An Algebraic Perspective on Boolean Function Learning

Ricard Gavaldà* Denis Thérien †

April 5th, 2010

Abstract

In order to systematize existing results, we propose to analyze the learnability of boolean functions computed by an algebraically defined model, programs over monoids. The expressiveness of the model, hence its learning complexity, depends on the algebraic structure of the chosen monoid. We identify three classes of monoids that can be identified, respectively, from Membership queries alone, Equivalence queries alone, and both types of queries. The algorithms for the first class are new to our knowledge, while for the other two are combinations or particular cases of known algorithms. Learnability of these three classes captures many previous learning results. Moreover, by using nontrivial taxonomies of monoids, we can argue that using the same techniques to learn larger classes of boolean functions seems to require proving new circuit lower bounds or proving learnability of DNF formulas.

This work was presented at ALT'2009. This version includes some proofs omitted in the ALT'2009 proceedings.

1 Introduction

In his foundational paper [Val84], Valiant introduced the (nowadays called) PAC-learning model, and showed that conjunctions of literals, monotone

*Department of Software (LSI), U. Politècnica de Catalunya, Barcelona, Spain. gavalda@lsi.upc.edu

†School of Computer Science, McGill University, Montréal, Québec, Canada. denis@cs.mcgill.ca

DNF formulas, and k-DNF formulas were learnable in the PAC model. Shortly after, Angluin proposed the (nowadays called) Exact learning from queries model, proved that Deterministic Finite Automata are learnable in this model [Ang87], and showed how to recast Valiant's three learning results in the exact model [Ang88].

Valiant's and Angluin's initial successes were followed by a flurry of PAC- or Exact learning results, many of them concerning (as in Valiant's paper) the learnability of Boolean functions, others investigating learnability in larger domains. For the case of Boolean functions, however, progress both in the pure (distribution-free, polynomial-time) PAC model or in the exact learning model has slowed down considerably in the last decade.

Certainly, one reason for this slowdown is the admission that these two models do not capture realistically many Machine Learning scenarios. So a lot of the effort has shifted to investigating variations of the original models that accommodate these features (noise tolerance, agnostic learning, attribute efficiency, distribution specific learning, subexponential time, ...), and important advances have been made here.

But another undeniable reason of the slowdown is the fact that *it is difficult* to find new learnable classes, either by extending current techniques to larger classes or by finding totally different techniques. Many existing techniques seem to be blocked by the frustrating problem of learning DNF, or by our lack of knowledge of basic questions on boolean circuit complexity, such as the power of modular or threshold circuits.

In this paper, we use algebraic tools for organizing many existing results on Boolean function learning, and pointing out possible limitations of existing techniques. We adopt the *program over a monoid* as computing model of Boolean functions [Bar89, BST90]. We use existing, and very subtle, taxonomies of finite monoids to classify many existing results on Boolean function learning, both in the Exact and PAC learning models, into three distinct algorithmic paradigms.

The rationale beyond the approach is that the algebraic complexity of a monoid is related to the computational complexity of the Boolean functions it can compute, hence to their learning complexity. Furthermore, the existing taxonomies of monoids may help in detecting corners of learnability that have escaped attention so far because of lack of context, and also in indicating barriers for a particular learning technique. We provide some examples of both types of indications. Similar insights have led in the past to, e.g., the complete classification of the communication complexity of boolean functions

and regular languages [TT05, CKK$^+$07].

More precisely, we present three classes of monoids that are learnable in three different Exact learning settings:

Strategy 1: Groups for which lower bounds are known in the program model, all of which are solvable. Boolean functions computed over these groups can be identified from polynomially many Membership queries and, in some cases, in polynomial or quasipolynomial time. Membership learning in polynomial time is impossible for any monoid which is not a solvable group.

Strategy 2: Monoids built as wreath products of **DA** monoids and p-groups. These monoids compute boolean functions computed by decision lists whose nodes contain MOD_p gates fed by NC^0 functions of the inputs. These are learnable from Equivalence queries alone, hence also PAC-learnable, using variants of the algorithms for learning decision lists and intersection-closed classes. The result can be extended to MOD_m gates (for nonprime m) with restrictions on their accepting sets. All monoids in this class are nonuniversal (cannot compute all boolean functions), in fact the largest class known to contain only nonuniversal monoids. We argue that proving learnability of the most reasonable extensions of this class (either in the PAC or the Equivalence-query model) requires either new circuit lower bounds or learning DNF.

Strategy 3: Monoids in the variety named $\mathbf{LG_p} \circledcirc_m \mathbf{Com}$. Programs over these monoids are simulated by polynomially larger Multiplicity Automata (in the sequel, MA) over the field $\mathbb{F}_p$, and thus are learnable from Membership and Equivalence queries. Not all MA can be translated to programs over such monoids; but all classes of Boolean functions that, to our knowledge, were shown to be learnable via the MA algorithm (except the full class of MA itself) are in fact captured by this class of monoids. We conjecture that this is the largest class of monoids that can be polynomially simulated by MA, hence it defines the limit of what can be learned via the MA algorithm in our algebraic setting.

These three classes subsume a good number of the classes of Boolean functions that have been proved learnable in the literature, and we will detail them when presenting each of the strategies. Additionally, with the algebraic interpretation we can examine more systematically the possible extensions these results, at least within our framework. By examining natural extensions of our three classes of monoids, we can argue that any substantial extension of two of our three monoid classes provably requires solving two notoriously hard problems: either proving learnability of DNF formulas or proving new

lower bounds for classes of solvable groups. This may be an indication that substantial advance on the learnability of circuit-based classes similar to the ones we capture in our framework may require new techniques.

Admittedly, there is no reason why every class of boolean functions interesting from the learning point of view should be equivalent to programs computed over a class of monoids, and certainly our classification leaves out many important classes. Among them are classes explicitly defined in terms of threshold gates, or by read-k restrictions on the variables, or by monotonicity conditions. This is somehow unavoidable in our setting, since threshold gates have no natural analogue on finite monoids, and because multiple reads and variable negation are free in the program model. Similarly, the full classes of MA and DFA cannot be captured in our framework, since for example the notion of automata size is critically sensitive to the order in which the inputs are read, while in the program model variables can always be renamed with no increase in size.

Our taxonomy is somehow complementary to that involving threshold functions, as in [HS07, She08]. Some classes of Boolean functions are captured by both that approach and ours, while each one contains classes not captured by the other.

2 Background

2.1 Boolean functions

We build circuits typically using AND, OR, and MOD gates. We sometimes use *and*, *or* to denote AND and OR gates of bounded fan-in, where the bound will be clear from the context. We use the generalized model of MOD_m gates that come equipped with an accepting set $A \subseteq [m]$[1] indicated as superindex: A MOD_m^A gate outputs is 1 iff the sum of its inputs mod m is in A. We simply write MOD_m gates to mean MOD_m^A gates with arbitrary A's. For each k, NC_k^0 is the set of boolean functions depending each on at most k variables.

We often compose classes of boolean functions. For two classes of boolean functions C and D, $C \circ D$ denotes functions in C with inputs replaced with functions in D.

The class of functions DL is that computed by decision lists where each node contains one variable. Therefore, e.g., $\mathrm{DL} \circ \mathrm{NC}_k^0$ are decision lists whose

[1] $[m]$ denotes the set $\{0 \cdots - 1\}$ throughout the paper.

nodes contain boolean functions depending on at most k variables.

We will use the computation model called *Multiplicity Automata*, MA for short. The following is one of several equivalent definitions; see e.g. [BV96, BBB+00] for more details. A multiplicity automaton over an alphabet Σ and a field F is a nondeterministic finite automaton over Σ where we associate an element of F to each transition. The value of the automaton on an input $x \in \Sigma^\star$ is the sum over all accepting paths of the products of the elements along the path, where sum and product are over the field. Alternatively, an MA with s states can be defined by associating an $s \times s$ matrix over F to each letter in Σ. The value of the automaton on $x_1 \dots x_n$ is the product of the matrices associated to letters $x_1, \dots x_n$, pre- and post-multiplied by two fixed row and column vectors.

We will at some point use the notion of *rank* of a binary tree, which is a useful combinatorial notion when discussing decision tree complexity [EH89, Blu92]. Intuitively, it is the depth of the largest complete tree that can be embedded in the tree:

Definition 1 *[EH89] The* rank *of a binary tree is defined as follows:*

- *The rank of a leaf is 0.*
- *If the two children of a tree T have ranks r_L and r_R, then the rank of T is $r_L + 1$ if $r_L = r_R$, and is $\max\{r_L, r_R\}$ if $r_L \neq r_R$.*

2.2 Learning Theory

We assume some familiarity with Valiant's PAC model and especially Angluin's model of Exact learning via queries. In the latter, a class of boolean functions $\mathcal{C}$ is agreed between Teacher and Learner. The Teacher fixes a *target function* $f \in \mathcal{C}$ on n variables in an adversarial way, and discloses n to the Learner, an algorithm. The Learner's goal is to identify f precisely. To do so, it can ask Membership and Equivalence queries. A Membership query is a string $x \in \{0,1\}^n$, and the answer from the Teacher must be $f(x)$. An Equivalence query is the representation of a function g, and the answer from the Teacher must be either 'Yes' if $f = g$, or a counterexample x such that $f(x) \neq g(x)$. The Learner succeeds if it eventually produces the representation of a function equal to f.

