until we achieve a satisfactory approximation.

We demonstrate the usefulness of this approach with the example of the majority function. When learning a majority function of 41 Boolean variables using the parameters $m_1 = 100$, $m_2 = 800$ and $\theta = 0.08$, *without* the retry mechanism, we receive the results depicted by the dotted-dashed line in figure 8–left: 13.7% error. The shaded area in figure 8–left depicts the errors incurred when the algorithm was run without retries. The algorithm accumulated errors since without the retries it missed eight (of a total of 42) large coefficients. *With* the retries, all the correct coefficients were found². The results we received *with* the retries had 0% error. These results are presented in the dotted curve in figure 8–right.

The distribution of the results in figure 8 differs from that shown in the previous examples: here, the results are centered in *numerous* peaks. This is a direct consequence of the Fourier

```

while ( ( nof_retries > 0 ) AND (  $B_\alpha \not\approx B_{\alpha 0} + B_{\alpha 1}$  ) )
  nof_retries --;
  re-approximate(  $B_\alpha$  );
  if ( previously-unvisited(subtree stemming off  $\alpha 0$  ) ) then
    traverse( subtree stemming off  $\alpha 0$  ); /* includes
                                         * reapproximating  $B_{\alpha 0}$  */
  else
    re-approximate(  $B_{\alpha 0}$  );
  endif
  if ( previously-unvisited(subtree stemming off  $\alpha 1$  ) ) then
    traverse( subtree stemming off  $\alpha 1$  ); /* includes
                                         * reapproximating  $B_{\alpha 1}$  */
  else
    re-approximate(  $B_{\alpha 1}$  );
  endif
endwhile

```

Figure 7. Highlights of the retry pseudo-code.

Figure 8. Majority function. The Fourier transform results with retries, plotted in a dotted line on the right, show no errors. The Fourier transform results without retries, plotted in the dashed-dotted line on the left, show the errors shaded in black. Note the majority-function characteristic distribution of the results.

transform representation of the majority function, which consists of 42 *equal* coefficients—one coefficient for each singleton (a vector of the form 00..010..00) and an additional coefficient for the parity coefficient (the all-ones vector). Since all the coefficients are equal, let us label with a c_{maj} the value of each coefficient. For an input vector with z zeros (and $41 - z$ ones), we will have a contribution of $\pm|(2z - 41) * c_{\text{maj}}|$ to the result, where $2z - 41$ is obviously odd. The parity will contribute an additional $\pm c_{\text{maj}}$ (depending on whether z is odd or even), so that the total contribution to the final result is an even factor of c_{maj} . Since $c_{\text{maj}} = \binom{40}{20} \frac{1}{2^{40}} \sim 0.12$, we have peaks around factors of 0.24. The distribution

around the peaks is due to the fact we only approximate each coefficient and get a value close to $\binom{40}{20}2^{-40}$.

It is important to note that the time increase brought about by incorporating the retry mechanism is minimal. In the above example of the majority function, utilizing the retry mechanism brought about, a valuable decrease in error, while on average, it only has an increase of *under* 4% in the runtime: from an average runtime of 12 minutes and 26 seconds without using the retry mechanism, to an average run time of 12 minutes and 51 seconds when using the retry mechanism to re-approximate the *TEST* predicate once upon discovering an inconsistency.

Inconsistencies are detected from the bottom up, allowing for a speedy correction mechanism. When an inconsistency is identified, a possibility of retraversal of one of the child-subtrees exists. During this retraversal, coefficients that were not previously noted may be found, leading to the construction of a better hypothesis. The retry mechanism allows for as many re-approximations as the user wants. We chose to always use one retry. On the one hand, the cost of the mechanism is minimal (as in the example above). On the other hand, the added value of the first retry can be enormous.

The results for the 16-term DNF presented in Section 4.1 are those of a run without the retry mechanism. Using the retry mechanism, the error decreases by 30% from 9.6% to 6.9% when using the same parameters: $m_1 = 100$, $m_2 = 12500$ and $\theta = 0.02$. When using the retry mechanism, 228 non-zero coefficients were discovered, approximately 20% more than without using the retry mechanism. More examples of the advantage of the retry mechanism are summarized in Table 1 in Section 5, where the error rates recorded when the retry mechanism was activated are compared to the error rates obtained when the mechanism was not activated.

