

Learning Matrix Functions over Rings

Nader H. Bshouty

Dept. Computer Science

University of Calgary

2500 University Drive NW

Calgary, AB, Canada T2N 1N4

Email: bshouty@cpsc.ucalgary.ca

Christino Tamon

Dept. Mathematics and Computer Science

Clarkson University

P.O. Box 5815

Potsdam, NY 13699-5815, U.S.A.

Email: tino@sun.mcs.clarkson.edu*

David K. Wilson

Dept. Computer Science

University of Calgary

2500 University Drive NW

Calgary, AB, Canada T2N 1N4

Email: wilsond@cpsc.ucalgary.ca

Abstract

Let R be a commutative Artinian ring with identity and let X be a finite subset of R . We present an exact learning algorithm with a polynomial query complexity for the class of functions representable as

$$f(x) = \prod_{i=1}^n A_i(x_i)$$

where for each $1 \leq i \leq n$, A_i is a mapping $A_i : X \rightarrow R^{m_i \times m_{i+1}}$ and $m_1 = m_{n+1} = 1$. We show that the above algorithm implies the following results.

1. Multivariate polynomials over a finite commutative ring with identity are learnable using equivalence and substitution queries.
2. Bounded degree multivariate polynomials over \mathbb{Z}_n can be interpolated using substitution queries.
3. The class of constant depth circuits that consist of bounded fan-in *mod* gates, where the modulus are prime powers of some fixed prime, is learnable using equivalence and substitution queries.

Our approach uses a decision tree representation for the hypothesis class which takes advantage of linear dependencies. This paper generalizes the learning algorithm for automata over fields given in [BBBKV96].

Keywords: Exact learning; Automata over rings; Multivariate polynomials over rings; Interpolation.

*Work partly done while the author was a student at Dept. Computer Science, University of Calgary.

1 Introduction

The learnability of multiplicity automata over the field of rationals using equivalence and substitution queries was shown to be possible by Bergadano and Varricchio [BV94]. Their algorithm was a direct generalization of Angluin’s algorithm for learning deterministic finite automata [A87]. Using algebraic techniques, Beimel et al. [BBBKV96] found a simpler algorithm for the above class which led to the discovery of additional learnable concept classes. These new classes include multivariate polynomials over finite fields, bounded degree multivariate polynomials over infinite fields and the *XOR* of terms. This new algorithm also provides a better query complexity for some other classes already known to be learnable such as decision trees and polynomials over $\text{GF}(2)$.

The algorithm in [BBBKV96] works for many classes of functions with a fixed length input, i.e., $f : \Sigma^n \rightarrow \mathcal{K}$ where Σ is the input alphabet and \mathcal{K} is a field. We present an algorithm that generalizes the learning of these types of functions to functions over rings that contain a finite sequence of ideals, i.e., we show how to learn some functions that are based on an algebraic structure that is less restrictive than a field.

Our algorithm uses a target class of matrix functions defined as follows. Let R be a commutative Artinian ring with identity and let X be a finite subset of R . A function $f : X^n \rightarrow R$ is called a *matrix function* if it can be written as

$$f(x) = \prod_{i=1}^n A_i(x_i)$$

for some set of mappings $A_i : X \rightarrow R^{m_i \times m_{i+1}}$ with $m_1 = m_{n+1} = 1$. An assignment $x = (x_1, \dots, x_n) \in X^n$ determines which matrices, $A_i(x_i)$, should be multiplied together to give a value for $f(x)$. The use of matrices to represent functions is well studied, particularly automata, and, in fact, a matrix based representation is also used in [BV94] and [BBBKV96]. Our representation is tailored to the representation of functions with fixed length inputs which is more restrictive than a matrix representation of automata where inputs have variable length.

Our learning algorithm uses a hypothesis class, based on decision trees, that allows us to take advantage of linear dependencies. Intuitively, each level of the tree can have a bounded number of *independent* nodes that continue to the next level while the remaining nodes are all linearly *dependent* on the values of the leaves that the independent nodes lead to. In combination with this hypothesis class our algorithm uses tables, that contain labeled examples, to learn the target matrix function. These tables are very similar to the observation tables introduced in [A87] and used, subsequently, in [BV94].

We show our algorithm implies the following results.

- Multivariate polynomials over a finite commutative ring with identity are learnable using equivalence and substitution queries.
- Bounded degree multivariate polynomials over \mathbb{Z}_n can be interpolated using substitution queries¹.

The former follows after we show how to represent multivariate polynomials as matrix functions. The latter follows using a generalization of Schwartz’s Lemma [S80] after a bound on the number of possible zeros of any target has been established (Schwartz’s Lemma [S80] outlines a probabilistic technique to determine if an unknown polynomial is equivalent to zero). We note that whether or

¹We focus on the case when n is a composite number.

not our algorithm runs in polynomial time is dependent upon the particular application. We also give another application that does not require the full generalization of our algorithm.

- The class of constant depth circuits that consist of bounded fan-in *mod* gates, where the modulus are prime powers of some fixed prime, is learnable from equivalence and substitution queries.

The remainder of the paper is organized as follows. We begin with a definition of the exact learning model followed by a brief overview of a paper [BBV96] that provides some intuition about matrix functions. Following this are the algebraic definitions necessary to describe our generalization to restricted rings. Next we define our hypothesis class, *decision programs*, and then establish their equivalence with matrix functions. We then describe our algorithm, establish its correctness and then give the applications described above.

We remark that the algebraic definitions in section 4 are central to showing our algorithm works for functions over restricted rings. It may be easier, on a first reading of the paper, for the reader to skim this section and read sections 5 and 6.1-6.3 under the assumption that the target is defined over a field.

2 The Learning Model

Our learning algorithm is set in the exact learning model which means it has access to two types of queries which answer questions about an unknown target formula f . The first type of question is a substitution query, $SuQ(x)$, which takes as input a member of the target domain, some $x \in X^n$ and outputs $f(x)$. The second type of question is an equivalence query, $EQ(h)$, which takes as input some hypothesis h from a hypothesis representation class H and answers YES if h and f are equivalent, otherwise, a counterexample, $c \in X^n$, is returned such that $f(c) \neq h(c)$. More information about this model can be found in [A88]. We note that substitution queries are a generalization of membership queries.

3 A Review of Automata Related Learning

As mentioned before, the paper [BV94] demonstrated that multiplicity automata are learnable in the exact model. Using this result, Bshouty et al. [BBV96] showed that many other classes admit representations that imply their learnability using the algorithm for multiplicity automata. The intuition behind our learning algorithm and choice of hypothesis class are partially based on the representations presented in [BBV96] so we briefly outline some of the contents of this paper for motivational purposes.

First we define the matrix representation of \mathcal{K} -Automata where \mathcal{K} is a field. Let $\mathcal{J} \subseteq \mathcal{K}$ where $\mathcal{J} = \{\alpha_1, \dots, \alpha_r\}$ and let \mathcal{J}^* be the set of all strings over \mathcal{J} . Let $\mathcal{A} = \{A_{\alpha_1}, \dots, A_{\alpha_r}\}$ be a set of $m \times m$ matrices over \mathcal{K} . Let $AU(\mathcal{J}, \mathcal{K})$ be the set of all functions $f : \mathcal{J}^* \rightarrow \mathcal{K}$ where there exists a row vector $\lambda \in \mathcal{K}^{1 \times m}$, a column vector $\gamma \in \mathcal{K}^{m \times 1}$ and a set of r matrices $\mathcal{A} = \{A_{\alpha_1}, \dots, A_{\alpha_r}\} \subseteq \mathcal{K}^{m \times m}$ such that

$$f(\alpha_{i_1}, \dots, \alpha_{i_s}) = \lambda A_{\alpha_{i_1}} \cdots A_{\alpha_{i_s}} \gamma$$

for all $(\alpha_{i_1}, \dots, \alpha_{i_s}) \in \mathcal{J}^*$. We can easily restrict the above functions to an ordered input of exactly n variables. Bergadano and Varrichio [BV94] showed that the above class with an ordered input of length n is learnable from equivalence and substitution queries in time polynomial in the number of variables n , $|\mathcal{J}|$ and the minimum sized m for any representation of the target in $AU(\mathcal{J}, \mathcal{K})$.

