Learning Expressions and Programs over Monoids

Ricard Gavaldà*
Univ. Politècnica de Catalunya

Pascal Tesson †
Université Laval

Denis Thérien ‡
McGill University

June 21st, 2005

Abstract

We study the problem of learning an unknown function represented as an expression or a program over a known finite monoid. As in other areas of computational complexity where programs over algebras have been used, the goal is to relate the computational complexity of the learning problem with the algebraic complexity of the finite monoid. Indeed, our results indicate a close connection between both kinds of complexity. We focus on monoids which are either groups or aperiodic, and on the learning model of exact learning from queries. For a group G, we prove that expressions over G are efficiently learnable

*Department of Software (LSI), Universitat Politècnica de Catalunya. Jordi Girona Salgado 1–3, E-08034 Barcelona, Spain. Email: gavalda@lsi.upc.edu.

Partially supported by EU EP27150 (Neurocolt II), by the IST Programme of the EU under contract number IST-1999-14186 (ALCOM-FT), by CIRIT 1997SGR-00366, TIC2000-1970-CE, and by the Spanish Government PB98-0937-C04-04 (project FRESCO).

†Département d'Informatique et de Génie Logiciel, Université Laval, Pavillon Adrien-Pouliot. Université Laval (Québec), Canada, G1K 7P4. E-mail: pascal.tesson@ift.ulaval.ca. Work done while at McGill University.

‡School of Computer Science, McGill University. 3480 University, Montréal, Québec H3A 2A7 Canada. Email: denis@cs.mcgill.ca.

Supported by NSERC, FCAR, and the von Humboldt Foundation.

if G is nilpotent, and impossible to learn efficiently (under cryptographic assumptions) if G is nonsolvable. We present some results for restricted classes of solvable groups, and point out a connection between their efficient learnability and the existence of lower bounds on their computational power in the program model. For aperiodic monoids, our results seem to indicate that the monoid class known as **DA** captures exactly learnability of expressions by polynomially many Evaluation queries. When using programs instead of expressions, we show that our results for groups remain true, while the situation is quite different for aperiodic monoids.

1 Introduction

Formal models of the process of learning have been proposed since the 60s to give mathematical foundation to machine learning tasks, mostly to concept learning. The first models, such as Gold's identification in the limit, were of recursion-theoretic flavor and did not emphasize efficient use of computational resources. In the mid 80s, several models that take time and memory into consideration were proposed, thus allowing the use of concepts and tools from computational complexity theory in the study of the complexity of learning.

The resulting area, known as Computational Learning Theory, has produced an important number of results stating that specific classes of functions are or are not efficiently learnable in rigorously defined learning models. But, in general, the properties that determine whether a class is easy or hard to learn are not yet well identified. We propose an algebraic approach that, in the long run, might help in clarifying such properties.

Programs over finite monoids and other algebraic structures are models of computation that have been successfully used to expose the deep reasons why some computational problems can or cannot be solved within certain resource bounds. In this paper we initiate the study of formal models of function learning from an algebraic point of view, i.e., we would like to determine the complexity of learning a class of functions from the classes of algebras that are powerful enough to compute the class of functions (in the program model or related ones). We concentrate on the case where the algebra M consists of an associative operation with an identity, i.e., when M is a monoid.

So far, programs over monoids have been studied mostly as devices to compute boolean functions, and many circuit complexity classes have been shown to admit characterizations as those problems solved by programs over particular algebraic structures [Bar89, BT88, BST90]. We look instead at programs over M as computing functions from M^n into M. *Expressions* over M are a particular type of programs that appear very naturally in this context.

We study the problem of learning an unknown *target* function from M^n to M, for a fixed and known finite monoid M. It is assumed only that the function is computed by some expression or program on n variables over M. We work mostly in Angluin's query-based model of exact learning [Ang87, Ang88], where algorithms can ask Evaluation queries, Equivalence queries, or both.

As a general result, we show that the problem of learning expressions over any monoid reduces to that that of learning boolean functions computable by programs over the monoid. That is, if boolean programs over M are polynomial-time learnable, so are expressions over M. The converse is not true in general, and we provide specific examples where learning boolean functions is exponentially harder than learning expressions.