To measure the resources used by the Learner, we assume that associated to $\mathcal{C}$ there is a notion of "function size"; typically $\mathcal{C}$ is defined by a set of

representations of functions, and the size of f is the size of the shortest representation of f in $\mathcal{C}$. This can be made formal through the notion of "representation class" that we omit here. The resources used by the Learner are measured as a function of n, the arity of the target function f, and the size of f in $\mathcal{C}$, which we normally denote with s.

There are two variants of learning that we will mention: we say that the Learner "polynomially identifies $\mathcal{C}$" if it identifies every $f \in \mathcal{C}$ using a polynomial (in the sense above) number of queries, and any amount of computation time. We say that it "polynomially learns $\mathcal{C}$" if it does so using polynomial time. Thus, polynomial learning implies polynomial identification. Another kind of distinction arises considering the class of functions that the Learner can use as their final guesses for f and as intermediate hypothesis in the form of Equivalence queries. In proper learning, only (representations of) functions in $\mathcal{C}$ can be used; if a larger class can be used, learning is called improper. In this paper, we will in general allow improper learning.

We will use repeatedly the well-known *Composition Theorem* (see e.g. [KLPV87]) which states that if a class C (with minor syntactical requirements) is learnable in polynomial time then $C \circ \mathrm{NC}^0_k$ is also learnable in polynomial time for every fixed k. The result is valid for both the Equivalence-query model and the PAC model, but the proof fails in the presence of Membership queries.

2.3 Monoids and Programs

Recall that a *monoid* is a set equipped with a binary operation that is associative and has an identity. All the monoids in this paper are finite; some of our statements about monoids might be different or fail for infinite monoids.

A *group* is a monoid where each element has an inverse. A monoid is *aperiodic* if there is some number t such that $a^{t+1} = a^t$ for every element a. Only the one-element monoid is both a group and aperiodic. A theorem by Krohn and Rhodes states that every monoid can be built from groups and aperiodic monoids by repeatedly applying the so-called *wreath product*. The wreath product of monoids A and B is denoted with $A \star B$.

A *program* over a monoid M is a pair (P, A), where $A \subseteq M$ is the *accepting set* and P is an ordered list of instructions. An instruction is a triple (i, a, b) whose semantics is as follows: read (boolean) variable x_i; if $x_i = 0$, emit element $a \in M$, and emit element $b \in M$ if $x_i = 1$. A list of instructions P defines a sequence of elements in M on every assignment a to the variables.

We denote with $P(a)$ the product in M of this sequence of elements. If $P(a) \in A$ we say that the program accepts a, and that it rejects a otherwise; alternatively, we say that the program evaluates to 1 (resp. 0) on a.

Each program on n variables thus computes a boolean function from $\{0,1\}^n$ to $\{0,1\}$. For a monoid M, $\mathcal{B}(M)$ is the set of boolean functions recognized by programs over M. If $\mathcal{M}$ is a set of monoids, $\mathcal{B}(\mathcal{M})$ is $\bigcup_{M \in \mathcal{M}} \mathcal{B}(M)$.

A monoid M is said to *divide* a monoid N if M is a homomorphic image of a submonoid in N. A set of monoids closed under direct product and division (i.e., taking submonoids and homomorphic images) is called a variety (technically, a pseudovariety since we are dealing with finite monoids). The following varieties will appear in this paper:

- **Com**: All commutative monoids.
- **Ab**: All Abelian groups. Recall that every finite Abelian group is a direct product of a number of groups of the form $\mathbb{Z}_{p_i^\alpha}$ for different primes p_i.
- $\mathbf{G_p}$: All p-groups, that is, groups of cardinality a power of the prime p.
- $\mathbf{G_{nil}}$: Nilpotent groups. For the purposes of this paper, a group is nilpotent iff it is the direct product of a number of groups, each of which is a p_i-group for possibly different p_i's. All Abelian groups are nilpotent. For interpretation, it was shown in [PT88] that programs over nilpotent groups are equivalent in power to polynomials of constant degree over a ring of the form $(\mathbb{Z}_m)^k$.
- **G**: The variety of all groups.
- **DA**: A variety of aperiodic monoids to be defined in Section 4.2. For interpretation, it was shown in [GT03] that programs over monoids in **DA** are equivalent in power to decision trees of bounded rank.

2.4 Learning Programs over Monoids: Generalities

Every monoid M defines a set of boolean functions $\mathcal{B}(M)$ with an associated notion of function size, namely the length of the shortest program over M. The general question we ask is thus "given M and a learning model, is $\mathcal{B}(M)$ polynomial-time learnable in that learning model?". Polynomiality (or other

bounds) is on the number of variables and size in M of the target function, denoted with s as already mentioned.

For a set of monoids $\mathcal{M}$, we say for brevity "programs over $\mathcal{M}$ are learnable" or even "$\mathcal{M}$ is learnable" to mean "for every fixed $M \in \mathcal{M}$, $\mathcal{B}(M)$ is learnable", that is, there may be a different algorithm for each $M \in \mathcal{M}$, with a different running time. In other words, each algorithm works for a fixed M that it "knows". Models where a single algorithm must work for a whole class of monoids are possible, but we do not pursue them in this paper.

The following easy result is useful to compare the learning complexity of different monoids:

Fact 2 *If M divides N and $\mathcal{B}(N)$ is learnable (in any of the learning models in this paper), then $\mathcal{B}(M)$ is also learnable.*

In contrast, we do not know whether learnability is preserved under direct product (which is to say, by taking fixed-size boolean combinations of classes of the form $\mathcal{B}(M)$): if it was, many of the open problems in this paper would be resolved, but have no general argument or counterexample.

A fact we will often use to tranfer learning results between monoid territory and circuit territory is the following, implicitly proved and used in [BT88].

Fact 3 *For every monoid M and group G, $\mathcal{B}(M \star G) = \mathcal{B}(M) \circ \mathcal{B}(G)$.*

This is in fact true whenever G is a monoid that can implement a "reset operation", which for groups in particular can be implemented by taking product $|G|$ times.

3 Learning from Small-Weight Assignments

The small-weight strategy applies to function classes with the following property.

Definition 4 *For an assignment $a \in \{0,1\}^n$, the* weight *of a defined as the number of 1s it contains, and denoted $w(a)$. A representation class $\mathcal{C}$ is* k-narrowing *if every two different functions $f, g \in \mathcal{C}$ of the same arity differ on some assignment of weight at most k. (k may actually be a function of some other parameters, such as the arity of f and g or their size in $\mathcal{C}$).*

The following is essentially proved in [GTT06].

Theorem 5 *If $\mathcal{C}$ is k-narrowing, then $\mathcal{C}$ can be identified with $n^{O(k)}$ Membership queries (and possibly unbounded time).*

The algorithm witnessing this is simple: ask all assignments in $\{0,1\}^n$ of weight at most k, of which there are at most $n^{O(k)}$. Then find any function $f \in \mathcal{C}$ consistent with all answers. By the narrowing property, that f must be equivalent to the target.

3.1 Groups with Lower Bounds

It was shown in [Bar89] and [GTT06], respectively, that nonsolvable groups and nongroups can compute any conjunction of variables and their negations by a polynomial-size program. Any class of functions with this property is not n-narrowing, and by a standard adversary argument, it requires 2^n Membership queries to be identified. Therefore we have:

Fact 6 *If M is not a group, or if M is a nonsolvable group, then $\mathcal{B}(M)$ cannot be identified with a subexponential number of Membership queries.*

Therefore, Membership learnability of classes of the form $\mathcal{B}(M)$ is restricted, at most, to solvable groups. There are two maximal subclasses of solvable groups for which lower bounds on their computational power are known, and in both cases the lower bound is essentially a narrowing property.

Fact 7
1. *For every nilpotent group M there is a constant k such that $\mathcal{B}(M)$ is k-narrowing [PT88]. Therefore $\mathcal{B}(M)$ can be identified from $n^{O(k)}$ Membership queries (and possibly unbounded time).*
2. *For every group $G \in \mathbf{G_p} \star \mathbf{Ab}$ there is a constant c such that $\mathcal{B}(M)$ is $(c \log s)$-narrowing [BST90]. Therefore, programs over G of length s can be identified from $n^{O(\log s)}$ Membership queries.*

The next two theorems give specific, time-efficient versions of this strategy for Abelian groups and $\mathbf{G_p} \star \mathbf{Ab}$ groups. These are, to our knowledge, new learning algorithms.

Theorem 8 *For every Abelian group G, $\mathcal{B}(G)$ is learnable from Membership queries in time n^c, for a constant $c = c(G)$.*

Theorem 9 *For every $G \in \mathbf{G_p} \star \mathbf{Ab}$ with p prime, $\mathcal{B}(G)$ is learnable from Membership queries in $n^{c \log s}$ time, for a constant $c = c(G)$.*

(Recall that s stands for the length of the shortest program computing the target function). Proofs are given in the Appendix.