We now show that the above set of functions, $AU(\mathcal{J}, \mathcal{K})$, contains representations for multivariate polynomials which, given the above result of [BV94], implies that these are learnable in the exact model. Let $f(x) = \sum_{\sigma \in I} \alpha_{\sigma} x_1^{\sigma_1} \cdots x_n^{\sigma_n}$ be a multivariate polynomial over \mathcal{K} where I is a set of $n + 1$ tuples that contain the coefficients and variable degrees of each term. We will show that there exists a function in $AU(\mathcal{J}, \mathcal{K})$ that computes f . Note that the term $x_1^{\sigma_1} \cdots x_n^{\sigma_n}$ is computable by the function

$$A_1(x_1) \cdots A_n(x_n),$$

where $A_i(y) = [y^{\sigma_i}]$ is a 1×1 matrix function. Also note that if $h(x) = \prod_{j=1}^n A_j(x_j)$ then $ch(x)$, for some constant $c \in \mathcal{K}$, is computable by a function of the same structure (by multiplying all entries of A_1 by c). Finally, if f and g are computable by functions of this type then so is $h = f + g$. More specifically, if $f(x) = \prod_{j=1}^n A_j(x_j)$ and $g(x) = \prod_{j=1}^n B_j(x_j)$, and m_A, m_B are the maximum dimensions over all the matrices A_j and B_j (respectively), then define

$$C_j(y) = \begin{bmatrix} A_j(y) & 0_{M \times M} \\ 0_{M \times M} & B_j(y) \end{bmatrix}$$

where C_j is an $2M \times 2M$ matrix, $M = \max\{m_A, m_B\}$, whose top left corner $M \times M$ entries is filled with the matrix A_j and whose lower right corner $M \times M$ entries is filled with the matrix B_j . All other entries are zero including the possible extra zeros needed to pad any A_j or B_j whose dimension is less than M (not shown in the above representation of $C_j(y)$). Note that $h'(x) = \prod_{j=1}^n C_j(x_j)$ will be a $2M \times 2M$ matrix that is all zeros except for cells $(1, 1)$ and $(M + 1, M + 1)$ which will contain $f(x)$ and $g(x)$, respectively. To get the desired sum we can include in the multiplication (in the obvious way) a row and column vector of length $2M$ consisting of ones in positions 1 and $M + 1$ with zeros everywhere else. The above ideas can also be used to establish that the XOR of conjunctions and Sat j -DNF (DNF that have at most j terms satisfied by any input) are learnable in the exact model (this is an alternate method to the techniques used in [BBBKV96] where these results are also shown).

Because we know that the input has fixed length and the values of the input variables occupy a specific location in the input string, we do not need the full generality of the set $AU(\mathcal{J}, \mathcal{K})$ and our matrix functions take this into account. Using the graph based hypothesis, mentioned previously, we develop an algorithm that takes advantage of this structure.

4 Algebraic Definitions

We will assume familiarity with some basic algebraic structures such as Abelian groups, rings and fields. Let R be a commutative ring with identity. An *ideal* I of R is a subset of R that forms an additive subgroup and is closed under scalar multiplication by elements of R . The notation Ra , for $a \in R$, will stand for the set $\{ra | r \in R\}$ and the notation $Ra_1 + \cdots + Ra_m$, for $a_i \in R$, will stand for the set $\{\sum_{i=1}^m r_i a_i | r_i \in R\}$. The former is known as the principal ideal generated by a and the latter is known as the ideal generated by a_1, \dots, a_m . We let $\{0\}$ represent the trivial ideal generated when $a = 0$.

An R -*module* M is an Abelian group equipped with a multiplication from $R \times M$ to M such that the following is satisfied for all $r, s \in R$ and $a, b \in M$:

1. $r(a + b) = ra + rb$,
2. $(r + s)a = ra + sa$,

3. $(rs)a = r(sa)$,
4. $1a = a$.

Note that if R was a field then M would be a vector space over R . We call N a *submodule* of M if N is a subgroup of M and N is an R -module. An example of an R -module is R^n .

A ring is called *Artinian* if for any descending chain of ideals $I_0 \supset I_1 \supset I_2 \cdots$ there is an r such that $I_r = I_{r+1} = I_{r+2} = \cdots$. A ring is called *Noetherian* if for any ascending chain of ideals $I_0 \subset I_1 \subset \cdots$ there is an r such that $I_r = I_{r+1} = \cdots$. It is known that an Artinian ring is also Noetherian (see Theorem 3.2, page 16, [M]). In addition to R being a commutative ring with identity we will also assume that it is Artinian. We now formally define the rank of an ordered set of elements from an R -module.

Definition 1 Let M be an R -module and $v_1, \dots, v_m \in M$. We define $\text{rank}(v_1, \dots, v_m)$ to be the number of distinct sets in the sequence $Rv_1, Rv_1 + Rv_2, \dots, Rv_1 + \dots + Rv_m$. We say that the sequence v_1, v_2, \dots, v_m is independent if $\text{rank}(v_1, \dots, v_m) = m$.

We will use $l(M)$ to denote the maximal number of changes within any ascending chain of submodules of M . Notice that the notions “ideal of R ” and “submodule of R ” (where R is considered a submodule) coincide. The following two lemmas provide important relationships concerning the rank of members of R^n and the longest ascending chain in R^n . If $v \in R^n$ we let $v = \vec{0}$ represent the all zero vector, that is, all n coordinates of v are equal to zero. We will let $\{\vec{0}\}$ denote the submodule that only contains $v = \vec{0}$.

Lemma 4.1 If $v_1, v_2, \dots, v_m \in R^n$ and $v_1 \neq \vec{0}$ then $\text{rank}(v_1, \dots, v_m) \leq n \cdot l(R)$.

Proof We prove this using induction on n . For the base case $n = 1$, assume we have a sequence v_1, \dots, v_m with a rank of r . The sequence $Rv_1 \subseteq Rv_1 + Rv_2 \subseteq \dots \subseteq Rv_1 + \dots + Rv_m$ changes exactly $r - 1$ times. Since $\{0\} \subset Rv_1$ we see that $l(R)$ is at least r .

Now assume the claim holds for $n < k$. First we note that for $u_1, \dots, u_q \in R^n$

$$\text{rank}(u_1, \dots, u_q) = \text{rank}(u_1, \dots, u_i - \sum_{j < i} \lambda_j u_j, \dots, u_q),$$

for any i and any λ_j 's. Take any sequence $v_1, \dots, v_m \in R^k$ that is of rank m . Let $v_{i,j}$ be the j th coordinate of the vector $v_i \in R^k$. Consider the sequence $v_{1,1}, v_{2,1}, \dots, v_{m,1}$. Using the fact above, if

$$(1) \quad v_{i,1} = \sum_{j < i} \lambda_j v_{j,1}$$

then we can replace v_i by $w_i = v_i - \sum_{j < i} \lambda_j v_j$ without changing the rank of the sequence. We do this for each v_i that satisfies Equation (1). Let w_1, \dots, w_m be the resulting vectors. The number of w_i 's that are nonzero in the first coordinate is at most $l(R)$ by the base case. The removal of all w_i 's with a nonzero first coordinate leaves us with $m - l(R)$ vectors that induce a set of vectors in R^{k-1} . By induction the rank of these vectors is at most $(k - 1)l(R)$. Therefore, $m \leq kl(R)$. ■

Lemma 4.2 The length of the longest chain of submodules in R^n is n times the length of the longest chain of ideals in R , that is,

$$l(R^n) = n \cdot l(R).$$

Proof Let $l(R) = l$. Let $\{0\} = R_0 \subset R_1 \subset \dots \subset R_l = R$ be a strict sequence of ideals in R . Let $e_i \in R^n$ be the unit vector whose i th coordinate is 1 and the rest are 0. Consider the following submodule inclusions. We start with

$$R_0 \subset R_1 e_1$$

along with

$$R_1 e_1 \subset R_2 e_1 \subset \dots \subset R_l e_1.$$

Now for $2 \leq i \leq n$ and $1 \leq j \leq l - 1$ we build the following,

$$R e_1 + \dots + R e_{i-1} + R_j e_i \subset R e_1 + \dots + R e_{i-1} + R_{j+1} e_i$$

$$R e_1 + \dots + R e_{i-1} \subset R e_1 + \dots + R e_{i-1} + R_1 e_i.$$

This induces a chain of submodules of length $n \cdot l(R)$. Hence, $l(R^n) \geq n \cdot l(R)$.