We present several results on the complexity of learning expressions and programs over specific classes of monoids. We concentrate on monoids that are either *groups* or *aperiodic*, as these two classes are known to be the building blocks of all monoids via the so-called *semidirect product* operation [Eil76].

In the setting we consider, a monoid M is fixed and known to the learning algorithm from the start. Thus, when we say, e.g., "expressions over Abelian groups are learnable" we mean that for every Abelian group G there is an algorithm, specific to G, that learns expressions over G.

For the case expressions over groups, we prove:

- Expressions over Abelian groups are polynomial-time learnable both from a linear number of Evaluation queries and from a linear number of Equivalence queries.
- Nilpotent groups, a generalization of Abelian groups, are learnable in polynomial time from Evaluation queries.
- Expressions over solvable groups formed by iterated semidirect products of cyclic groups are polynomial-time learnable from Evaluation and Equivalence queries.

- Expressions over a larger subclass of solvable groups can be identified by a probabilistic strategy using polynomially many Evaluation queries, and by a deterministic strategy using quasipolynomially many Evaluation queries (though possibly in exponential time). We show that similar results will hold for any group for which we can prove a certain type of lower bounds on program size; that is, proving limitations in computing power provides upper bounds on learning complexity. These two properties are restatements of the combinatorial simplicity of the monoid.

- Expressions over nonsolvable groups are not learnable unless NC^1 circuits also are, even in the very strong learning model of PAC-prediction with Membership queries [AK95]. Recall that, under plausible cryptographic assumptions, NC^1 circuits are not learnable in this or any other standard model of learning [AK95].

In the aperiodic case, our results involve the class of aperiodic monoids known as **DA** [Sch76, TT02]. For some algorithmic problems on monoids, it is known that feasibility depends essentially on membership in **DA**. For example, the membership problem is known to be PSPACE-complete for any aperiodic monoid outside of **DA** [BMT92]. Also, the word problem for an aperiodic monoid can be resolved in sublinear communication complexity (in the 2-player setting) if and only if the monoid belongs to **DA** [RTT98]. Our results are:

- Expressions over **DA** monoids are identifiable from polynomially many Evaluation queries (though possibly in exponential time) and learnable from Equivalence queries in polynomial time.

- It is known that there are exactly two minimal aperiodic monoids not belonging to **DA**. We show that expressions over any of these two minimal monoids are not learnable with subexponentially many Evaluation queries, even with arbitrary computation time. We conjecture that the same is true for any monoid outside of **DA**, given that every monoid outside of **DA** is divided by at least one of these two minimal monoids.

- For two important subclasses of **DA**, idempotent R-trivial monoids and commutative aperiodic monoids, we give actual polynomial-time algorithms for learning expressions using Evaluation queries.

- In addition, expressions over an idempotent commutative monoid are learnable from Equivalence queries that are themselves expressions over the monoid. Interestingly, this class of monoids is precisely the intersection of the two classes mentioned in the previous point, that is, idempotent R-trivial and aperiodic commutative. On the other hand, learning expressions over non-idempotent, aperiodic monoids requires exponentially many Equivalence queries if these must also be expressions over the same monoid.

Besides learning expressions, we also study the learnability of functions $M^n \mapsto M$ represented as programs over M, instead of expressions.

- For groups, we show that all our results about expressions also apply to programs, with only polynomial differences in running time and number of queries.

- We show, however, that for every monoid which is not a group, exponentially many Evaluation queries are required to learn programs. This implies an exponential gap in the complexity of learning expressions and programs for some aperiodic monoids.

Certainly the picture is still partial, with many upper and lower bounds missing. But our results seem to indicate a very close connection between the complexity of the monoid (in the algebraic sense) and its learning complexity.