3.2 Interpretation in Circuit Terms

Let us now interpret these results in circuit terms. It is easy to see that programs over a fixed Abelian group are polynomially equivalent to a boolean combination of some fixed number of MOD_m gates, for some m. Theorem 8 then implies:

Corollary 10 *For every m, boolean combinations of s MOD_m gates are learnable from Membership queries in time n^c, for $c = c(m, s)$.*

Also, it is shown in [BST90] that programs over a fixed group in $\mathbf{G_p} \star \mathbf{Ab}$ are polynomially equivalent to $\mathrm{MOD}_p \circ \mathrm{MOD}_m$ circuits. Such circuits were shown in [BBTV97] to be polynomial-time learnable from Membership and Equivalence queries in polynomial time, by showing that they have small Multiplicity automata – a generalization of their construction is used in Section 5. Theorem 9 shows that Membership queries suffice, if quasipolynomial time is allowed:

Corollary 11 *For every prime p and every m, the class of functions computed by $\mathrm{MOD}_p \circ \mathrm{MOD}_m$ circuits of size s are learnable from Membership queries in time $n^{O(\log s)}$.*

As an example, the 6-element permutation group on 3 points, S_3, can be described as a wreath product $\mathbb{Z}_3 \star \mathbb{Z}_2$. Intuitively each permutation can be described by a rotation and a flip, which interact when permutations are composed so direct product does not suffice. Programs over S_3 are polynomially equivalent to $\mathrm{MOD}_3 \circ \mathrm{MOD}_2$ circuits, and our result claims that they are learnable from $n^{c \log s}$ Membership queries for some c.

3.3 Open Questions on Groups and Related Work

While programs over nilpotent groups can be identified from polynomially many Membership queries, we have not resolved whether a time-efficient

algorithm exists, even in the far more powerful PAC+Membership model. In other words, we know that the values of such a program on all small-weight assignments are sufficient to identify it uniquely, but can these values be used to efficiently predict the value of the program on an arbitrary assignment?

In circuit terms, by results of [PT88], such programs can be shown to be polynomially equivalent to fixed-size boolean combinations of $\mathrm{MOD}_m \circ \mathrm{NC}^0$ circuits or, equivalent, of polynomials of constant degree over $\mathbb{Z}_m$. We are not even aware of algorithms learning a single $\mathrm{MOD}_m^A \circ \mathrm{NC}^0$ circuit for arbitrary sets A. When m is prime, one can use Fermat's little theorem to make sure that the MOD_m gate receives only inputs summing to either 0 or 1, at the expense of increasing the arity of the NC^0 part. Then, one can set up a set of linear equations where the unknowns are the coefficients of the target polynomial and each small-weight assignment provides an equation with constant term either 0 or 1. The solution of this system must be equivalent to the target function.

For solvable groups that beither nilpotent nor in $\mathbf{G_p} \star \mathbf{Ab}$, the situation is even worse in the sense that we do not have lower bounds on their computational power, i.e., we cannot show that they are weaker than NC^1. Observe that any learning result would establish a separation with NC^1, conditioned to the cryptographic assumptions under which NC^1 is nonlearnable. In another direction, while lower bounds do exist for $\mathrm{MOD}_p \circ \mathrm{MOD}_m$ circuits, we do not have them for $\mathrm{MOD}_p \circ \mathrm{MOD}_m \circ \mathrm{NC}^0$; linear lower bounds for some particular cases were given in [CGPT06].

Let us note that programs over Abelian groups (equivalently, boolean combinations of MOD_m gates) are particular cases of the multi-symmetric concepts studied in [BCJ93]. Multi-symmetric concepts are there shown to be learnable from Membership and Equivalence queries, while we showed that for these particular cases Membership queries suffice. XOR's of k-terms and depth-k decision trees are special cases of $\mathrm{MOD}_m \circ \mathrm{NC}^0$ previously noticed to be learnable from Membership queries alone [BK].

4 Learning intersection-closed classes

In this section we observe that classes of the form $\mathrm{DL} \circ \mathrm{MOD}_m^A \circ \mathrm{NC}^0$ are learnable from Equivalence queries (for some particular combinations of m and accepting sets A). The algorithm is actually the combination of two well-known algorithms (plus the composition theorem to deal with NC^0).

1) The algorithm for learning submodules of a module in [HSW90] (though probably known before); 2) the algorithm in the companion paper extending it to nested differences of intersection-closed classes, also in [HSW90].

It had been shown in [BBTV97] that decision lists whose nodes contain polynomials of constant degree over $\mathbb{F}_2$ are learnable from Equivalence queries, essentially by the same algorithm. This result extends to other fields $\mathbb{F}_p$, the main point being that the accepting set plays no role in the case $p = 2$, but one must deal with it for other primes p.

We furthermore show that the classes above have natural algebraic interpretation, and use this interpretation that they may be very close to a stopping barrier for a certain kind of learning.

4.1 The Learning Algorithm

Theorem 12 *For every m and k the class $\mathrm{DL}\circ\mathrm{MOD}_m^{[m]-\{0\}}\circ\mathrm{NC}_k^0$ is learnable from Equivalence queries in time polynomial in m, 2^{2^k}, and n^k.*

Proof. By the composition theorem, it suffices to show that $\mathrm{DL}\circ\mathrm{MOD}_m^{[m]-\{0\}}$ is learnable from Equivalence queries. We can furthermore assume that the MOD_m gates receive only variables (not constants) as inputs, since we can replace any constant 1 input with a dummy variables and appeal to the composition theorem again.

Following [HSW90], for a class $\mathcal{C}$, let $nd(\mathcal{C})$ be the set of nested differences of concepts in $\mathcal{C}$, i.e., of functions of the form $f_1 - (f_2 - (f_3 - (\ldots)))$ with each $f_i \in \mathcal{C}$ (we are here identifying the function f with the set of inputs on which f is 1). Also, let $not(\mathcal{C})$ the set of negations of functions in $\mathcal{C}$. Our algorithm is based on the observation that, for every class $\mathcal{C}$ containing the constant functions, $\mathrm{DL} \circ \mathcal{C} = nd(not(\mathcal{C}))$. To see this, consider a function in $\mathrm{DL} \circ \mathcal{C}$. By inserting dummy nodes, we can assume w.l.o.g. that the list computing it starts with a node emitting 0 and that nodes emitting 0 and 1 alternate. Let the list be of the form $L = (f, 0), (g, 1), L'$ where $f, g \in \mathcal{C}$ and L' denotes the rest of the list (and the function it computes). Observe then that the function computed by L is equivalent to the nested difference $not(f) - (not(g) - L')$ and proceed inductively.

So we have to show that $nd(\mathcal{C})$ is learnable, for $\mathcal{C}$ the class of $\mathrm{MOD}_m^{\{0\}}$ functions whose inputs are variables only. Every $\mathrm{MOD}_m^{\{0\}}$ function is the restriction to inputs in $\{0,1\}^n$ of the set of solutions of a homogeneous equation over $\mathbb{Z}_m$, therefore a submodule of $\mathbb{Z}_m^n$. The set of all submodules of $\mathbb{Z}_m^n$

is closed under intersection (the intersection of two submodules is a submodule), and therefore we are trying to learn nested differences of submodules of $\mathbb{Z}_m^n$. Thus, we can apply the algorithm for learning nested differences, and learn the class with a number of queries equal to the maximum size of a set of linearly independent tuples in $\mathbb{Z}_m^n$, which is known to be at most $n \log m$.

The hypothesis class is thus nested differences of submodules, where each submodule is defined as the span of a set of of previous counterexamples in $\{0,1\}^n$. Determining whether another vector belongs to the span of a given set of vectors amounts to solving systems of linear equations over $\mathbb{Z}_m^n$, which can be done in polynomial time, and therefore the hypothesis class is polynomial-time evaluatable. ∎

Note that in Theorem 12 the running time does not depend on the length of the decision list that is being learned. In fact, as a byproduct of this proof one can see that the length of these decision lists can be limited to a polynomial of m and n^k without actually restricting the class of functions being computed. Intuitively, this is because there can be only as many linearly independent such MOD gates, and a node whose answer is determined by the previous ones in the decision list can be removed. Thus, for constant m and k, this class can compute at most $2^{n^{O(1)}}$ n-ary boolean functions and is not universal.

Also, note that we claim this result for MOD_m gates having all but 0 as accepting elements. In the special case that m is a prime p, we can deal with arbitrary accepting sets:

Lemma 13 *For every prime p and every k, $\mathrm{DL} \circ and_k \circ \mathrm{MOD}_p \circ \mathrm{NC}^0_k$ is included in $\mathrm{DL} \circ \mathrm{MOD}_p^{[p]-\{0\}} \circ \mathrm{NC}^0_{(p-1)k^2}$.*

(Observe that, in the above, each usage of MOD without a superscript allows for a different accepting set.) To prove Lemma 13, we show first that every function in MOD_p^A is included in $\mathrm{MOD}_p^{[p]-\{0\}} \circ \mathrm{NC}^0_{p-1}$. Indeed, by a now standard use of Fermat's little theorem, one can see that

$$\mathrm{MOD}_p^A(x_1, \ldots, x_n) = \prod_{a \notin A} (\sum_{i=1}^{n} x_i - a)^{p-1},$$

and distributing in the right-hand side, one obtains a 0/1-valued, degree-$(p-1)$ polynomial, computable in $\mathrm{MOD}_p^{[p]-\{0\}} \circ \mathrm{NC}^0_{p-1}$. The *and*, equivalently

the product, of k such polynomials is a polynomial of degree $(p-1)k$, and the lemma follows.