4.3. Determining the parameters

One of our aims is to determine the values of the different parameters, m_1 , m_2 and θ . The parameter θ determines the accuracy of the hypothesis function that is determined by the user; it is the threshold between large and small coefficients. This parameter is used within the search process for large coefficients, in the *TEST* predicate as explained in Section 2.3.2. We recall Eq. (9):

$$E_{x \in \{0,1\}^{n-k}} E_{y \in \{0,1\}^k}^2 [f(yx) \chi_\alpha(y)] \geq \theta^2,$$

that formally defines the computation performed by the *TEST* predicate. As detailed in the theoretical overview, in order to arrive at an efficient algorithm, this computation is approximated. The approximation, B_α , is achieved by approximating the two expectations, the first by a random sample of size m_1 and the second by a random sample of size m_2 , as formally stated below:

$$B_\alpha = \frac{1}{m_1} \sum_{i=1}^{m_1} \left[\frac{1}{m_2} \sum_{j=1}^{m_2} f(y_{i,j} \ x_i) \cdot \chi_\alpha(y_{i,j}) \right]^2.$$

The two parameters, m_1 and m_2 , therefore define the accuracy of the approximation of the predicate and must be chosen so as to ensure that, with high probability, the approximation is good. By good we mean that if $E_x[f_\alpha^2(x)] \geq \theta^2$ then with high probability, $B_\alpha \geq \theta^2/2$. The theoretical values of these parameters, as a function of θ , can be found in Kushilevitz and Mansour (1993). In this work we are interested in finding the lowest bounds that will lead good empirical results: ones that are both fast and accurate. We take a closer look at these two parameters.

We have determined empirically that choosing a value of $m_1 = 100$, is sufficient to receive a good approximation of the expected value over x_i . Regardless of the target function or value of θ chosen, we have found that increasing the number of samples for the approximation of this expected value yield a negligible improvement in the approximation. In fact, for higher values of θ (≥ 0.1), a smaller value of m_1 will produce an equally good approximation. The value of m_2 , however, cannot be chosen independently of θ .

Choosing the right value for m_2 is of great importance. We have noticed on more than one occasion that to a certain degree, *increasing* the value of m_2 actually *decreases* the overall running time. This is not necessarily obvious at first. One would think that any increase in the number of times we loop in the algorithm would only increase the running time. However, one must not forget that a larger value for m_2 means a more accurate approximation of the values we are searching, and therefore less chance of redundant recursive calls. The running time, of course, is linear in the number of recursive calls.

We can see this exemplified in figure 9 where we see the number of recursive calls increase drastically as m_2 *decreases*. In order to present figure 9, we learned the same 3 term DNF always using $\theta = 0.05$ and $m_1 * m_2 = 100000$. The trials differ in the specific values chosen in each trail for m_2 ; for example, when $m_2 = 500$, $m_1 = 200$.

Figure 9. Determining m_2 . Note that the number of recursive calls grows dramatically as the value of m_2 value decreases. For example, for $m_2 = 400$, the number of recursive calls is 14,433 compared with only 1,329 recursive calls for $m_2 = 500$.

We experimented with various values for m_2 . Values that are less than $1/\theta^2$ can yield random, or very inaccurate results. Our empirical studies indicate that $m_2 = 5/\theta^2$ provides a highly accurate approximation, whose calculation takes a satisfyingly short time, and that is the value we have chosen for m_2 .

4.4. *Special cases*

When $k = \|\alpha\|$ is either very small or very large, the values we choose for m_1 and m_2 are sometimes self-defeating. That is to say, when $k \sim n$ (n is the number of variables in the target function), we still loop m_1 ($\gg 2^{n-k}$) times, though often the looping supplies us with no new information. For example, when $k = n - 4$, in order to calculate the *exact* expectation value we only need to sample $2^4 = 16$ different values, while ordinarily we would have sampled m_1 random times when *always* $m_1 \gg 16$. Thus, we gain no new information, and only waste valuable time. The same holds for very small values of k , and the corresponding m_2 ($\gg 2^k$) values.