Finally, let us comment on the relevance of these results for the mainstream of computational learning theory. A good deal of the effort in this theory has been on learning classes of boolean functions, and especially those inside NC^1 since it seems hopeless to try to learn larger classes. As mentioned, many of the central complexity classes definable by small-depth circuits can be also defined by programs over central classes of monoids. For example, polynomial-length programs over aperiodic monoids compute exactly the functions in AC^0, polynomial-length programs over solvable monoids compute the functions in CC^0 (polynomial-size circuits made of MOD_q gates for a fixed q), and polynomial-length programs over nonsolvable groups (or monoids) compute all functions in NC^1 [Bar89, BT88].

On the other hand, in the learning context, various small fragments of AC^0 and CC^0 are known to be learnable, and NC^1 is known not to be learnable under plausible complexity assumptions. In many cases, progress on learning is barred by the fact that we do not have a basic understanding of the computing power of larger fragments of these classes, e.g., we cannot prove lower bounds on circuit size. We believe that the study of the learnability of expressions is interesting because it provides a different perspective on learning problems near the borderline of current knowledge.

The paper is organized as follows. Section 2 provides the necessary background on algebra and learning theory. Section 3 presents general results on learnability of expressions over monoids. Sections 4, 5, 6, and 7 present our results on expressions over Abelian, nilpotent, solvable, and nonsolvable groups, respectively. Section 8 deals with expressions over aperiodic monoids, and finally Section 9 deals with learnability of programs over monoids.

2 Preliminaries

2.1 Algebraic Background and Notation

A *monoid* is a pair $(M, \cdot)$ where M is a set and $\cdot$, the product over M, is a binary associative operation on M with an identity element, denoted e. In this paper we consider only finite monoids.

When the product operation $\cdot$ is clear or implicit, we often refer to M as the monoid instead of writing $(M, \cdot)$. Conversely, when several monoids are in play, we often write $\cdot_M$ and e_M to denote the product in M and the identity of M respectively. We use additive notation when the monoid is an Abelian group G, that is, we write $+$ (or $+_G$) for the group operation and 0 (or 0_G) for its identity.

The direct product of monoids M and N is $(M \times N, \cdot)$ where $(m_1, n_1) \cdot (m_2, n_2) = (m_1 \cdot_M m_2, n_1 \cdot_N n_2)$.

A submonoid of $(M, \cdot_M)$ is a pair $(N, \cdot_N)$ such that it is itself a monoid, $N \subseteq M$, and $\cdot_N$ is the restriction of $\cdot_M$ on N. Note that we deviate from the usual definition of submonoid in that the identity of N is not required to be that of M; this may be the case for nongroups.

A monoid $(N, \cdot_N)$ is a morphic image of monoid $(M, \cdot_M)$ if there is a morphism from M to N, i.e., an application $\phi : M \mapsto N$ mapping the identity of M to the identity of N and such that $\phi(x \cdot_M y) = \phi(x) \cdot_N \phi(y)$.

A *group* is a monoid where each element has a (two-sided) inverse. A monoid is *aperiodic* if it has no submonoid which is a non-trivial group. Equivalently, a monoid M is aperiodic if there exists an integer t such that, for all $m \in M$, we have $m^t = m^{t+1}$.

A monoid M *divides* a monoid N if M is a homomorphic image of a submonoid of N. A *pseudo-variety* of finite monoids is a class of monoids closed under finite direct product and taking submonoids and homomorphic images (in other words, under division).

2.2 Expressions and Programs over Monoids

We consider two computation models based on products over a monoid: expressions and programs.

An *expression* over a monoid M and variables $x_1, \ldots, x_n$ is a string over the alphabet $M \cup \{x_1, \ldots, x_n\}$. Such an expression defines quite naturally a function from M^n to M: to evaluate (the function represented by) the expression over a vector or assignment $(w_1, w_2, \ldots, w_n)$ in M^n, replace each occurrence of each variable x_i in the expression with the value w_i, then multiply the elements in the resulting string in $M^\star$ to obtain a single value in M. For example, assume that a, b, c, d are four elements in M. Then the value of expression $ax_2x_1x_2bx_3cx_1x_3d$ on the assignment (c, b, a) is the element $a \cdot b \cdot c \cdot b \cdot b \cdot a \cdot c \cdot c \cdot a \cdot d$, where $\cdot$ is the product in M. We write $|E|$ to denote the length of the expression E.