As a corollary of Theorem 12 and Lemma 13, we have:

Corollary 14 *For every prime p and every k, $\mathrm{DL} \circ and_k \circ \mathrm{MOD}_p \circ \mathrm{NC}^0_k$ is learnable from Equivalence queries in time n^c, where $c = c(p,k)$.*

If we ignore the issue of proper learning and polynomials in the running time, this subsumes at least the following known results:

- k-decision lists (which are $\mathrm{DL} \circ \mathrm{NC}^0$) [Riv87]. k-decision lists in turn subsumed k-CNF and k-DNF, and rank-k decision trees.
- Systems of equations over $\mathbb{Z}_m$ (which are a subclass of $\mathrm{DL} \circ \mathrm{MOD}_m^{[m]-\{0\}}$).
- Polynomials of constant degree over finite fields, restricted to boolean functions. When the field has prime cardinality p, these are equivalent to $\mathrm{MOD}_p \circ \mathrm{NC}^0$.
- Decision lists having polynomials of bounded degree over $\mathbb{F}_2$ at the nodes [BBTV97].
- Strict width-2 branching programs [BBTV97]. This is because it is easy to show that these are polynomially simulated by $\mathrm{DL} \circ \mathrm{MOD}_2 \circ \mathrm{NC}^0$.

These are virtually all known results on learning Boolean functions in the pure PAC model (no Membership queries) that do not involve threshold gates or read-restrictions, neither of which can be captured in our algebraic setting. Observe that each of them contains at most $2^{n^{O(1)}}$ functions, hence is not universal. In the next section we remark that the class of monoids we have identified is in fact the largest known to contain only nonuniversal monoids, so it is no chance that it unifies these previous results.

4.2 Interpretation in Algebraic Terms

Classes closely related to those in the previous section have clear precise algebraic interpretations. They involve the class **DA** of monoids, of which we give here an operational definition. Formal definitions can be found e.g. in [Sch76, GT03, Tes03, TT04].

Let M be a monoid in **DA**. Then the product of elements $m_1, \ldots, m_n$ in M can be determined by knowing the truth or falsehood of a fixed number of boolean conditions of the form "$m_1 \ldots m_n$, as a string over M, admits a factorization of the form $L_0 a_1 L_1 a_2 \ldots a_k L_k$", where 1) the a_i are elements of M, 2) each L_i is a language such that $x \in L_i$ can be determined solely by the set of letters appearing in x, and 3) the expression $L_0 a_1 L_1 a_2 \ldots a_k L_k$ is unambiguous, i.e., every string has at most one factorization in it.

As mentioned already in the introduction, it was shown in [GT03] that programs over monoids in **DA** are equivalent in power to decision trees in bounded rank [EH89], where the required rank of the decision trees is related to the parameter k in its definition in the particular **DA** monoid. In particular, programs over a fixed **DA** monoid can be simulated both by CNF and DNF formulas of size $n^{O(1)}$ and by decision lists with bounded-length terms at the nodes, and can be learned in the PAC and Equivalence-query models [EH89, Riv87, Sim95].

We then have the following characterization:

Theorem 15 *1.* $\mathcal{B}(\mathbf{DA} \star \mathbf{G_{nil}}) = \bigcup_{m,k} \mathrm{DL} \circ \mathrm{MOD}_m \circ \mathrm{NC}^0_k = \bigcup_{m,k} \mathrm{DL} \circ \mathrm{MOD}^{\{0\}}_m \circ \mathrm{NC}^0_k$.

2. $\mathcal{B}(\mathbf{DA} \star \mathbf{G_p}) = \bigcup_k \mathrm{DL} \circ \mathrm{MOD}_p \circ \mathrm{NC}^0_k = \bigcup_k \mathrm{DL} \circ \mathrm{MOD}^{\{0\}}_p \circ \mathrm{NC}^0_k = \bigcup_k \mathrm{DL} \circ \mathrm{MOD}^{[p]-\{0\}}_p \circ \mathrm{NC}^0_k$.

Proof. We prove part (1) by a series of claims:

Claim a. $\mathcal{B}(\mathbf{DA} \star \mathbf{G_{nil}}) = \mathcal{B}(\mathbf{DA}) \circ \mathcal{B}(\mathbf{G_{nil}})$, by Fact 3.

Claim b. For every monoid M in **DA** there is some k such that $\mathcal{B}(M) \subseteq \mathrm{DT}_k$ where DT_k is the class of functions computed by rank-k decision trees. Vice-versa, for every k there is some $M_k \in \mathbf{DA}$ such that $\mathrm{DT}_k \subseteq \mathcal{B}(M_k)$. This was shown in [GT03].

Claim c. For every k we have $\mathrm{DT}_k \subseteq \mathrm{DL} \circ and_k$. This was observed by Blum [Blu92].

Claim d. For every nilpotent group G there are m and k such that $\mathcal{B}(G) \subseteq or_{m^k} \circ and_k \circ \mathrm{MOD}^{[0]}_m \circ and_k$. Vice-versa, for every c and m there is some nilpotent group G such that $or_c \circ and_c \circ \mathrm{MOD}^{[0]}_m \circ and_c \subseteq \mathcal{B}(G)$. This is a statement in circuit terms of the result in [PT88] mentioned in the introduction, that nilpotent groups are equivalent in power to polynomials of constant degree over rings of the form $(\mathbb{Z}_m)^k$. Intuitively, the *or* ranges over the accepting subset of $(\mathbb{Z}_m)^k$, and the *and* checks every component of $\mathbb{Z}_m$.

Claim e. $\mathrm{DL} \circ OR \circ \mathcal{C} = \mathrm{DL} \circ \mathcal{C}$. This is because a check for $f_1 \vee \cdots \vee f_s$ inside a node in a decision list can be simulated by s consecutive nodes, each checking some f_i.

From claims a to e, we have that for every $M \in \mathbf{DA} \star \mathbf{G_{nil}}$ there is some k such that

$$\mathcal{B}(M) \subseteq \mathrm{DT}_k \circ or_k \circ and_k \circ \mathrm{MOD}_m^{[0]} \circ and_k \subseteq \mathrm{DL} \circ and_k \circ \mathrm{MOD}_m^{[0]} \circ and_k.$$

Together with the following Claim f, this shows that for some K

$$\mathcal{B}(M) \subseteq \mathrm{DL} \circ \mathrm{MOD}_m^{[0]} \circ_K .$$

Claim f. For every k and p there is some $K = K(k, m)$ with $and_k \circ \mathrm{MOD}_m^{[0]} \subseteq \mathrm{MOD}_m^{[0]} \circ and_K$.

Proof of Claim f. Let $m = p_1^{\alpha_1} \cdot \cdots \cdot p_\ell^{\alpha^\ell}$ for distinct primes p_i. Take a function $f = f_1 \wedge \cdots \wedge f_k$, where each $f_i \in \mathrm{MOD}_m^{[0]}$. Since a number is 0 modulo m if and only for all i it is 0 modulo $p_i^{\alpha_i}$, and MOD_m functions are by definition 0/1-valued, we have for every i

$$f = \bigwedge_{i=1}^{k} \bigwedge_{j=1}^{\ell} f_{ij}$$

where each f_{ij} is in $\mathrm{MOD}_{p_i^{\alpha_i}}{}^{[0]}$. Crucially, Beigel and Tarui [BT94] showed that for every prime p and every α, and some r, every $\mathrm{MOD}_{p^\alpha}{}^{[0]}$ gate is equivalent to a degree-r, 0/1-valued polynomial over $\mathbb{F}_p$. That is, there are degree-r polynomials P_{ij} over $\mathbb{Z}$ such that $f_{ij} = 1$ if and only if $P_{ij} = 0 \mod p_j$. Define now $P_j = \prod_{i=1}^{k} P_{ij}$, which is a degree kr polynomial. We have $f = 1$ if and only if, for every j, $P_j = 0 \mod p_j$, i.e., $p_j^{\alpha_j - 1} P_j = 0 \mod p_j^{\alpha_j}$. By the Chinese Remainder Theorem, there is a polynomial P over $\mathbb{Z}$ of degree $kr\ell$ such that $p_j^{\alpha_j - 1} P_j = P \mod p_j^{\alpha_j}$ for every j. We thus have $f = 1$ if and only if $P = 0 \mod m$, which means that f is in $\mathrm{MOD}_m^{[0]} \circ and_{kr\ell}$. *(end proof of Claim f)*

The converse inclusion in part (1) of the theorem is that $\bigcup_{m,k} \mathrm{DL} \circ \mathrm{MOD}_m \circ \mathrm{NC}_k^0 \subseteq \mathcal{B}(\mathbf{DA} \star \mathbf{G_{nil}})$. It follows from the claims above with only one additional observation converse to Claim c:

Claim g. For every k, $\mathrm{DL} \circ and_k = \mathrm{DT}_1 \circ and_k$.