Let $\{\vec{0}\} = M_0 \subset M_1 \subset \dots \subset M_L = R^n$ be a sequence of R -submodules. Let $v_i \in M_i - M_{i-1}$ for $1 \leq i \leq L$. We claim that v_1, v_2, \dots, v_L are *independent*. If not, then for some i , $v_i = \sum_{j < i} \lambda_j v_j \in M_{i-1}$, which is a contradiction. By Lemma 4.1, $L \leq n \cdot l(R)$. ■

5 Matrix Functions and Decision Programs

5.1 Matrix Functions

We recall the definition of matrix functions given in the introduction. Let X be a finite subset of R . A function $f : X^n \rightarrow R$ is called a *matrix function* if it can be written as $f(x) = \prod_{i=1}^n A_i(x_i)$, for some set of mappings $A_i : X \rightarrow R^{m_i \times m_{i+1}}$ with $m_1 = m_{n+1} = 1$. An assignment $x = (x_1, \dots, x_n) \in X^n$ determines which matrices, $A_i(x_i)$, should be multiplied together to give a value for $f(x)$. The size of a matrix function f is the maximum dimension of any matrix in any equivalent form of f that minimizes $\sum_{i=2}^n m_i$.

5.2 Decision Programs

We introduce a target representation called *decision programs*². These programs are a special type of decision tree that compute functions from X^n to R where X is a finite subset of R . The root of the program is on level 1 and there are $n+1$ levels in total. Level $n+1$ consists of sinks or leaves that are labeled with values from R . Only nodes on level $n+1$ are referred to as leaves. The remaining nodes found on levels 1 through n are classified as either independent or dependent. Independent nodes have outdegree $|X|$ with each exiting edge labeled by a distinct member of X . Dependent nodes have outdegree 0 and are labeled by a linear combination of some other independent nodes from the same level. Specifically, if v_j is a dependent node then $v_j = \sum_{t \in T} \lambda_t v_t$ where T holds the subscripts of the independent nodes on that level and $\lambda_t \in R$. All nodes have indegree exactly one except for the root. The *size* of a decision program is equal to the total number of nodes (independent, dependent, and sinks).

Given an assignment $x = (x_1, \dots, x_n)$, a decision program evaluates x in the following recursive manner. The computation to evaluate x starts at the root. If during the computation we reach a node v at level j then we do the following based on whether v is independent or dependent. If v is independent then we follow the edge labeled x_j to the next level. If v is dependent then

² A similar representation was also considered by Bergadano and Varricchio [BV95] under the name of *transition graphs*.

the computation continues using those independent nodes that are used to represent v . A new computation is started at each of these nodes using the input and the values returned are combined according to the linear combination of v . If v is a sink then we return the label of v . Notice that at any time if a dependent node is labeled by $0 \cdot v$, for any node v , then the value of the output required from following this path will be zero and no further computation is required. We mention this as we will use a special case decision program to represent the constant 0 function. This will be a decision program with every node on the second level labeled by $0 \cdot v_I$ where v_I is an imaginary vector.

5.3 Equivalence of Matrix Functions and Decision Programs

In this subsection we will show that a function is a matrix function if and only if it is computable by a decision program. First we show how to convert any decision program into an equivalent matrix function.

Theorem 5.1 *If P is a decision program computing a function f then f is a matrix function.*

Proof Suppose that f is computable by a decision program P . Assume that for each $1 \leq i \leq n$ the number of nodes at level i in P is m_i . Each node at level i in P computes a function over the variables $\{x_i, \dots, x_n\}$. Let $v_{i,1}, \dots, v_{i,m_i}$ be the nodes at level i in P and let $f_{i,1}, \dots, f_{i,m_i}$ be the functions computed at those nodes, respectively. We will show that f is a matrix function. Let $A_i(d)$, for $d \in X$, be an $m_i \times m_{i+1}$ matrix over R which we define as follows. All entries in A_i are zeros except the ones we will specify next. If $v_{i,j}$ has an outgoing edge to $v_{i+1,k}$ then entry $A_i(d)[j,k] = 1$. If $v_{i,j}$ has an outgoing edge to $v_{i+1,k}$ and this edge is labeled with $d \in X$ then again $A_i(d)[j,k] = 1$. If $v_{i,j}$ is a dependent node labeled with a linear combination $v_{i,j} = \sum_{w \in W} \lambda_w v_{i,w}$, where W is some set of independent nodes at level i , then we do the following. Suppose that for each $w \in W$, \hat{w} is the unique node for which the edge $(v_{i,w}, v_{i+1,\hat{w}})$ labeled d exists, i.e., $A_i(d)[w, \hat{w}] = 1$. Then we set $A_i(d)[j, \hat{w}] = \lambda_w$. Using these definitions, we notice that for $i = 1$ to n we have

$$\begin{pmatrix} f_{i,1} \\ \vdots \\ f_{i,m_i} \end{pmatrix} = A_i(x_i) \begin{pmatrix} f_{i+1,1} \\ \vdots \\ f_{i+1,m_{i+1}} \end{pmatrix}.$$

Using the above repeatedly, we obtain the theorem by taking $A_n(x_n)(f_{n+1,1}, \dots, f_{n+1,m_{n+1}})^T$ as the last mapping. ■

Now we show that every matrix function is computable by some decision program. The notation $l(R)$, used in the following theorem, was defined in section 4.

Theorem 5.2 *Any matrix function is computable by some decision program of size at most*

$$(|X| + 1) + |X|l(R) \sum_{i=2}^n m_i.$$

Proof A decision program for a matrix function $f(x_1, \dots, x_n)$ can be built as follows. First, if f is equivalent to the constant zero function then it is very simple to build a decision program with all leaves labeled zero that has the above size bound. If f is not equivalent to the constant zero function then we start building an $n + 1$ level decision program by placing a root node on level one and $|X|$ nodes on level 2. Each node on level 2 is connected to the root by one edge and each edge is labeled by one distinct member of X .

We proceed as follows for levels 2 to n . When at level j we determine how many nodes to put on the next level by determining the dependencies of the nodes that already exist on level j . If the process is at level j and there are w nodes then we can view these nodes on this level as a set of matrix functions

$$B_{j,1} \prod_{j=i}^n A_j(x_j), \dots, B_{j,w} \prod_{j=i}^n A_j(x_j)$$

where each $B_{j,k}$ is a $1 \times m_j$ matrix induced by the the assignment $(x_1, \dots, x_{j-1}) \in R^{j-1}$ that leads to the associated node. We order the functions associated with these nodes arbitrarily except for making sure that $B_{j,1}$ is not composed only of zeros. Now we check each function in order to determine if it is independent or dependent on the previous independent functions. If a function is linearly dependent on the previous independent functions then the node associated with the function is labeled accordingly. If a function is independent of all independent nodes before it then we create $|X|$ nodes on the next level connected to this associated node. Each connecting edge is labeled by one distinct member of X . When the above process reaches level $n + 1$, the leaf level, every node is labeled with the appropriate output value of the function. A straight forward inductive proof shows that the above construction results in a decision program that is equivalent to the given matrix function.