We therefore add the following feature: for small and large values of k we calculate exactly the expected value thus decreasing the algorithm's overall run time and adding to the accuracy of the final hypothesis.

4.5. *Determining the threshold*

Once the list of large coefficients is built and we compute the hypothesis $h(x)$, we still need to determine the threshold, a , to which we compare $h(x)$ (i.e., predict +1 iff $h(x) > a$). In the theoretical work it is assumed that $a = 0$, and, indeed, a priori one cannot deduce a better guess. In our experimentations we observed that fixing a different value for a , depending on the hypothesis $h(x)$ we calculated, can be extremely useful. In order to find the value of a we consider a number of random examples and determine the optimal a with respect to them. Once the error curve is found which can be achieved within a few iterations (which usually takes less than a few seconds to run), we can determine the best threshold of the Boolean decision we make.

For example, when trying to learn an 8-term DNF with the parameters $m_1 = 100$, $m_2 = 5600$ and $\theta = 0.03$ we receive the results depicted in figure 10. If we use the zero threshold we receive a total of 1.22% overall error. However, if we choose the threshold to be 0.32, we get a diminished error of 0.068%.

A nontrivial example for the application of this mechanism is portrayed in the case of the function D6000_A (see Section 5). This function returns a positive value 96% of the instances. For most seed values, the resulting hypothesis from the Fourier transform algorithm of this function using a 0 threshold will yield only positive values. In these cases the error will be close to 4%. Using the appropriate threshold value, this feature is detected and the total error is reduced, sometimes by as much as 80%. The results of 20 runs with different seed values are depicted in figure 11.

Figure 10. 8 term DNF. This (randomly generated) DNF of 40 variables returns -1 about 43% of the time. The results shown above are for the values of $\theta = 0.03$, $m_2 = 5600$ and $m_1 = 100$. The hypothesis used 112 non-zero coefficients. The error detected for this function calculated with two thresholds as explained in Section 4.3 was 1.2% and 0.07%.

5. Results on real testbeds

Our motivation to implement the theoretical Fourier transform algorithm stemmed from a very practical origin: we were approached by a representative of a commercial firm (who requested to remain anonymous) with a very urgent need on their part. They wanted to be able to reproduce a state-free controller, that is, to perform reverse-engineering of the controller. They could only view the controller as a black-box that accepts numerous binary inputs and outputs a binary output. They were interested in the Fourier transform algorithm, and with that scenario in mind, we set out to implement it in a manner that would enable them to run it on their very real problem.

We implemented the original algorithm, and we started running it on very simple examples (single-term DNF), gradually increasing the level of difficulty of the functions used as inputs to include decision trees, more complex DNFs and other functions. During this process, we augmented the algorithm to include the enhancements described in previous sections in order to make it a useful tool for many different settings.

Once we reached a stable version of the algorithm, we asked our contact in the firm to supply us with a batch of functions that they attempted to represent. We picked the four most complex functions amongst them (usually ones that were dependent on the definitions of many other, simpler functions) and run our algorithm on them. It should be noted that we run the algorithm as-is, without making *any* changes in it and choosing the parameters according to the rules described previously (Section 4).

Figure 11. Effect of threshold values for D6000.A. The lightly shaded bars depict the error rates of the algorithm when the 0 threshold value was used. The darker shaded bars correspond to the error produced when the optimal threshold value was utilized for the different runs. On average, the error observed when the 0 threshold value was used was 3.5%. Using the optimal threshold value reduced the error rates to an average of 1.5%: an average reduction to less than half the original error.

Each one of the four test-functions had 535 Boolean inputs—an order of magnitude more than any function we have used as input to our algorithm beforehand. One function (D4283.A) is a large DNF consisting of 53 terms, each one containing between 11 and 2 literals. The other three functions are a combination of DNFs using other functions. The results of the runs using the parameters $m_1 = 100$, $m_2 = 2000$ and $\theta = 0.05$ are summarized in Table 1. The running times, error rates and the number of coefficients discovered are the averages over 20 runs of the experiment with different seed values⁴. Remember that each error rate is calculated over a different test set, generated specifically for this purpose. Each test set is generated using a random bit vector generator different than the one used to generate the training set in order to minimize the chances of any overlap between the training and test sets. The values of the error rates that are averaged out may have been recorded for different threshold values. The experimental standard deviation is recorded alongside the error rates. The number of coefficients discovered is rounded to the nearest

Table 1. Real testbed result summary.