For part (2), use once more the Fermat trick to show that when p is prime, all accepting sets for MOD_p functions except for the empty set and $[p]$ have

the same power, modulo NC^0 functions at the input. This way, one has Claim g for every nontrivial accepting set, not only [0]. ∎

From this theorem and Lemma 13, it follows that we can learn programs over $\mathbf{DA} \star \mathbf{G_p}$ monoids from Equivalence queries, yet we do not know how to learn (to our knowledge) programs over $\mathbf{DA} \star \mathbf{G_{nil}}$ in any model. This algebraic interpretation lets us explore this gap in learnability and, in particular, the limitation of the learning paradigm in the previous subsection.

Since every p-group is nilpotent and it can be shown that $\mathbf{DA} \star \mathbf{G_{nil}}$ monoids can only have nilpotent subgroups, we have

$$\mathbf{DA} \star \mathbf{G_p} \subseteq \mathbf{DA} \star \mathbf{G_{nil}} \subseteq \mathbf{DA} \star \mathbf{G} \cap \mathbf{M_{nil}},$$

where $\mathbf{M_{nil}}$ is the class of monoids having only nilpotent subgroups. Yet, there is an important difference in what we know about $\mathbf{DA} \star \mathbf{G_p}$ and $\mathbf{DA} \star \mathbf{G_{nil}}$. Following [Tes03, TT04], a monoid M is said to have the Polynomial Length Property (or PLP) if every program over M, regardless of its length, is equivalent to another one whose length is polynomial in n. Clearly, every monoid in PLP is nonuniversal, and the converse is conjectured in [Tes03, TT04]. More specifically, the following was shown in [Tes03, TT04].

- Every monoid *not* in $\mathbf{DA} \star \mathbf{G} \cap \mathbf{M_{nil}}$ is universal.
- Every monoid in $\mathbf{DA} \star \mathbf{G_p}$ has the PLP, hence is not universal.

The question of either PLP or universality is thus open for $\mathbf{DA} \star \mathbf{G_{nil}}$, sitting between $\mathbf{DA} \star \mathbf{G_p}$ and $\mathbf{DA} \star \mathbf{G} \cap \mathbf{M_{nil}}$, so resolving its learnability may require new insights besides the intersection-closure/submodule-learning algorithm. Note that, contrary to one could think, $\mathbf{DA} \star \mathbf{G_{nil}}$ is *not* equal to $\mathbf{DA} \star \mathbf{G} \cap \mathbf{M_{nil}}$: there are monoids that, in this context, can be built by using unsolvable groups and later using homomorphisms to leave only nilpotent groups that cannot be obtained starting from nilpotent groups alone. Current techniques seem insufficient (and may remain unable forever) to analyze even these traces of unsolvability.

Are there other extensions of $\mathbf{DA} \star \mathbf{G_p}$ that we could investigate from the learning point? The "obvious" is trying to extend the $\mathbf{DA}$ or $\mathbf{G_p}$ parts separately. For the $\mathbf{DA}$ part, it is known [Sch76, Tes03] that every aperiodic monoid not in $\mathbf{DA}$ necessarily is divided by one of two well-identified

monoids, named U and BA_2. Monoid U is the syntactic monoid of the language $\{a,b\}^\star aa\{a,b\}^\star$, and programs over U are equivalent in power, up to polynomials, to DNF formulas. Therefore, by Fact 2, extending **DA** in this direction implies learning at least DNF. Monoid BA_2 is the syntactic monoid of $(ab)^\star$, and interestingly, although it is aperiodic, programs over it can be simulated (essentially) by OR gates fed by parity gates. In fact it in $\mathbf{DA} \star \mathbf{G_p}$ for every p, so we know it is learnable.

If we try to extend on the group part, we have already mentioned that the two classes of groups beyond $\mathbf{G_p}$ for which we have lower bounds are $\mathbf{G_{nil}}$ and $\mathbf{G_p} \star \mathbf{Ab}$. We have already discussed the problems concerning $\mathbf{DA} \star \mathbf{G_{nil}}$. For $\mathbf{G_p} \star \mathbf{Ab}$, they correspond to $\mathrm{MOD}_p \circ \mathrm{MOD}_m$ circuits, and we showed them to be learnable from Membership queries alone in the previous section. With Equivalence queries, however, learning $\mathrm{MOD}_p \circ \mathrm{MOD}_m$ would also imply learning $\mathrm{MOD}_p \circ \mathrm{MOD}_m \circ \mathrm{NC}^0$ and, as discussed in the previous section, this seems difficult because we cannot even prove now that these circuits cannot do NC^1. In particular, even learning programs over S_3 (i.e. $\mathrm{MOD}_3 \circ \mathrm{MOD}_2$ circuits) from Equivalence queries alone seems unresolved now.

5 Learning as Multiplicity Automata

The learning algorithm for multiplicity automata [BV96, BBB$^+$00] elegantly unified many previous results and also implied learnability of several new classes. It has remained one of the "maximal" learning algorithms for boolean functions, in the sense that no later result has superseded it.

Theorem 16 *[BV96, BBB$^+$00] Let F be any finite field. Functions $\Sigma^\star \to F$ represented as Multiplicity Automata over a fixed finite field are learnable from Evaluation and Equivalence queries in time polynomial in the size of the MA and $|\Sigma|$.*

We can use Multiplicity Automata to compute boolean functions as follows: We take $\Sigma = \{0,1\}$, and some accepting subset $A \subseteq F$, and the function evaluates to 1 on an input if the MA outputs an element in A, and 0 otherwise. However, as basically argued in [BBTV97] we can use Fermat's little theorem to turn an MA into one that always outputs either 0 or 1 (as field elements) with only polynomial blowup, and therefore we can omit the accepting subset.

In this section we identify a class of monoids that can be simulated by MA's, but not the other way round. Yet, it can simulate most classes of boolean functions whose learnability was proved via the MA-learning algorithm.

Note that it will be impossible to find a class of submonoids that, in our setting, is precisely equivalent (up to polynomial blowup) to the whole class of MA. This is true for the simple reason that the complexity of a function measured as "shortest program length" cannot grow under renaming of input variables: it suffices to change the variable names in the instructions of the program. MA, on the other hand, read their input in the fixed order $x_1, \ldots, x_n$, so renaming the input variables in a function can force an exponential growth in MA size. Consider as an example the function $\bigwedge_{i=1}^{n}(x_{2i-1} = x_{2i})$: clearly, it is computed by the MA of size $O(n)$ that simply checks equality of appropriate pairs of adjacent letters in its input string. However, its permutation $\bigwedge_{i=1}^{n}(x_i = x_{2n-i+1})$ is the palindrome function, whose MA size is roughly 2^n over any field.

Our characterization uses the notion of *Mal'tsev product* of two monoids A and B, denoted $A \circledcirc{m} B$. We do not define the algebraic operation formally. We use instead the following property, specific to our case [Wei87]: Let M be a monoid in $\mathrm{L}\mathbf{G_p} \circledcirc{m} \mathbf{Com}$, i.e., the Mal'tsev product of a monoid in $\mathbf{G_p}$ by one in $\mathbf{Com}$. Then, the product in M of a string of elements $m_1 \ldots m_n$ can be determined from the truth of a fixed number of logical conditions of the following form: There are elements $a_1, \ldots, a_k$ in M and commutative languages $L_0, \ldots, L_k$ over $M^\star$ such that the number of factorizations of $m_1 \ldots m_n$ of the form $L_0 a_1 L_1 a_2 L_2 \ldots L_{k-1} a_k L_k$, taken modulo p, is is some given value $p' < p$.

Contrived as it seems, the class $\mathrm{L}\mathbf{G_p} \circledcirc{m} \mathbf{Com}$ is a natural borderline in representation theory. Recent and deep work by Margolis *et al* [AMV05, AMSV09] shows that semigroups in $\mathrm{L}\mathbf{G_p} \circledcirc{m} \mathbf{Com}$ are exactly those that can be embedded into a semigroup of upper-triangular matrices over a field of characteristic p (and any size).

The main result in this section is:

Theorem 17 *Let M be a monoid in $\mathrm{L}\mathbf{G_p} \circledcirc{m} \mathbf{Com}$. Suppose that M is defined as above by the a boolean combination of at most ℓ conditions of length at most k using commutative languages whose monoid has size C. Then every program of length s over M is equivalent to an MA over $\mathbb{F}_p$ of size $(s + C)^c$, where $c = c(p, \ell, k)$.*

Corollary 18 *Programs over monoids in* $\mathbf{LG_p} \circledcirc_m \mathbf{Com}$ *are polynomially simulated by MAs over* $\mathbb{F}_p$ *that are direct sums of constant-width MA's.*

Proof. (of Theorem 17) (Sketch). Fix a program (P, A) over M of length s. Let $m_1, \ldots m_s$ be the sequence of elements in M produced by the instructions on P for a given input $x_1 \ldots x_n$. The value of P for an input, hence whether it belongs to A, can be determined from the truth or falsehood of ℓ conditions as described above, each one given by a tuple of letters $a_1, \ldots, a_k$ and commutative languages $L_0, \ldots, L_k$.