To get the size bound on this decision program we notice that the sequence of vectors

$$B_{j,1}, \dots, B_{j,w} \in R^{m_j}$$

can have at most $l(R)m_j$ linearly independent elements by Lemma 4.2. This implies that the sequence of linearly independent elements of the above set of functions is also $l(R)m_j$. Since each independent node accounts for $|X|$ nodes on the next level and the first two levels have $|X| + 1$ nodes, the result follows. ■

6 Learning Matrix Functions

This section contains a description of our algorithm for learning matrix functions followed by an analysis which establishes both its correctness and its complexity. Before these are given we define some additional notation and introduce a construct that plays an important part in our learning process.

6.1 Additional Notation

We now address some related notation that will be useful during the description of our learning algorithm. As learning progresses our algorithm will be adding nodes to the hypothesis decision program. We will let v_j^i denote the i th node added to the hypothesis on level j . We can view each node on level j as computing a function over X^{n-j+1} . We will use $h_j^i(x)$ for $x \in X^n$ to denote the output value of the hypothesis decision program when the computation begins at node v_j^i . Notice this computation is independent of the first $j - 1$ components of x . We will let $f_j^i(x)$ denote the output value of the target with an input composed of the prefix that leads from the root of the hypothesis to v_j^i concatenated with the last $n - j + 1$ components of x . Again, this computation is independent of the first $j - 1$ components of x . Finally, if v_j^i is an independent node we will use $v_j^i(x)$ to denote the node on level $j + 1$ that is reached following the edge labeled x_j that exits v_j^i .

6.2 Dependency Tables

The algorithm uses decision programs as its hypothesis class along with a set of related tables that are used to determine the dependencies of nodes on the same level. The algorithm will maintain one table for each level from 2 to n with rows labeled by prefixes of vectors from X^n and columns labeled by suffixes of vectors from X^n . Specifically, a table at level j will have each row labeled with a vector from X^{j-1} and each column labeled with a vector from X^{n-j+1} . For any level j the content of the table at the row labeled (x_1, \dots, x_{j-1}) and the column labeled (x_j, \dots, x_n) is $f(x_1, \dots, x_n)$. The entries for all tables are provided by substitution queries. The size of a table is the number of rows times the number of columns.

We now describe the relationship between a table and the nodes that exist on the same level in the hypothesis decision program. Each node on level j will be associated with a vector composed of the following. Let w_0 be the vector of length $j - 1$ that leads from the root of the hypothesis to the node v_j^i . If the labels of the columns of the table on level j are w_1, w_2, \dots, w_k then the vector associated with v_j^i is

$$(f(w_0w_1), f(w_0w_2), \dots, f(w_0w_k)).$$

This associated vector only appears as a row in the table on level j if it cannot be represented as a linear combination of the associated vectors for nodes that are above it in the table. The index of this row would be w_0 . If the associated vector can be expressed as a linear combination of the rows in the table then this node is labeled with this equation. We can designate which rows appear in this equation using the names of the nodes associated with these rows. These associated vectors determine the classification of each node in the existing hypothesis decision program.

We will show that an increase in the size of any table during the running of our algorithm indicates that progress is being made towards finding an equivalent representation of the target. A row is added to the bottom of a table any time an independent node is added to the hypothesis or an existing dependent node is found to be independent. When a column is added to the table it indicates that the dependency of an existing node is going to change, i.e., the label of a dependent node will change or a dependent node will become independent. Now we give the description of the algorithm.

6.3 The Learning Algorithm

An overview of the algorithm is as follows. The algorithm begins by building an initial hypothesis that is equivalent to the constant zero function. This initial hypothesis is artificially defined so that the initialization of the dependency tables can be performed by the main part of the algorithm. Once initialization is complete the algorithm repeats two main steps until an equivalent hypothesis is found. The first of these two steps is an equivalence query on the current hypothesis. The second step processes any counterexample returned by the preceding step. When this second step is performed due to a counterexample returned for the initial hypothesis an extra step is included to initialize the columns of each dependency table. We now describe the algorithm in detail.

For brevity of presentation, we use an artificial representation for the constant zero function. This will consist of a root with $|X|$ uniquely labeled edges leading to a separate dependent node on the next level. Each dependent node will be labeled with $0 \cdot v_I$ where v_I is an imaginary vector. Also, during the running of the algorithm, if a table has no rows yet and an associated vector with a particular node is all zero, then it can be labeled with $0 \cdot v_I$. The algorithm starts by setting $h \equiv 0$.

The algorithm now enters its main part. The first main step consists of asking an equivalence query using the current hypothesis h . If the query returns an answer of YES then the algorithm

outputs h and halts. If a counterexample c is returned then the algorithm proceeds to the second main step which uses c to modify the hypothesis. If c is the counterexample returned due to a query with $h \equiv 0$ then the algorithm will add a column to table T_j , $3 \leq j \leq n$ labeled c_j, \dots, c_n . This means each node that will be added in response to this first counterexample will now have an associated vector as defined in the last subsection.

The second main step employs a recursive procedure called *Process* which traces through the current hypothesis looking for an appropriate place of modification. The counterexample c and a possible point of modification, v_j^i , are the two arguments of this procedure. The processing of any returned counterexample c starts with a call of *Process* (c, v_1^1) and proceeds according to two cases based on whether the input node is independent or dependent.

Case 1: Node v_j^i is independent.

This node will not be modified further so the algorithm proceeds to the next level of the hypothesis by calling *Process* ($c, v_j^i(c)$).

Case 2: Node v_j^i is dependent.

This means v_j^i is labeled by some linear combination of independent nodes from that level, i.e., $v_j^i = \sum_{t \in \tau_j} \lambda_t v_j^t$ where τ_j contains the superscripts of the independent nodes on that level. The counterexample will show that either this linear combination is wrong or one of the functions h_j^t , $t \in \tau$, has an error in it. The algorithm first checks the functions that make up the linear combinations, i.e., it checks, using substitution queries, if $h_j^t(c) = f_j^t(c)$ (recall the notation of subsection 6.1). The first found disagreement results in a call to *Process* ($c, v_j^i(c)$). If no disagreements are found then the error is with the linear combination that currently labels v_j^i . A column is now added to the dependency table on this level labeled with the suffix (c_j, \dots, c_n) and additional table values are added using substitution queries. The algorithm now re-checks all labels of dependent nodes on this level. Notice this column witnesses that fact that at least one of these labels is incorrect. Each time a dependent node is found to be independent, a row is added to the bottom of the table on this level that corresponds to this node. If none of the nodes change to independent ones during the above then the processing is considered complete. The addition of an independent node creates $|X|$ nodes on the next level and the algorithm must determine the classification of these nodes. To complete the hypothesis decision program the algorithm uses another recursive procedure called *CleanUp*. For each new independent node, v_j^i , the algorithm calls *CleanUp* (v_j^i) which proceeds according to the value of j .

Case 1: $j = n$.

The algorithm will add $|X|$ leaves to the last level. Each new leaf will have one incoming edge from v_j^i labeled with a distinct element from X . If (a_1, \dots, a_{n-1}) is the prefix that leads to v_j^i then each new leaf is labeled by $f(a_1, \dots, a_{n-1}, x)$ if its incoming edge is labeled by $x \in X$.

Case 2: $j < n$.