Function	# of coefficients discovered	Approximate runtime (in hours)	The error rate (%)	Error rate without the retry mechanism (%)
D2650	5 (5–7)	$7\frac{2}{3}$	0.6 ± 0.03	0.6 ± 0.03
D5000_B	20 (15–23)	2.5	1.9 ± 0.05	2.0 ± 0.05
D4283_A	23 (22–25)	36.8^3	4.2 ± 0.08	4.4 ± 0.07
D6000_A	19 (13–26)	2.4	1.5 ± 0.07	1.5 ± 0.07

Table 2. Real testbed result summary with $\theta = 0.1$, $m_1 = 100$ and $m_2 = 500$.

Function	Runtime with the retry mechanism	Error rate with the retry mechanism (%)
D2650	5.9 mins $\pm$ 5 secs	0.61 ± 0.04
D5000_B	20.2 mins $\pm$ 49 secs	4.0 ± 0.1
D4283_A	2.1 hrs $\pm$ 5 mins	7.6 ± 0.3
D6000_A	18.6 mins $\pm$ 45 secs	4.0 ± 0.1

integer. In the same column, the range of the (maximal and minimal) number of discovered coefficients appears in parenthesis.

Noting that *no* further changes were made to the algorithm prior to the runs on the “real” testbeds, that the parameters were chosen in accordance to the standard rules, and considering that these are “real” test sets, the error rates, ranging from under 1% to 4% are encouraging to extremes. Further refinements are, of course, possible. For example, if we run D2650 with the parameters $m_1 = 100$, $m_2 = 500$ and $\theta = 0.1$, we will find the same 5 coefficients in well under than 2 hours and still get 0.6% error.

In Table 2 we have the results of the same experiments ran with different parameters; the coefficient largeness threshold, θ was chosen to be 0.1. Subsequently, according to our guidelines for choosing the approximation parameters, m_1 was chosen to be 50 and m_2 was chosen to be 500. A higher value of θ means that the hypothesis produces will not be more accurate than the one yielded by a lower value of θ . It can only be as good or less accurate since less coefficients are included in it. What we gain by raising the interest threshold is that the algorithm runtime will be significantly lower. These results are summarized in Table 2. There is a tradeoff between runtimes and error rates. However, the substantially higher run times of Table 2 would not be considered very long for important problems, especially at the cost of 2–3 times the error (compare to Table 1).

The company found our results encouraging, and planned to pursue this direction. Unfortunately, the project for which it was intended has been cancelled.

6. Comparison to C4.5

In this section we present a comparison of the Fourier transform algorithm to another algorithm: Quinlan’s (1993) C4.5 algorithm. The Fourier transform algorithm is set in the Membership Queries model, while C4.5 is not. We therefore need to explain how the comparison was made.

The Fourier transform algorithm requires the membership queries in order to approximate the value of the $TEST(p)$ predicate that determines whether a large coefficient with the prefix p exists, as detailed in Section 2.3.3. We have recorded the list of bit-vectors that the Fourier transform algorithm queried on each of its run, along with the target function’s value over the bit-vectors. This list was to be the input for a C4.5 run.

When the Fourier transform algorithm requires a large sample for an accurate $TEST$ predicate approximation, the bit-vectors list can be rather large. Each approximation requires $m_1 \cdot m_2$ bit-vectors. As detailed in Eq. (11), every m_2 bit-vectors share a common suffix of size $n - k$, with m_2 randomly selected prefixes of size k . Due to technical problems, the version of the C4.5 algorithm we used could not, in most cases, handle a large training sample such as the one recorded by the Fourier transform algorithm. We therefore, recorded only m_1 bit-vectors for each $TEST$ value approximation, each one with a different suffix. Since the bit-vectors are randomly chosen, the information loss is minimal. Only for the parity-function case, the training set recorded by the Fourier transform algorithm was small enough to be recorded in its entirety.