For each such condition, we build an MA to check it as follows: The MA is the direct sum of $\binom{s}{k}$ MA's, one for each of the positions where the $a_0 \ldots a_k$ witnessing a factorization could appear. Each MA concurrently checks that each of the chosen positions contains the right a_i (when the input variable producing the corresponding element m_j is available) and concurrently checks whether the subword w_i between a_i and a_{i+1} is in the language L_i. Crucially, since L_i is in **Com**, membership of w_i in L_i can be computed by a fixed-width automaton, regardless of the order in which the variables producing w_i are read. The automaton produces 0 if this check fails for some i, and 1 otherwise. It can be checked that the resulting automaton for each choice has size polynomial in s.

For each condition $L_0 a_1 L_1 \ldots a_k L_k$, counting the number of factorizations mod p amounts to taking the sum of the MA built for all possible guesses and adding them over $\mathbb{F}_p$.

To conclude the proof, take all MA's resulting from the previous construction and raise them to the p-th power. That increases their size by a power of p, and by Fermat's little theorem they become 0/1-valued. The boolean combination of several conditions can be then expressed by (a fixed number) of sums and products in $\mathbb{F}_p$, with polynomial blowup. ∎

We next note that several classes that were shown to be learnable by showing they were polynomially simulated by MA.

Theorem 19 *The following classes of boolean functions are polynomially simulated by programs over* $\mathbf{LG_p} \circledcirc_m \mathbf{Com}$*, hence are learnable from Membership and Equivalence queries as MA:*

- *Polynomials over* $\mathbb{F}_p$ *(when viewed as computing boolean functions)*

- *Unambiguous DNF functions; these include decision trees k-term DNF for constant k.*

- *constant-degree, depth-three, $\Sigma\Pi\Sigma$ arithmetic circuits [KS06], when restricted to boolean functions.*

An interesting case is that of $O(\log n)$-term DNF. It was observed in [Kus97] $c \log n$-term DNF can be rewritten into DFA of size roughly n^c, hence learned from Membership and Equivalence queries by Angluin's algorithm [Ang87]. It is probably false that $c \log n$-term DNF can be simulated by programs over a fixed monoid in $\mathbf{LG_p} \circledcirc{m} \mathbf{Com}$. However, we note that for every c and n, we note that for every c and n, $c \log n$-term DNF is simulated by a monoid of size n^c that is easily computed from c and n and commutative, hence in $\mathbf{LG_p} \circledcirc{m} \mathbf{Com}$. Indeed, let $M_{c,n}$ be the monoid consisting of all bit vectors of length $c \log n$, with bitwise-OR as monoid operation. Then a program over $M_{c,n}$ simulates a $c \log n$-term DNF by reading $x_1, \ldots x_n$ in sequence and upon reading each x_i emitting the vector that has 0 in the positions corresponding to terms proved to be false by the value of x_i that is read. Then, a given DNF is true on an assignment if the product in $M_{c,n}$ of all vectors emitted in this way is not the all-0 vector. Thus, although $O(\log n)$-term DNF is not strictly speaking captured by our framework, it is by a very uniform extension of it.

Finally, we conjecture that $\mathbf{LG_p} \circledcirc{m} \mathbf{Com}$ is the largest class of monoids that are polynomially simulated by MA, hence, the largest class we can expect to learn from MA within our algebraic framework:

Conjecture 20 *If a monoid M is not in $\mathbf{LG_p} \circledcirc{m} \mathbf{Com}$, then programs over M are not polynomially simulated by MA's over $\mathbb{F}_p$.*

The proof of this conjecture should be within reach given the characterization given in [TT06] of the monoids that are *not* in $\mathbf{LG_p} \circledcirc{m} \mathbf{Com}$: this happens iff the monoid is divided by either the monoids U or BA_2 described before, or by a so-called T_q monoid or by a monoid whose commutator subgroup is not a p-group. It would thus suffice to show that programs over these four kinds of monoids cannot always be polynomially simulated by MA over $\mathbb{F}_p$.

6 Conclusions and Future Work

We have explained within three algorithmic paradigms

a large fraction of known learning results on boolean functions that do not use threshold gates, restrictions on reads, or monotonicity conditions. Algebraic interpretations, in terms of programs over monoids, of each of the three paradigms sheds some light on the their limitations.

This approach could be taken as a framework to systematize results in other learning models, such as learning in the presence of noise, attribute-efficient learning, or learning classes defined using monotonicity (by using existing notions of "monotone programs over monoids").

References

[AMSV09] Jorge Almeida, Stuart W. Margolis, Benjamin Steinberg, and Mikhail V. Volkov. Representation theory of finite semigroups, semigroup radicals and formal language theory. *Trans. Amer. Math. Soc.*, 3612:1429–1461, 2009.

[AMV05] Jorge Almeida, Stuart W. Margolis, and Mikhail V. Volkov. The pseudovariety of semigroups of triangular matrices over a finite field. *RAIRO - Theoretical Informatics and Applications*, 39(1):31–48, 2005.

[Ang87] D. Angluin. Learning regular sets from queries and counterexamples. *Information and Computation*, 75:87–106, 1987.

[Ang88] D. Angluin. Queries and concept learning. *Machine Learning*, 2:319–342, 1988.

[Bar89] D.A. Barrington. Bounded-width polynomial-size branching programs recognize exactly those languages in NC^1. *Journal of Computer and System Sciences*, 38:150–164, 1989.

[BBB+00] A. Beimel, F. Bergadano, N.H. Bshouty, E. Kushilevitz, and S. Varricchio. Learning functions represented as multiplicity automata. *Journal of the ACM*, 47:506–530, 2000.

[BBTV97] F. Bergadano, N.H. Bshouty, C. Tamon, and S. Varricchio. On learning branching programs and small depth circuits. In *Proc. 3rd European Conference on Computational Learning Theory (EuroCOLT'97), Springer-Verlag LNCS*, volume 1208, pages 150–161, 1997.

[BCJ93] Avrim Blum, Prasad Chalasani, and Jeffrey C. Jackson. On learning embedded symmetric concepts. In *COLT*, pages 337–346, 1993.

[BK] Nader H. Bhshouty and Eyal Kushilevitz. Learning from membership queries / online learning. Course notes in N. Bshouty's homepage.

[Blu92] Avrim Blum. Rank-r decision trees are a subclass of r-decision lists. *Information Processing Letters*, 42(4):183–185, 1992.

[BST90] D.A. Mix Barrington, H. Straubing, and D. Thérien. Non-uniform automata over groups. *Information and Computation*, 89:109–132, 1990.

[BT88] D.A. Mix Barrington and D. Thérien. Finite monoids and the fine structure of NC^1. *Journal of the ACM*, 35:941–952, 1988.

[BT94] Richard Beigel and Jun Tarui. On ACC. *Computational Complexity*, 4:350–366, 1994.

[BV96] F. Bergadano and S. Varricchio. Learning behaviors of automata from multiplicity and equivalence queries. *SIAM Journal on Computing*, 25:1268–1280, 1996.

[CGPT06] Arkadev Chattopadhyay, Navin Goyal, Pavel Pudlák, and Denis Thérien. Lower bounds for circuits with mod_m gates. In *FOCS*, pages 709–718, 2006.

[CKK+07] Arkadev Chattopadhyay, Andreas Krebs, Michal Koucký, Mario Szegedy, Pascal Tesson, and Denis Thérien. Languages with bounded multiparty communication complexity. In *STACS*, pages 500–511, 2007.

[EH89] Andrzej Ehrenfeucht and David Haussler. Learning decision trees from random examples. *Information and Computation*, 82(3):231–246, 1989.

[GT03] Ricard Gavaldà and Denis Thérien. Algebraic characterizations of small classes of boolean functions. In *STACS*, pages 331–342, 2003.

[GTT06] Ricard Gavaldà, Pascal Tesson, and Denis Thérien. Learning expressions and programs over monoids. *Inf. Comput.*, 204(2):177–209, 2006.

[HS07] Lisa Hellerstein and Rocco A. Servedio. On pac learning algorithms for rich boolean function classes. *Theor. Comput. Sci.*, 384(1):66–76, 2007.

[HSW90] David P. Helmbold, Robert H. Sloan, and Manfred K. Warmuth. Learning nested differences of intersection-closed concept classes. *Machine Learning*, 5:165–196, 1990.

[KLPV87] Michael J. Kearns, Ming Li, Leonard Pitt, and Leslie G. Valiant. On the learnability of boolean formulae. In *STOC*, pages 285–295, 1987.

[KS06] Adam R. Klivans and Amir Shpilka. Learning restricted models of arithmetic circuits. *Theory of Computing*, 2(1):185–206, 2006.

[Kus97] Eyal Kushilevitz. A simple algorithm for learning $o(logn)$-term dnf. *Inf. Process. Lett.*, 61(6):289–292, 1997.

[PT88] P. Péladeau and D. Thérien. Sur les langages reconnus par des groupes nilpotents. *Compte-rendus de l'Académie des Sciences de Paris*, pages 93–95, 1988. Translation to English as ECCC-TR01-040, Electronic Colloquium on Computational Complexity (ECCC).

[Riv87] Ronald L. Rivest. Learning decision lists. *Machine Learning*, 2(3):229–246, 1987.