The algorithm adds $|X|$ new nodes to level $j + 1$ in an arbitrary order each with an incoming edge from v_j^i labeled with a distinct element of X . Each new node is checked, in order, to see if it can be labeled by a linear equation of independent nodes on that level. If it cannot then a new row is added to the bottom of the table for this node. This is done before moving on to check the dependency of the next new node. If no nodes are found to be independent then the processing is complete. For any node v_{j+1}^k found to be independent, call *CleanUp* (v_{j+1}^k).

6.4 Analysis of the Algorithm

6.4.1 Properties of the Algorithm

We begin by proving several useful claims. We use T_j to represent the dependency table on level j .

Claim 1 *Every table contains rows of independent vectors at any time during the execution of the algorithm.*

Proof A node is classified as independent by the algorithm if its associated table vector cannot be expressed as a linear combination of the rows that exist in the table. The claim now follows from the facts that new rows are added to the bottom of the table and new columns do not affect independence. ■

Claim 2 *For $2 \leq j \leq n$, T_j contains at most $l(R)m_j$ rows.*

Proof As described previously, a node on level j defines a vector $\gamma \in R^{j-1}$ that leads to this node from the root. Suppose that the algorithm has placed rows in T_j that correspond to the assignments $\gamma_1, \gamma_2, \dots, \gamma_w$. Also assume that the columns of T_j are labeled with $\delta_1, \dots, \delta_s \in R^{n-j+1}$. We now view the target matrix function as the product of two matrices B and C . Each assignment $\gamma_i \in R^{j-1}$ defines a matrix $B(\gamma_i)$ of dimension $1 \times m_j$ and each assignment $\delta_i \in R^{n-j+1}$ defines an $m_j \times 1$ matrix $C(\delta_i)$.

$$\begin{aligned} f(x_1, \dots, x_n) &= A_1(x_1) \cdots A_n(x_n) \\ &= (A_1(x_1) \cdots A_{j-1}(x_{j-1})) (A_j(x_j) \cdots A_n(x_n)) \\ &= B(x_1, \dots, x_{j-1}) C(x_j, \dots, x_n) \end{aligned}$$

The i th row of T_j can be written as

$$B(\gamma_i) \left[\begin{array}{c|c|c|c} C(\delta_1) & C(\delta_2) & \cdots & C(\delta_s) \\ \hline & & & \end{array} \right]$$

Let $C_{T_j} = [C(\delta_1), \dots, C(\delta_s)]$ be the $m_j \times s$ matrix above. Since the row vectors

$$B(\gamma_1)C_{T_j}, B(\gamma_2)C_{T_j}, \dots, B(\gamma_w)C_{T_j}$$

are independent then so is the sequence

$$B(\gamma_1), B(\gamma_2), \dots, B(\gamma_w).$$

Now by Lemma 4.2, the number of elements in this sequence can be at most $l(R)m_j$. ■

Claim 3 *For $2 \leq j \leq n+1$ the number of nodes on level j is at most $l(R)m_{j-1}|X|$.*

Proof This follows from the above claim since the number of nodes on any level $2 \leq j \leq n+1$ is $|X|$ times the number of independent nodes on level $j-1$. ■

Claim 4 *For any fixed set of r rows in any table T_j , the number of times the linear combination of any dependent node on level j changes with respect to the r rows is at most $l(R)(r+1)$.*

Proof Consider some dependent node u_0 on some level with the following r table vectors $\{u_1, \dots, u_r\}$. Suppose that at some point during the running of the algorithm it is known that $u_0 = \sum_{i=1}^k \alpha_i u_i$. Consider the set of all linear combinations of u_0 in terms of vectors from T_j :

$$\Lambda = \{(\lambda_0, \lambda_1, \dots, \lambda_k) \mid \sum_{i=0}^k \lambda_i u_i = 0\}.$$

It is easy to verify that Λ is an R -module. The relation $u_0 = \sum_{i=1}^k \alpha_i u_i$ implies that $\tilde{\alpha} = (1, -\alpha_1, \dots, -\alpha_k) \in \Lambda$.

Now if some counterexample shows that $u_0 \neq \sum_{i=1}^k \alpha_i u'_i$, where u'_i is the new independent row vector corresponding to u_i after the addition of a new column, then

$$\Lambda' = \{(\lambda'_0, \lambda'_1, \dots, \lambda'_k) \mid \sum_{i=0}^k \lambda'_i u'_i = 0\} \subset \Lambda.$$

It follows that $(1, -\alpha_1, \dots, -\alpha_k) \notin \Lambda'$ and hence the inclusion is strict (note that Λ' is an R -submodule of Λ). So a sequence of updates on the linear combination of u_0 in terms of a fixed number of rows in T_j induces a strict descending sequence of R -modules. The length of such a chain is at most $l(\Lambda) \leq (r+1)l(R)$ by Lemma 4.2. ■

Claim 5 *The number of columns in T_j is at most*

$$|X|l(R)m_{j-1} \cdot l(R)\sum_{i=1}^{l(R)m_j} (i+1).$$

Proof By Claim 4, we know that each node can change its linear dependencies with respect to a fixed set of r independent nodes at most $(r+1)l(R)$ times. The value of r can be at most $l(R)m_j$ by Claim 2. In the worst case we assume that each node goes through as many changes as is possible for each possible number of independent rows. The current claim now follows from Claim 3 which states that the maximum number of nodes on level j is at most $|X|l(R)m_{j-1}$. ■

For each leaf L of h , there is a unique path $P_L = (v_1, v_2, \dots, v_{n+1})$, where v_1 is the root, $v_{n+1} = L$, and v_2, \dots, v_n are independent nodes. With each such unique path P_L , we can associate an assignment $b_L \in X^n$ where b_j is the edge label of (v_j, v_{j+1}) , for $j = 1, 2, \dots, n$.

Claim 6 *Suppose h is a hypothesis decision program of the algorithm. For any leaf L in h , $f(b_L) = h(b_L)$.*

Proof Follows by the construction of h by the algorithm. ■

6.4.2 Correctness of the Algorithm

We know that if the algorithm halts it will output an equivalent form of the target because the algorithm halts when the equivalence query returns an answer of YES. We must now argue that the algorithm makes progress towards finding an equivalent hypothesis to the target. We do this with the following lemma.

Lemma 6.1 *Let c denote the counterexample returned by an equivalence query using the current hypothesis h . A call of *Process* (c, v_1^1) will result in an increase in the number of columns of at least one dependency table associated with h .*

Proof We know that $f(c) \neq h(c)$ and we now argue that this disagreement is due to an incorrect labeling of a dependent node somewhere on a level between 2 and n . The disagreement will not be due to any independent node by Claim 1 nor will it be due to a labeling of a leaf by Claim 6. These two observations imply that the current contents of levels 1 or $n + 1$ are not the cause of error and neither is any independent node on levels 2 through n . This means that the error is due to an incorrect labeling of some dependent node. The procedure *Process* searches until it finds such a node and at this point a new column is added to the table on this level. ■

6.4.3 Query Complexity

Equivalence Queries As established above, each counterexample will result in an increase in the number of columns of some table. Letting $m = \text{size}(f)$ we get the following bound on the number of equivalence queries using Claim 5,

$$O(|X|(l(R))^4 m^3 n).$$

Substitution Queries These queries are used to label the leaves and direct the subroutine *Process* to a place of modification but these uses are small compared to their use in maintaining dependency tables and labels for dependent nodes. A bound on these for one level is provided by multiplying the number of columns a table may have by the total possible number of nodes that a level may have. These values follow by Claim 5 and Claim 3, respectively. A bound for the total number of substitution queries given the above is

$$O(|X|^2 (l(R))^5 m^4 n)$$

where $m = \text{size}(f)$.

6.4.4 Time Complexity

The only portion of the algorithm that is of concern for establishing a polynomial time algorithm is the procedure for finding the dependence of the vectors associated with the nodes in the hypothesis. If this procedure can be accomplished in polynomial time then the algorithm will run in polynomial time. The applications of the next section deal with finite integer rings for which the above problem is a polynomial time procedure (for example, see the second part of Lemma 4.1).