Using the training set recorded by the Fourier transform algorithm as input for the C4.5 algorithm, we ran both algorithms over four different functions, all of 41 input variables. Those functions are:

1. Parity function.
2. Majority function.
3. A 3-deep decision tree.
4. A 16 term DNF.

We then run the Fourier transform algorithm over the functions mentioned in Section 5, the real-testbed functions, with the lenient parameters of $m_1 = 50$, $m_2 = 60$, $\theta = 0.3$ for D2650, D6000_A and D5000_B, and $m_1 = 50$, $m_2 = 500$, $\theta = 0.1$ for D4283_A. Each function has 535 variables, and once again we recorded the bit vectors the Fourier transform algorithm used and gave them as input to the C4.5 algorithm. Note that the C4.5 runtimes include the time it take the C4.5 algorithm to load the training set file.

The results are summarized in Table 3. Each entry contains the result of 20 different learning trials of the algorithms over the same function, differentiated by the training set, that is determined by the random seed value chosen.

Table 3. Comparison between the Fourier transform algorithm and the C4.5 algorithm.

Function	FT error %	C4.5 error %	FT run time	C4.5 run time
parity	0 ± 0	50 ± 0.2	2 secs $\pm$ 0 secs	2.4 mins $\pm$ 7 secs
majority	2 ± 3	30.6 ± 3	12.6 mins $\pm$ 4 secs	4.9 mins $\pm$ 25 secs
3-deep DT	0 ± 0	0 ± 0	5.5 mins $\pm$ 3 secs	45 secs $\pm$ 5 secs
16 term DNF	16.6 ± 0.3	0.63 ± 0.05	1.9 hrs $\pm$ 2 mins	10.4 mins $\pm$ 2 mins
D2650	0.61 ± 0.03	0.51 ± 0.15	5.8 mins $\pm$ 7 secs	9.6 mins $\pm$ 19 secs
D5000_B	4.0 ± 0.1	2.4 ± 0.6	2.3 mins $\pm$ 0.6 secs	4.9 mins $\pm$ 16 secs
D4283_A	7.6 ± 0.3	0.18 ± 0.13	2.1 hrs $\pm$ 5 mins	30 mins $\pm$ 2 mins
D6000_A	3.9 ± 0.1	3 ± 0.6	2.2 mins $\pm$ 0.4 secs	4.7 mins $\pm$ 11 secs

Some functions, like the parity or majority functions, are very “hard” for the C4.5 algorithm that produces a decision tree as its hypothesis, since only a full tree will produce a good hypothesis. As we can see in Table 3, these functions are quickly and very accurately learned by the Fourier transform algorithm; no error was ever detected in the parity function trials, and in most cases none was detected for the majority function, either⁵.

The 3-deep decision tree was easily and accurately learned by both algorithms, though the C4.5 algorithm was faster.

The 16-term DNF, was quickly and accurately learned by the C4.5 algorithm. The Fourier transform algorithm was significantly slower in this case, and error has incurred. However, it should be noted that rather “lax” threshold parameters were used to speed up the runs ($\theta = 0.04$ instead of 0.02 that we used in the 40-bit 16-term shown before). Stricter parameters would result in more accurate results, but at the run-time cost. This is a general point: the Fourier transform algorithm can be as accurate as the user would like it to be, where the cost would be reflected by an increased run-time.

On the real test beds, in one case we received almost identical error rates for both algorithms, with running time slightly in favor of the Fourier transform algorithm. In the other case, the C4.5 algorithm performed significantly better in both run time and hypothesis accuracy.

7. Conclusions

In this work we have implemented the Fourier transform algorithm and showed that it is a useful practical tool, as well as a powerful theoretical technique. We invested a considerable amount of time in optimizing the algorithm, and mentioned in this paper the major points and methodologies we have developed that dramatically reduced the error and runtime, thereby enhancing the overall behavior of the algorithm. The implemented algorithm runs in a reasonable amount of time and is able to recover quite successfully various functions. Most notable is the algorithm’s performance on the “real” testbeds, of very large functions. Without an further enhancements, the algorithm performed extremely well on them.