Expressions are a particular case of programs. A *program* over M with domain D is a list of instructions of the form (i, f), where $i \in \{1, \ldots, n\}$ and f is a function $D \mapsto M$. Instruction (i, f) is interpreted as follows: read the value of variable x_i and append $f(x_i)$ to the string of monoid elements to be multiplied. Hence, expressions are programs over the domain M whose instructions use only constant functions and the identity function.

Previously, programs have been used mostly to compute boolean functions [Bar89, BT88]. In these *boolean programs*, the domain D is {true,false} and M is partitioned into *True* and *False* sets to interpret a boolean result. Expressions could also be used to compute boolean functions, say by encoding true and false inputs by distinct elements of the monoid. Here we will be mainly interested in programs and expressions computing functions from M^n into M, rather than boolean functions.

2.3 Learning Expressions over Monoids: Problem Statement

In this section we define the learning problem that we study. We use mostly Angluin's query-based model of learning [Ang87, Ang88], although Valiant's

PAC model [Val84] or related ones are mentioned occasionally. All definitions are standard for function learning in these models, although we give them only in the terms of our specific problem, learning expressions over monoids. For background in computational learning theory, the reader is referred to the books, surveys, and bibliography in the learning theory server `www.learningtheory.org`.

We define the task of learning an expression; the more general one of learning a program is analogous.

The task of a learning algorithm (or *learner*) is to find out a *target function* from M^n to M fixed by a *teacher* in an adversarial way. The function is assumed to be computable by an expression over M and variables x_1, ..., x_n. Not all variables necessarily occur in the expression. The learning algorithm is initially given n and a number m guaranteed to be no smaller than the length of a shortest expression computing the target function[1]. The monoid M is fixed and known both to the teacher and the learning algorithm.

The learning algorithm must produce some representation or encoding of the function from M^n into M computed by the target expression. In particular, this representation may or may not itself be an expression over M. This point is discussed later.

In order to learn, teacher and learner exchange information on the target function; more precisely, in this model the learner is allowed to formulate *queries* that the teacher answers. The types of queries used in this paper will be defined later.

The resources used by the algorithm are measured as a function of n and m. The class of expressions over M is learnable in time $t(n,m)$ if there is an algorithm that learns every expression over M in time $t(n,m)$. In particular, we study mostly whether expressions over a class of monoids are *polynomial-time learnable*, meaning whether for each monoid M in the class there is a polynomial $p(n,m)$ such that expressions over M are learnable in time $p(n,m)$.

Similarly, we say that expressions over a monoid M are *identifiable* with interaction $s(n,m)$ if there is an algorithm that learns every expression over M using an amount of interaction $s(n,m)$ with the teacher (and arbitrary computation time). The meaning of "amount of interaction" can only be made formal once the allowed types of queries and answers are fixed,

[1]Symbols n and m will by default have these meanings in the paper, i.e., the number of variables and an upper bound on the length of shortest expression for the target function.

but in general it has to be bounded by the number of bits of information exchanged by the teacher and the learning algorithm. Identifiability thus represents the information-theoretic cost of solving a learning task, ignoring the computational complexity of the problems that the learning algorithm has to solve internally at each stage of the process.

Let us stress that we assume that the monoid M is fixed and known to the learning algorithm, and thus we regard $|M|$ as a constant. We present some algorithmic schemes that learn whole classes of monoids and whose running time is exponential (or more) in the size of the monoid. We still call these algorithms "polynomial-time" as long as their dependence on n and m is polynomial. In some cases, we can show a stronger notion of polynomial-time learnability for a class of monoids, which we can regard as "uniform learnability": the learning algorithm receives, as part of its input, a description of some monoid M in the class (e.g., as its multiplication table), and it must learn the target function, an expression over M, in time polynomial in n, m, and $|M|$. This implies, in particular, that all structural information about M that is required for learning can be obtained from its description in polynomial time.