[Sch76] M.P. Schützenberger. Sur le produit de concaténation non ambigu. *Semigroup Forum*, 13:47–75, 1976.

[She08] Alexander A. Sherstov. Communication lower bounds using dual polynomials. *Bulletin of the EATCS*, 95:59–93, 2008.

[Sim95] Hans-Ulrich Simon. Learning decision lists and trees with equivalence-queries. In *EuroCOLT*, pages 322–336, 1995.

[Tes03] Pascal Tesson. *Computational Complexity Questions Related to Finite Monoids and Semigroups*. PhD thesis, School of Computer Science, McGill University, 2003.

[TT04] Pascal Tesson and Denis Thérien. Monoids and computations. *Intl. Journal of Algebra and Computation*, 14(5-6):801–816, 2004.

[TT05] Pascal Tesson and Denis Thérien. Complete classifications for the communication complexity of regular languages. *Theory Comput. Syst.*, 38(2):135–159, 2005.

[TT06] Pascal Tesson and Denis Thérien. Bridges between algebraic automata theory and complexity. *Bull. of the EATCS*, 88:37–64, 2006.

[Val84] L.G. Valiant. A theory of the learnable. *Communications of the ACM*, 27:1134–1142, 1984.

[Wei87] Pascal Weil. Closure of varieties of languages under products with counter. *J. of Comp. Syst. Sci.*, 2(3):229–246, 1987.

Appendices

Appendix A: Proof of Theorem 8

We use the following lemma, showing that Abelian groups have the narrowing property. The proof is a very simple case of the proof in [PT88] for all nilpotent groups, and is omitted in this version.

Lemma 21 *Let f and g be two functions computed by programs over an Abelian group G. Then either f and g are identical, or they differ on an assignment of weight at most $|G|^2$.*

Now fix an Abelian group G, of cardinality g for short. Let (P, A) denote the target program, computing a boolean function f. We first note that we can assume w.l.o.g. that P is a list of instructions of the form $(1, e, \alpha_1)(2, e, \alpha_2) \dots (n, e, \alpha_n)$, where e is the identity of G. This is because we can first reorder instructions according to the variable they read, merge instructions reading the same variable, factor out the product α of the constants emitted by each instruction upon reading 0 values, then assume that that α is the identity by multiplying the accepting set by α^{-1}. So learning the target function amounts to learning the sequence of constants $\alpha_1, \dots \alpha_n$ and the set $A \subseteq G$.

Second, note that (P, A) induces an equivalence relation $\sim$ on the set $[n]$ by setting $i \sim j$ iff $\alpha_i = \alpha_j$.

The algorithm first asks for the value of f on all assignments of weight at most g^2+1, of which there are about n^{g^2+1}. Let S^+ be that set of assignments, and S the subset of S^+ formed by assignments of weight at most g^2. The algorithm then computes another equivalence relation $\approx$ in the following way: for every i, and $a \in S$, $b \in \{0, 1\}$, let $a[i := b]$ be the assignment obtained by setting $x_i = b$ in a; we extend the notation to $a[i := b; j := c]$ and so on with the obvious meaning. Then $i \approx j$ iff either $i = j$ or for every $a \in S$ we have $f(a[i := 1; j := 0]) = f(a[i := 0; j := 1])$; in words, whether setting x_i and resetting x_j, or the other way round, never makes a difference with respect to Membership in small weight assignments. This relation can be computed in polynomial time from the answers to S^+.

Clearly, $\approx$ is reflexive and symmetric. To see that it is transitive, assume $i \approx j$ and $j \approx k$, for distinct i, j, k. Take any assignment $a \in S$, and say that $x_j = 0$ in S. Then we have

$$\begin{aligned} f(a[i := 1;\, k := 0]) &= f(a[k := 0][i := 1;\, j := 0]) \\ &= f(a[k := 0][i := 0;\, j := 1]) \\ &= f(a[i := 0][k := 0;\, j := 1]) \\ &= f(a[i := 0][k := 1;\, j := 0]) \\ &= f(a[i := 0;\, k := 1]) \end{aligned}$$

and a similar argument holds when $x_j = 1$. Therefore $i \approx k$. Note also that if $\alpha_i = \alpha_j$, setting either x_i or x_j must have the same effect on the value of P on any assignment, and therefore $i \sim j$ implies $i \approx j$, which in turn means $|\approx| \leq |\sim| \leq g$.

Having computed $\approx$, then the algorithm exhaustively runs through all functions $f : |\approx| \to g$ and sets $B \subseteq G$ standing for the guesses of coefficients for each equivalence class and accepting sets. There are $g^g \cdot 2^g$ such guesses. For each of them, f, the algorithm builds a program (P_f, B), where in P_f the value of α_i is set to $f(r)$ for each i in the r-th equivalence class of $\approx$. The algorithm outputs any pair (P_f, B) such that the function computed by (P_f, B) equals $f(a)$ on every $a \in S$, i.e., agrees with the target on all of S. At least one such (P_f, B) exists: namely, the target program (P, A) which is of the form P_f by the fact that $\sim$ refines $\approx$. And by Lemma 21, any such pair (P_f, B) output by the program must agree with (P, A) not only on S but on all of $\{0,1\}^n$.

Appendix B: Proof of Theorem 9

We will use the following machinery from [BST90] to analyze the computing power of groups in $\mathbf{G_p} \star \mathbf{Ab}$, based on Fourier analysis on finite fields. Let F be a finite field of order at least 3. As is well known, F^* is a cyclic group under the field multiplication. Fix a generator g of this group. For the purposes of this paper, a linear form on boolean variables $x_1 \ldots x_n$ is an expression of the form $g^{\alpha_1 x_1} \cdot \cdots g^{\alpha_n x_n} = g^{\sum_i \alpha_i x_i}$. Observe that a linear form is never 0, no matter the value of $x_1, \ldots, x_n$; this is a crucial difference between polynomials and sums of linear forms that largely explains the lower bound and, intuitively, makes learnability much easier.

The following result in [BST90] translates programs over $\mathbf{G_p} \star \mathbf{Ab}$ to sums of linear characters over appropriate finite fields.

Fact 22 *[BST90] Let G be a monoid in $\mathbf{G_p} \star \mathbf{Ab}$. Then there is a finite field F such that programs over G are polynomially simulated by sums of linear forms over F. That is, for every program over G on n variables and length s there is a sum of at most poly(n, s) linear forms equivalent to it; i.e., this sum evaluates to 0 on an assignment a if the program rejects a, and to 1 if the program accepts a.*

In fact, a converse form of this result also holds, but we will not need it in this paper. Together with the following result it essentially provides the the lower bound 7, part (2).

Fact 23 *[BST90] For a fixed finite field F, any two sums of s linear characters are either equivalent or differ on an assignment of weight at most $c \log s$, for a constant $c = c(F)$.*

We now prove the learning result. Fix a group $G \in \mathbf{G_p} \star \mathbf{Ab}$ assume the target is computed by a program over G, and c be the constant given by Fact 23 for G. W.l.o.g., we will assume it is computed as a sum of at most s linear forms over a field F, where s is polynomially larger than the program's length. For brevity, we will from now on call sums of linear characters "polynomials", since they can be viewed as polynomials in the derived variables g^{x_i}; hence s is the number of terms in the target polynomial.

The learning algorithm is as follows:

1. Read n and s;
2. Let W be the set of strings of length at most n and weight at most $c \log(s^2\ n)$;
3. Ask Membership queries $f(x \cdot y)$ for every pair of strings $x, y \in W$ with $|x \cdot y| = n$;
4. Build (as described below) an MA over F out of the answers, and output it;

To describe how the MA is built, we need some terminology.

The Hankel matrix of function f is the matrix whose rows and columns are indexed by $\{0,1\}^\star$ and such that $H[x,y] = f(xy)$. For this to make sense, we define f to evaluate to 0 on all bit vectors whose length is not n

For every string x, $row(x)$ and $col(x)$ denote the row and column of H indexed by x. Equivalently, $row(x)$ (resp., $col(x)$) is the function mapping each y to $row(x)(y) = f(x \cdot y)$ (resp., $col(x)(y) = f(y \cdot x)$). For a subset $S \subseteq \{0,1\}^\star$, we denote by $row_S(x)$ and $col_S(x)$ the restriction of the functions above to arguments in S (equivalently, the subvector of $row(x)$ / $col(x)$ indexed by strings in S).

The MA is built as follows. Intuitively, we use some strings in W to stand for states of the MA. The states are naturally grouped in $n+1$ "levels" $\ell = 0 \ldots n$, corresponding to their distance from the initial state, but also to the lengths of the strings labelling the states.

1. For every $\ell = 0 \ldots n$, choose a maximal set $S^\ell = \{s_1^\ell, \ldots, s_{r_\ell}^\ell\}$ of strings in W of length ℓ such that the r_ℓ vectors $row_W(s_i^\ell)$ are linearly independent in F. That is, S^ℓ forms a basis for the set of vectors $\{\, row_W(x) \mid x \in W,\ \ |x| = \ell \,\}$. Note that $r_0 = 1$ and that $s_1^0 = \epsilon$.