7 Applications

7.1 Polynomials over Finite Commutative Rings with Identity

We have already shown, in Section 3, how a multivariate polynomial can be represented by a matrix function. So we immediately have the following lemma.

Lemma 7.1 *Let R be a finite commutative ring with an identity. Let $f(x_1, \dots, x_n)$ be a multivariate polynomial over R . Then there is a matrix function $A(x_1, \dots, x_n) = \prod_{j=1}^n A_j(x_j)$ over R that computes f .*

A corollary of Lemma 7.1 is that if R is a finite commutative ring with identity then the class of multivariate polynomials over R is exactly learnable from equivalence and substitution queries.

7.2 Randomized Interpolation of Polynomials over \mathbb{Z}_n

The result from the previous section implies that polynomials over \mathbb{Z}_n are exactly learnable from equivalence and substitution queries. In this section we will show a randomized interpolation algorithm for multivariate polynomials over \mathbb{Z}_n , that is, we will show how to replace equivalence queries by substitution queries for this case. The idea is to use *random sampling*, i.e., to test whether $f \equiv h$, take sufficiently many random points to confirm that $f - h \equiv 0$. So equivalence queries are replaced with randomized zero-testing. In particular, we show a generalization of Schwartz's results [S80] for the ring \mathbb{Z}_n . For this, just as in Schwartz's result, we need to establish an upper bound on the number of zeros that a polynomial of certain degree can have over \mathbb{Z}_n .

Definition 2 For a polynomial $f(x)$, we let $z(f)$ be the set of all zeros of f , i.e., $z(f) = \{a : f(a) = 0\}$. Note that $|z(f)|$ is the number of zeros of the polynomial f . For a natural number n let $\delta(n)$ denote the minimal constant d such that there is a polynomial of degree d over \mathbb{Z}_n with nonzero coefficients that is equivalent to the zero polynomial. For a natural number n let $\rho(n)$ denote the smallest prime that divides n .

We start by noticing the following fact relating $\delta(n)$ and $\rho(n)$.

Fact 7.1 $\delta(n) = \rho(n)$, for all $n \geq 2$.

Proof We have two cases depending on whether n is prime or not. If n is a prime then $\delta(n) = n$ since $x^n - x$ is an example of a zero polynomial with nonzero coefficients and there exists no polynomial of degree less than n that has n zeros over the field \mathbb{Z}_n .

Now we prove the claim for composite n . Suppose that $n = \prod_{j=1}^m p_j^{a_j}$ is the unique prime factorization of n , where $p_1 < p_2 < \dots < p_m$ are primes and a_i are natural numbers.

To show $\delta(n) \leq \rho(n)$, take the polynomial

$$h(x) = p_1^{a_1-1} p_2^{a_2} \dots p_m^{a_m} x(x^{p_1-1} - 1).$$

We claim that for all $a \in \mathbb{Z}_n$, $h(a) \equiv 0 \pmod{n}$. The case for $a = 0$ is trivial. For $a \neq 0$, note that n divides $h(a)$ if and only if p_1 divides $a(a^{p_1-1} - 1)$; but p_1 does divide $a(a^{p_1-1} - 1)$, for any $a \neq 0$.

Now we need to show that $\delta(n) \geq \rho(n)$. Assume for contradiction that there is a polynomial h of degree $d < \rho(n) = p_1$ that is zero on all $a \in \mathbb{Z}_n$. Since h is nonzero modulo n , it is nonzero modulo p_i , for some $i \in \{1, 2, \dots, m\}$. This is a contradiction to the fact that $\delta(p) = p$ for primes p . ■

Lemma 7.2 Let n be a natural number. Let $f(x)$ be a univariate polynomial of degree $d < \delta(n)$ over \mathbb{Z}_n . Then

$$|z(f)| \leq \frac{dn}{\rho(n)}.$$

Proof For the proof we assume that $f(x)$ has the maximum number of zeros among all polynomials of degree d over \mathbb{Z}_n . Assume that $n = p_1^{a_1} \dots p_k^{a_k}$, where each p_i is a prime number and each a_i is a positive integer. We also assume that the primes p_i 's are indexed so that $p_1 < p_2 < \dots < p_k$.

The following fact will allow us to bound $|z(f)|$ by the number of zeros of another polynomial whose construction is based on f . For any constant $\gamma \in \mathbb{Z}_n$, $z(\gamma f) \supseteq z(f)$, in particular, $|z(f)| \leq |z(\gamma f)|$. Let $f(x) = \sum_{j=0}^d \alpha_j x^j$. Without loss of generality, we assume that for some index i_0 ,

$$(2) \quad p_{i_0}^{a_{i_0}-1} \prod_{j \neq i_0} p_j^{a_j} \mid \alpha_d,$$

Otherwise we can multiply $f(x)$ by some of the constants p_1, \dots, p_k so that α_d satisfies Equation 2. In fact, without loss of generality, we also may assume that there is an index i_0 such that for all k ,

$$(3) \quad p_{i_0}^{a_{i_0}-1} \prod_{j \neq i_0} p_j^{a_j} \mid \alpha_k.$$

Otherwise, we could multiply the current polynomial with p_{i_0} and obtain another polynomial of smaller degree. We repeat this process until the resulting polynomial satisfies Equation 3 above. The process cannot produce the zero polynomial without satisfying Equation 3 first. Note that the entire process only involves multiplication of polynomials by scalars and hence can only increase the number of zeros.

Let $\hat{f}(x)$ be the resulting polynomial that is of degree $\hat{d} \leq d$. Suppose that $\hat{f}(x) = \sum_{i=0}^{\hat{d}} \hat{\alpha}_i x^i$, where i_0 is such that $1 \leq i_0 \leq k$, such that for all $j = 0, 1, \dots, \hat{d}$

$$p_{i_0}^{a_{i_0}-1} \prod_{j \neq i_0} p_j^{a_j} \mid \hat{\alpha}_j.$$

We can write the polynomial $\hat{f}(x)$ as

$$\hat{f}(x) = \frac{n}{p_{i_0}} \sum_{i=0}^{\hat{d}} \tilde{\alpha}_i x^i = \frac{n}{p_{i_0}} \tilde{f}(x),$$

where $\hat{\alpha}_i = \frac{n}{p_{i_0}} \tilde{\alpha}_i$. Now we recall the following fact from number theory.

Fact 7.2 For any $k, x, y \in \mathbb{Z}_n$, $kx \equiv ky \pmod{n}$ if and only if $x \equiv y \pmod{\frac{n}{\gcd(k,n)}}$.

In particular, using the above fact we have

$$\hat{f}(x) \equiv 0 \pmod{n} \iff \tilde{f}(x) \equiv 0 \pmod{p_{i_0}}.$$

Note that one solution of the equation $\tilde{f}(x) \equiv 0 \pmod{p_{i_0}}$ will generate $\frac{n}{p_{i_0}}$ solutions in the equation $\hat{f}(x) \equiv 0 \pmod{n}$. We know that $z(\tilde{f}) \leq \hat{d} \leq d$, since p_{i_0} is prime, and that $\frac{n}{p_{i_0}} \leq \frac{n}{\rho(n)}$. ■

In what follows we will assume that probabilities are taken over the uniform distribution with respect to a certain set. For example, the notation $\Pr_{x \in A}[B]$ means the probability of event B when x is uniformly sampled from the set A . We restate the above lemma in a probabilistic setting.