We consider algorithms using two types of queries that are standard in the exact learning model: Evaluation queries and Equivalence queries. Let the target f be a function from set A to set B. In an Evaluation query, the learning algorithm produces an element $a \in A$ and the teacher must return $f(a)$.[2] In an Equivalence query, the learning algorithm produces some representation of a function $h : A \mapsto B$ which is interpreted as a *hypothesis* about the target function f; the teacher must return Yes if $h \equiv f$ on A, or else a counterexample: an element $a \in A$ such that $f(a) \neq h(a)$, together with the value of $f(a)$. The teacher answers faithfully but adversarially (i.e., it can choose the least helpful among all valid counterexamples).

One more distinction is necessary, especially in the presence of Equivalence queries. As said before, a learning algorithm assumes that its target function is computed by some expression over M, and has to come up with a representation or encoding of the target function. If it asks Equivalence queries, these must also be representations of functions $M^n \mapsto M$. We say that an algorithm learns *properly* if its final output and all Equivalence are themselves represented as expressions over M. One can also design learning

[2] Evaluation queries generalize Membership queries [Ang87, Ang88] for functions with non-binary range.

algorithms that use other, possibly more expressive, syntactic schemes for encoding functions $M^n \mapsto M$; in this case, learning is called *nonproper*. For example, to learn expressions over M we could imagine algorithms that produce expressions over a monoid N of which M is a submonoid, or circuits of some type that can somehow simulate expressions over M. Nonproper learning may sometimes make learning easier, either because it allows asking more informative Equivalence queries, or because it is computationally easier to reconstruct the target function or new queries from the answers to previous queries. In the following, when we say e.g. "expressions over M are learnable" we will mean nonproper learning; when proper learning is implied, we will explicitly say so.

3 General Results on Learning Expressions

Before investigating the learnability of specific classes of monoids, let us consider a few general questions.

There are several constructions for building monoids from other monoids. The most natural ones are those used to define monoid varieties, namely direct product, taking submonoids, and taking homomorphic images. It is a natural question whether learnability is preserved by these operations. We only have very partial answers so far.

Proposition 1. *1. For the three combinations of query types (Evaluation queries only, Equivalence queries only, or both) the following is true. If expressions over monoids M and N are polynomial-time learnable, then expressions over $M \times N$ are polynomial-time learnable (nonproperly).*

2. If expressions over M are polynomial-time learnable from Equivalence queries and N is a submonoid of M, then expressions over N are polynomial-time learnable (nonproperly) from Equivalence queries.

Proof. This proposition is proved by simulation arguments often used in learning theory and later in the paper.

Part (1): To simplify the proof, we assume that all algorithms ask at least one Equivalence query; for example, we can rewrite any algorithm so that, upon termination, it asks its output as an Equivalence query, whose answer will be Yes.

For $x = ((a_1, b_1), \ldots, (a_n, b_n)) \in (M \times N)^n$, let $\pi_1(x)$ and $\pi_2(x)$ denote the tuples $(a_1, \ldots, a_n) \in M^n$ and $(b_1, \ldots, b_n) \in N^n$, respectively. Let A_M and

A_N be polynomial-time learning algorithms for expressions over M and N, respectively. Let E be a target expression over $M \times N$ on n variables. To learn E, run A_M and A_N in parallel. When A_M asks a tuple $(a_1, \ldots, a_n) \in M^n$ for Evaluation, build the tuple $((a_1, e_N), \ldots, (a_n, e_N)) \in (M \times N)^n$, (recall that e_N is the identity for N) and ask it to the learner for E. If the answer is (a, b), return a as an answer to M. Evaluation queries posed by A_N are answered similarly.

When one of A_M and A_N asks an Equivalence query, stop the simulation until the other one is also asking an Equivalence query; call both queries h_M and h_N. Ask the pair (h_M, h_N) as a nonproper Equivalence query, with the intended meaning "is it true that on every input $x \in (M \times N)^n$, $E(x) = (h_M(\pi_1(x)), h_N(\pi_2(x))$?". If the answer is Yes, E has been learned nonproperly. If it is a counterexample $(x, E(x))$ (with $E(x) = (a, b)$), it must happen that $h_M(\pi_1(x)) \neq a$ or $h_n(\pi_2(x)) \neq b$. In the first case, return $(\pi_1(x), a)$ to h_M; otherwise, return $(\pi_2(x), b)$ to h_N.