2. For $\ell = 0 \ldots n$, choose a minimal set $E^\ell = \{e_1^\ell, \ldots, e_{r_\ell}^\ell\}$ of strings in W of length $n - \ell$ such that the submatrix $H[S^\ell, E^\ell]$ has full rank r_ℓ. Note that indeed we must have $|E^\ell| = |S^\ell| = r_\ell$, and that we have $|s_i^\ell\ e_j^\ell| = n$ for every i, j.

3. For every s_i^ℓ and every $a \in \{0, 1\}$, express $row_{E^{\ell+1}}(s_i^\ell \cdot a)$ as a linear combination of the $row_{E^{\ell+1}}(s_j^{\ell+1})$, with coefficients $\mu_{\ell,i,a,j}$. This is always possible because $H[S^\ell, E^\ell]$ has full rank.

4. The MA has a state named s_j^ℓ for every $\ell = 0 \ldots n$ and $j = 1 \ldots r_\ell$. The label of the transition from state s_i^ℓ to state $s_j^{\ell+1}$ with letter a is the coefficient $\mu_{\ell,i,a,j}$.

5. The initial state of the MA is $s_1^0 = \epsilon$. The final states are those $s_j^n \in S^n$ such that $f(s_j^n) = 1$.

Note that all the steps above can be carried out in time polynomial in $|W|$ from the answers of the queries of the algorithm. The running time of the algorithm is thus polynomial in $|W|$, which is $n^{O(\log(s^2 n))}$.
We make the following claims:

Claim $\sum_{\ell=0}^{n} r_\ell \leq n \cdot s$. Thus, for every ℓ, $r_\ell \leq n \cdot s$.
Proof. This is because if f is computed by an s-term polynomial as described above, it is computed by an MA of size at most $n \cdot s$. By the known result that the size of the smallest MA for a function is the rank of its Hankel matrix, there are at most ns linearly independent rows $row(x)$ in H. ∎

The following claim is the one where we exploit that the target is a sum of linear characters, rather than an arbitrary MA:

Claim For every $y \in W$ having length $n - \ell$, $col(y)$ is a linear combination of the set of vectors $col(e_j^\ell)$. In fact, the coefficients of this linear combination are exactly those expressing $col_{S^\ell}(x)$ as a linear combination of the r_ℓ vectors $col_{S^\ell}(e_j^\ell)$.

Proof. To give the coefficients a name, let $\beta_{y,j}$ be such that

$$col_{S^\ell}(y) = \sum_{j=1}^{r_\ell} \beta_{y,j}\ col_{S^\ell}(e_j^\ell).$$

Now observe that the function $col_{S^\ell}(y)$ is computed by a polynomial of size at most s, namely, the one obtained by setting the last $n-\ell$ arguments of P as in y. The same happens with each $col_{S^\ell}(e_j^\ell)$ function, and therefore the right hand side of the equation above is itself computed by a polynomial of size $s \cdot r_\ell \leq s^2 n$. Suppose that these two polynomials differ on any argument whatsoever. Then, since they have both size $\leq s^2 \cdot n$, by Fact 2 they must disagree on some argument in W. But if they disagree on some $x \in W$, then x witnesses that $col_W(y)$ is linearly independent of the set of vectors $col_W(e_j^\ell)$, because the unique coefficients that make the vectors equal on S^ℓ fail on x. This contradicts the fact that $H[S^\ell, E^\ell]$ has rank r_ℓ. ∎

By the same proof we can show a similar claim for every $x \in W$ of length ℓ and vectors $row(s_j^\ell)$. But, using Claim 6, we can in fact extend the claim to every x, not necessary in W. The proof is similar to an analogous claim in the learning algorithm for MA, and omitted in this version.

Claim For every $x \in \{0,1\}^\star$ of length $|x| = \ell$, $row(x)$ is a linear combination of the set of vectors $row(s_j^\ell)$. In fact, the coefficients of this linear combination are exactly those expressing $row_{E^\ell}(x)$ as a linear combination of the r_ℓ vectors $row_{E^\ell}(e_j^\ell)$.

Proof. For any x of length ℓ and y of length $n-\ell$, let us again give names to the coefficients: α and β are the sets of coefficients such that

$$row_{E^\ell}(x) = \sum_{j=1}^{r_\ell} \alpha_{x,j}\ row_{E^\ell}(s_j^\ell) \quad (1)$$

$$col_{S^\ell}(y) = \sum_{k=1}^{r_\ell} \beta_{y,k}\ col_{S^\ell}(e_k^\ell) \quad (2)$$

Note that the α's and β's exist and are unique since $H[S^\ell, E^\ell]$ has full rank. To prove the claim, fix x and assume that

$$row(x) \neq \sum_{j=1}^{r_\ell} \alpha_{x,j}\ row(s_j^\ell)$$

As in the proof of Claim 6, both left-hand side and right-hand side are functions computed by polynomials of size $\leq s^2 n$, so if they differ anywhere they must differ on some $y \in W$. That is, for some $y \in W$ we have

$$f(xy) = row(x)(y) \quad \neq \quad \sum_{j=1}^{r_\ell} \alpha_{x,j} \; row(s_j^\ell)(y) \tag{3}$$

On the other hand, by Claim 6 we have

$$f(xy) = col(y)(x) \quad = \quad \sum_{k=1}^{r_\ell} \beta_{y,k} \; col(e_k^\ell)(x) \tag{4}$$

We now show that this cannot be the case by showing that the right-hand sides of equations (3) and (4) are in fact equal, which contradicts the fact

that they are respectively different from and equal to $f(xy)$. Indeed,

$$\sum_{j=1}^{r_\ell} \alpha_{x,j}\ row(s_j^\ell)(y) = \quad \text{(by def. of } row \text{ and } col\text{)}$$

$$\sum_{j=1}^{r_\ell} \alpha_{x,j}\ col(y)(s_j^\ell) = \quad \text{(because } s_j^\ell \in S^\ell\text{)}$$

$$\sum_{j=1}^{r_\ell} \alpha_{x,j}\ col_{S^\ell}(y)(s_j^\ell) = \quad \text{(by def. of } \beta\text{, equation (2))}$$

$$\sum_{j=1}^{r_\ell} \alpha_{x,j} \left(\sum_{k=1}^{r_\ell} \beta_{y,k}\ col_{S^\ell}(e_k^\ell)(s_j^\ell) \right) = \quad \text{(rearranging the sum)}$$

$$\sum_{k=1}^{r_\ell} \beta_{y,k} \left(\sum_{j=1}^{r_\ell} \alpha_{x,j}\ col_{S^\ell}(e_k^\ell)(s_j^\ell) \right) =$$

(by def. of col and row and because $e_k^\ell \in E^\ell$ and $s_j^\ell \in S^\ell$)

$$\sum_{k=1}^{r_\ell} \beta_{y,k} \left(\sum_{j=1}^{r_\ell} \alpha_{x,j}\ row_{E^\ell}(s_j^\ell)(e_k^\ell) \right) = \quad \text{(by def. of } \alpha\text{, equation (1))}$$

$$\sum_{k=1}^{r_\ell} \beta_{y,k}\ row_{E^\ell}(x)(e_k^\ell) = \quad \text{(because } e_k^\ell \in E^\ell\text{)}$$

$$\sum_{k=1}^{r_\ell} \beta_{y,k}\ row(x)(e_k^\ell) = \quad \text{(by def. of } row \text{ and } col\text{)}$$

$$\sum_{j=1}^{r_\ell} \beta_{y,k}\ col(e_j^\ell)(x).$$

which proves our claim. ∎

Finally, using Claim 6 we can show that the MA produced by the algorithm is correct. Again, the proof is similar to the one for the MA learning algorithm and omitted in this version.

Claim The MA produced by the algorithm computes exactly f.
Proof.

We show how to compute $f(x)$ given the coefficients $\mu_{\ell,i,a,j}$ computed by the algorithm. For simplicity, we rename the states with a single subindex

(instead of by a pair (ℓ, i)), so the coefficients become $\mu_{i,a,j}$ (and it will happen that $\mu_{i,a,j} = 0$ for sure whenever $|s_j| \neq |s_i| + 1$.)

Note that computing $f(x)$ is equivalent to computing $row(\epsilon)(x)$. We show inductively how to compute $row(s_i)(z)$ for every suffix z of x every state s_i, by induction on $|z|$.

For $|z| = 0$, we have $row(s_i)(z) = row(s_i)(\epsilon) = f(s_i)$ which was a membership query we asked (this is the reason why s_i is a final state in the MA iff $f(s_i) = 1$).

For $z = az'$, and using Claim 6, we have

$$\begin{aligned} row(s_i)(z) &= row(s_i)(az') = row(s_i\ a)(z') = \\ &= \left(\sum_j \mu_{i,a,j}\ row(s_j)\right)(z') = \sum_j \mu_{i,a,j}\ \ (row(s_j)(z')) \end{aligned}$$

and we can now compute $row(s_i)(z)$ because inductively we have computed $row(s_j)(z')$ for suffix z' and every s_j.

Finally, we have $f(x) = row(\epsilon)(x)$ where $\epsilon = s_1^0$ is the initial state. So the computation takes time polynomial in n and s. ∎