Corollary 7.1 Let n be a natural number. Let $f(x)$ be a nonzero univariate polynomial of degree $d < \delta(n)$ over \mathbb{Z}_n . Then

$$\Pr_{x \in \mathbb{Z}_n} [f(x) = 0] \leq \frac{d}{\rho(n)}.$$

Let $f(x_1, \dots, x_k)$ be a multivariate polynomial over \mathbb{Z}_n with k variables (also called a k -variate polynomial over \mathbb{Z}_n). We say that f has degree at most d if each variable x_i has degree at most d . Let $z_k(n, d)$ denote the maximum number of zeros of any multivariate polynomial of degree d over \mathbb{Z}_n in k variables. The following lemma is based on the ideas in [S80].

Lemma 7.3 Let n be a natural number and let $d < \delta(n)$.

$$z_k(n, d) \leq \frac{dkn^k}{\rho(n)}.$$

We are now ready to state the randomized interpolation results for polynomials over \mathbb{Z}_n . A randomized interpolation algorithm for polynomials over \mathbb{Z}_n is a randomized learning algorithm that has access to a substitution oracle for some polynomial f over \mathbb{Z}_n . After querying the substitution oracle, the interpolation algorithm outputs a hypothesis polynomial h such that $f \equiv h$ with high probability.

Theorem 7.1 *Let n, k, d be positive integers that satisfy $kd < \frac{1}{2}\rho(n)$. Then any k -variate polynomial of degree d over \mathbb{Z}_n with t summands can be interpolated in randomized polynomial time. The running time of the algorithm is polynomial in n, k and t .*

Proof Let $f(x)$ be the target k -variate degree d polynomial over \mathbb{Z}_n which has t summands. Assume that f is nonzero since otherwise repeated sampling will discover this. The interpolation algorithm will simulate the exact learning algorithm using substitution and equivalence queries for multivariate polynomials over \mathbb{Z}_n (from Section 7.1). It suffices to show how to simulate equivalence queries using substitution queries via the extended Schwartz's lemma, i.e., probabilistic zero-testing.

Suppose that the hypothesis issued by the learning algorithm is $h(x)$. First we need to show how to transform h into a k -variate degree d polynomial over \mathbb{Z}_n . Note that a k -variate polynomial with t summands is representable as a matrix function $\prod_{i=1}^k A_i(x_i)$, where each matrix A_i is a mapping from \mathbb{Z}_n to $\mathbb{Z}_n^{t \times t}$. Using similar arguments as in [BBBKV96], we can transform each mapping A_i into a $t \times t$ matrix $H_i(x_i)$ such that each entry of H_i is a univariate polynomial of degree at most d in the variable x_i . To do this, we simply interpolate each entry using the corresponding entries from $\{A_i(a) : a \in \mathbb{Z}_n\}$ (note that a univariate degree d polynomial can have at most $d \frac{n}{\rho(n)}$ zeros; since $d < \rho(n)$, this is less than n). A brute force approach to interpolate a degree d single-variable polynomial over \mathbb{Z}_n takes n^{d+1} time. So, in total, this transformation takes $\mathcal{O}(kt^2n^{d+1})$ steps for all mappings $A_i, i = 1, 2, \dots, k$. It is not hard to see that the function defined by $\prod_{i=1}^k H_i(x_i)$ is a k -variate degree d polynomial that computes h .

Assume that $h \not\equiv f$. Then the probability that $(f - h)(x) = 0$ for a randomly chosen $x \in \mathbb{Z}_n^k$ is at most $\frac{kd}{\rho(n)} < \frac{1}{2}$, by the extended Schwartz's lemma. If h is the i -th equivalence query asked by the learning algorithm, then the interpolation algorithm draws a set S_i of m_i randomly chosen points from \mathbb{Z}_n^k , where

$$m_i = t(\log n + k \log(d + 1)) + (i + 1).$$

The probability that $(f - h)(x) = 0$ for all $x \in S_i$ is at most 2^{-m_i} . If the two polynomials f and h agree on all points of S_i then the interpolation algorithm halts and declares that h is the target polynomial, otherwise, there is a point $a \in S_i$ that is a counterexample for the learning algorithm.

The mistake probability of the interpolation algorithm during the i -th equivalence query is the probability that there is a polynomial $q \not\equiv f$ that agrees with f on all points of S_i . Since there are at most $(n(d + 1)^k)^t$ k -variate degree d polynomials with t summands over \mathbb{Z}_n , the probability that there is a k -variate polynomial $q \not\equiv f$ of degree d with t summands such that $(f - q)(x) = 0$, for all $x \in S_i$, is at most $\frac{(n(d+1)^k)^t}{2^{m_i}} \leq 2^{-(i+1)}$. Hence the probability that the interpolation algorithm makes a mistake at some iteration is at most $\sum_{i=1}^{\infty} 2^{-(i+1)} \leq \frac{1}{2}$. We can repeat to make the error probability smaller if necessary. ■

Note that the above result relied on the assumption that $\rho(n)$ is rather large. The following result gives evidence that if $\rho(n)$ is small then efficient interpolation is impossible even for the univariate case. We will illustrate this for the case of $\rho(n) = 2$, but this can be extended to other cases.

Lemma 7.4 *Let n be a natural number such that $\rho(n) = 2$, i.e., 2 is a prime factor of n . Let \mathcal{P} be the following class of multivariate polynomials over \mathbb{Z}_n :*

$$\mathcal{P} = \left\{ \frac{n}{2} \prod_{i=1}^k (x_i + \alpha_i) \mid \alpha_i \in \{0, 1\}, 1 \leq i \leq k \right\}.$$

Then any interpolation algorithm for \mathcal{P} requires $\Omega(2^k)$ substitution queries.

Proof The idea is to notice that a polynomial of the form

$$f(x) = \frac{n}{2} \prod_{i=1}^k (x_i + \alpha_i)$$

is nonzero modulo n if and only if $\prod_{i=1}^k (x_i + \alpha_i)$ is odd. The product is odd if and only if $x_i + \alpha_i$ is odd, for all $i = 1, 2, \dots, k$. Hence interpolation essentially reduces to finding the α_i 's, for which there are 2^k combinations.

The proof is based on the following adversary argument. The adversary maintains the set of Boolean vectors A which is initially set to $\{0, 1\}^k$. Each element $\alpha \in A$ can be thought as representing the candidate polynomial $f_\alpha(x) = \frac{n}{2} \prod_{i=1}^k (x_i + \alpha_i)$. At any time during the interpolation, the set A represents the subset of \mathcal{P} of polynomials that are consistent with all answers given by the adversary to queries from the interpolation algorithm. In fact, the adversary always answers 0 (mod n) to all the queries.

Define a mapping π from \mathbb{Z}_n^k to $\{0, 1\}^k$ as follows. For $a \in \mathbb{Z}_n^k$, let $\pi(a)$ be a Boolean vector such that for all $i = 1, 2, \dots, k$,

$$\pi(a)_i = \begin{cases} 1 & \text{if } a_i \text{ is even} \\ 0 & \text{otherwise} \end{cases}$$

Note that $f_\alpha(a) \not\equiv 0 \pmod{n}$ if and only if $\alpha = \pi(a)$, for $\alpha \in \{0, 1\}^k$. If the interpolation algorithm asks a substitution query with $a \in \mathbb{Z}_n^k$ then the adversary answers 0 (mod n) and removes the vector $\pi(a) \in \{0, 1\}^k$ from A . This removes the polynomial $f_{\pi(a)}$ from the candidate set A . It is easy now to see that the interpolation algorithm must make at least $2^k - 1$ substitution queries. ■

8 Constant-depth circuits with restricted MOD gates

We turn our attention to exactly learning small depth Boolean circuits with *MOD* gates. The problem of learning small depth Boolean circuits with logical gates, e.g., AND, OR and NOT, is one of the main open problems in learning theory. An important subcase is the problem of learning DNF or depth two OR of ANDs. In [BBTV97], several classes of small depth Boolean circuits with threshold and modulo gates were considered. These classes are surprisingly expressive (e.g., threshold of modulo gates include DNFs) and some restricted classes were shown to be learnable in the exact model.