We have shown that the algorithm naturally derives a confidence parameter. This parameter enables the user in many cases to conclude that the prediction received is accurate with extremely high probability, even if the overall error probability is not negligible.

Compared to other algorithms, the Fourier transform algorithm has *no* restrictions whatsoever on the class of Boolean functions it can learn. A drawback of the algorithm is its requirement to sample many points and query the function on them in order to have an accurate estimation of the predicate it evaluates, which reflects directly on its runtime.

Acknowledgments

This research was supported in part by The Israel Science Foundation administered by The Israel Academy of Science and Humanities and by a grant of the Israeli Ministry of Science and Technology. The authors would like to give special thanks to Prof. Jude Shavlik for his many helpful suggestions and comments as well as for his encouragement and patience.

Notes

1. We say that there is a *false* TRUE over a node marked with α if $EXIST(\alpha) = \text{TRUE}$ although the target function, f , has no large coefficients with the prefix α .
2. An additional two coefficients were found. However, their values were so small that their contribution was negligible and could not change the results and amount to an error.
3. The particularly large running time should be noted for D4283_A and, to some measure, for D2650. It is caused by swapping, as indicated by the percentage of idle time during the run. As we increased the memory the program was allowed to use, the program run time decreased, since there were less swaps. However, we did not have a machine with enough memory to be able to allow the program to consume as much memory as it needed and therefore recorded the runtimes as in Table 1.
4. Only 5 runs in the case of D4283_A.
5. Missing even one coefficient cause some error in the hypothesis in the majority function case, as explained in Section 4.2. Since the Fourier transform algorithm is probabilistic, that is liable to happen over a large number of runs.

References

- Angluin, D. (1987). Learning regular sets from queries and counterexamples. *Information and Computation*, 75, 87–106.
- Bellare, M. (1992). A technique for upper bounding the spectral norm with applications to learning. In *5th COLT (Workshop on Computational Learning Theory)*, number 5 (pp. 62–70).
- Blum, A., Furst, M., Jackson, J., Kearns, M., Mansour, Y., & Rudich, S. (1994). Weakly learning DNF and characterizing statistical query learning using fourier analysis. In *The 26th Annual ACM Symposium on Theory of Computing* (pp. 253–262).
- Furst, M. L., Jackson, J. C., & Smith, S. W. (1991). Improved learning of AC^0 functions. In *4th COLT (Workshop on Computational Learning Theory)* (pp. 317–325).
- Golden, R. M. (1996). *Mathematical Methods for Neural Network Analysis and Design*. MIT Press.
- Jackson, J. (1994). An efficient membership-query algorithm for learning DNF with respect to the uniform distribution. In *Proceedings of the 35th Symposium on Foundations of Computer Science* (pp. 42–53).
- Kushilevitz, E. & Mansour, Y. (1993). Learning decision trees using the fourier spectrum. *Siam Journal on Computing*, 22(6), 1331–1348. Earlier version appeared. In *Proceedings of the 23rd Annual IEEE Symposium on Foundations of Computer Science*, 1991.
- Linial, N., Mansour, Y., & Nisan, N. (1993). Constant depth circuits, fourier transform, and learnability. *Journal of the ACM*, 40(3), 607–620. Earlier version appeared in FOCS 1989.
- Mansour, Y. (1994). “Learning Boolean functions via the fourier transform” *Advances in Neural Computation*. Kluwer Academic Publishers.
- Mansour, Y. (1995). An $O(n^{\log \log n})$ learning algorithm for DNF under the uniform distribution. *Journal of Computer and Systems Sciences*, 50(3), 543–550.
- Nix, D. A. and Weigend, A. S. (1995). Learning local error bars for nonlinear regression. *Advances in Neural Information Processing Systems 7 (NIPS*94)*, G. Tesauro, D. S. Touretzky, & T. K. Leen (eds.), Cambridge, MA: MIT Press.
- Quinlan, J. R. (1993). *C4.5: Programs for Machine Learning*. Morgan Kaufmann.

Received July 2, 1996

Accepted November 25, 1998

Final Manuscript May 25, 1999