In this section we consider a different class of small depth Boolean circuits. We require that each gate of the circuit be a modulo gate and that the modulus be powers of a single fixed prime. We also require that the depth be a constant and that the size be polynomial in the number of inputs. We prove that this class of circuits is exactly learnable using equivalence and membership queries. It is open whether this result could be extended to the case where there are no restriction on the moduli or even when the moduli are products of two distinct primes.

The MOD_m gate or function on n inputs is defined as follows.

$$MOD_m(x_1, \dots, x_n) = \begin{cases} 1 & \text{if } \sum_{i=1}^n x_i \equiv 0 \pmod{m} \\ 0 & \text{otherwise} \end{cases}$$

The case that is of interest to us is when the modulus m is a power of some prime p , i.e., $m = p^k$. We point out that the following result can be proved using tools from [BBBKV96] without using the generalities of matrix functions.

Theorem 8.1 *Let p be a fixed prime. The class of constant depth Boolean circuits that contain MOD gates, where the moduli are powers of a single fixed prime p , is efficiently exactly learnable from equivalence and membership queries.*

Proof The idea is to show that a constant depth Boolean circuit C with MOD gates, where the moduli are powers of a single fixed prime p , can be transformed into a multiplicity automaton over \mathbb{Z}_p (or \mathbb{Z}_p -automaton for short). This transformation relies on several facts. First, it relies on the fact that \mathbb{Z}_p -automata are closed under the operations of addition, multiplication, and multiplication by a scalar. This fact is well-known (see [BBBKV96] and the references therein). In particular, the following can be proved: If $f(x)$ and $g(x)$ are computable with \mathbb{Z}_p -automaton of size s_f and s_g , respectively, then $cf(x)$, for $c \in \mathbb{Z}_p$, $f(x) + g(x)$, and $f(x)g(x)$ are computable by \mathbb{Z}_p -automaton of sizes s_f , $s_f + s_g$ and $s_f \times s_g$, respectively. Second, it relies on the fact that the MOD_p function (or counting modulo p) can be implemented by a \mathbb{Z}_p -automaton. Third, we need the following result due to Beigel and Tarui [BT94]. For any k there is a polynomial ψ of degree p^k that computes a Boolean function from $\{0, 1\}^n$ to $\{0, 1\}$ such that for all $x \in \{0, 1\}^n$

$$MOD_{p^k}(x_1, x_2, \dots, x_n) = 1 \iff \psi(x_1, \dots, x_n) \equiv 1 \pmod{p}.$$

In other words, counting modulo p^k can be *reduced* to counting modulo p . In fact the explicit expression for the polynomial ψ was given by [BT94] as follows.

$$\psi(x_1, \dots, x_n) = \prod_{i=1}^{k-1} \left(1 - \left(\sum_{S \subseteq [n]: |S|=p^i} \prod_{j \in S} x_j \right)^{p-1} \right).$$

Note that if each x_j is computed by an automaton of size at most s then, by the first fact, there is an automaton of size $(ns)^{p^k}$ that computes $\psi(x)$. To see this, note that $\sum_{S \subseteq [n]: |S|=p^i} \prod_{j \in S} x_j$ is computable by an automaton of size at most $\binom{n}{p^i} s^{p^i} \leq (ns)^{p^i}$. Thus, the i -th term in the product is computable in size $(ns)^{p^i(p-1)}$. The final product is computable in size $\prod_{i=1}^{k-1} (ns)^{p^i(p-1)} \leq (ns)^{p^k}$.

We claim inductively that if C has depth d then there is a \mathbb{Z}_p -automaton of size

$$m \sum_{j=1}^d p^{jk}$$

that computes C , where m is the largest fan-in of any gate of C . The base case is trivial since the circuit is an input variable, i.e., $d = 0$. For induction, suppose that the top gate of C is at depth $d + 1$ and computes the function $MOD_{p^k}(s_1, \dots, s_m)$, where each s_i is some subcircuit of C . By induction, each s_i is computable by a \mathbb{Z}_p -automaton of size at most $S = m \sum_{j=1}^d p^{jk}$. Thus, using our earlier observation, C is computable in size

$$(mS)^{p^k} = m^{p^k} (m \sum_{j=1}^d p^{jk})^{p^k} = m \sum_{j=1}^{d+1} p^{jk}.$$

This completes the induction. Since p, k, d are all constants and $m \in n^{\mathcal{O}(1)}$, this proves that there is a polynomial size \mathbb{Z}_p -automaton computing C . Thus C is exactly learnable using equivalence and membership queries with polynomial complexity. ■

Using similar arguments as in the proof of Theorem 8.1, one can show that a depth d circuit of size c (the total number of wires or edges in the circuit) that contains MOD gates, where the moduli are prime powers of a single fixed prime p , is computable by a \mathbb{Z}_p -automaton of size at most $c \sum_{j=1}^d p^{jK}$, where K is the largest power of p used by the MOD gates.

9 Concluding Remarks

Bergadano and Varricchio [BV95] introduced *transition graphs* for learning \mathcal{Q} -automata using equivalence queries only (assuming the counterexamples are shortest possible). Their transition graph is similar to our decision programs in that the graph is leveled and that certain nodes are linearly dependent of other nodes. However, in a decision program the dependencies are local to a level whereas in a transition graph the dependencies span two adjacent levels. In a decision program, there is an explicit distinction made between independent and dependent nodes. Their hypothesis updates also involve solving linear systems over the rationals, while we deal with solving dependencies over rings.

Acknowledgments

We thank Stefano Varricchio for referring us to [BV95] and pointing out the similarity between *decision programs* and *transition graphs*. We thank the anonymous referees for insightful comments, especially for improving the presentation in Sections 7 and 8 and for suggesting corrections to earlier omissions and errors.

References

- [A87] Dana Angluin. Learning Regular Sets from Queries and Counterexamples. *Information and Computation*, 75:87–106, 1987.
- [A88] Dana Angluin. Queries and Concept Learning. *Machine Learning*, 2:319–342, 1988.
- [BBV96] Francesco Bergadano, Nader H. Bshouty, and Stefano Varricchio. Learning Multivariate Polynomials from Substitution and Equivalence Queries. In *Electronic Colloquium on Computational Complexity*, TR96-008, 1996.
- [BBBKV96] Amos Beimel, Francesco Bergadano, Nader H. Bshouty, Eyal Kushilevitz, and Stefano Varricchio. On the Applications of Multiplicity Automata in Learning. In *Proceedings of the 37th IEEE Annual Symposium on Foundations of Computer Science*, Vermont, U.S.A., 349–358, 1996.
- [BBTV97] Francesco Bergadano, Nader H. Bshouty, Christino Tamon, and Stefano Varricchio. On Learning Branching Programs and Small Depth Circuits. In *Third European Conference on Computational Learning Theory (ECOLT)*, vol. 1208, Lecture Notes in Computer Science, 150-161, 1997.
- [BT94] Richard Beigel and Jun Tarui. On ACC. In *Computational Complexity*, 4:350–366, 1994.

- [BV94] Francesco Bergadano and Stefano Varricchio. Learning Behaviors of Automata from Multiplicity and Equivalence Queries. In *Proceedings of the 2nd Italian Conference on Algorithms and Complexity*, vol. 778, Lecture Notes in Computer Science, 54-62, 1994.
- [BV95] Francesco Bergadano and Stefano Varricchio. Learning Behaviors of Automata from Shortest Counterexamples. In *First European Conference on Computational Learning Theory (ECOLT)*, vol. 904, Lecture Notes in Artificial Intelligence, 380-391, 1995.
- [M] Hideyuki Matsumura. *Commutative Ring Theory*. Cambridge University Press, 1986.
- [S80] J. T. Schwartz. Probabilistic Algorithms for Verification of Polynomial Identities. In *Journal of the Association for Computing Machinery*, 27:701-717, 1980.