We claim that a Yes answer is achieved in polynomial time. To see this, let E_M be an expression over M obtained by replacing every constant $(a, b) \in M \times N$ with a, and interpreting variables over M. Define an expression E_N over N analogously. Note that the $|E_M| = |E_N| = |E|$. The strategy above ensures that A_M (resp., A_N) receives answers consistent with a teacher for E_M (resp., for E_N), and that at least one of A_M and A_N makes progress at every query. Therefore, in polynomial time a query (h_M, h_N) is produced such that h_M is equivalent to E_M and h_N is equivalent to E_N. This query receives a Yes answer from the teacher for E.

Part 2: Let E be an expression over N, mapping N^n to N and A an algorithm that learns expressions over M. To learn E, run A. Although A will issue queries h mapping M^n to M, the teacher for E is concerned only with the value of h on inputs in N^n. Precisely, on any query h, the teacher will answer Yes if $h(x) = E(x)$ for all $x \in N^n$, and will otherwise return a counterexample $(x, u) \in N^n \times N$ with $u = E(x) \neq h(x)$; note that while necessarily $u = E(x) \in N$, $h(x)$ may well be in $M \setminus N$, but that is still an acceptable counterexample for A.

Note that the algorithm makes at most the same number of queries it makes to learn the target expression considered as an expression over M – it may stop earlier if it queries an expression equivalent to it on N^n. Thus it works in polynomial time. □

We do not know whether learnability under Evaluation queries is preserved by taking submonoids. The difficulty is that the algorithm for the larger monoid may ask queries in $M^n \setminus N^n$; the teacher, knowing only a target function $f : N^n \mapsto N$, may not be able to answer these queries. In fact, there may be several different extensions of f from N^n to M^n, all computable by expressions over M, so the Evaluation query itself may not be well defined in terms of f. For homomorphic images and either type of query, the problem lies in inverting the homomorphism on answers to queries in a way that is guaranteed to be consistent with some expression over the larger monoid.

There is, however, an important case where we can show that learnability is preserved by taking submonoids, that of *inducible* submonoids [GR02]. A submonoid N of a monoid M is inducible if there exist an integer k and an expression $E(x_1, \ldots, x_k)$ over M such that N is precisely the range of $E(M^k)$.

Proposition 2. *For the three combinations of query types (Evaluation queries only, Equivalence queries only, or both) the following is true. If expressions over the monoid M are polynomial-time learnable and N is an inducible submonoid of M, then expressions over N are polynomial-time learnable (nonproperly).*

Proof. Let $E(x_1, \ldots, x_k)$ be an expression inducing N. For each $a \in N$, fix elements $b_1(a), \ldots, b_k(a) \in M$ such that $E(b_1(a), \ldots, b_k(a)) = a$. Let $F(x_1, \ldots, x_n)$ be a target expression over N, and consider the expression $G(y_1^1, \ldots, y_n^k)$ obtained by replacing each occurrence of x_i in F with the expression $E(y_i^1, \ldots, y_i^k)$. Note that for any assignment $(a_1, \ldots, a_n) \in N^n$, we have $F(a_1, \ldots, a_n) = G(b_1(a_1), \ldots, b_k(a_n))$. Also, the output of G is always in N because the range of E is N and F does not contain any constant in $M \setminus N$.

To learn F, run the learning algorithm for M. We will simulate the target G given access to the target F.

- An Evaluation query $(b_1^1, \ldots, b_n^k)$ to G (a vector in M^{kn}) is translated to the Evaluation query $(E(b_1^1, \ldots, b_1^k), \ldots, E(b_n^1, \ldots, b_n^k))$ to F (a vector in N^n); the answers must be the same by definition of G.

- An Equivalence query with hypothesis H is interpreted as the query "is $F(a_1, \ldots, a_n)$ equal to $H(b_1(a_1), \cdots, b_k(a_n))$ for all $(a_1, \ldots, a_n) \in N^n$?". A Yes answer means that H can be used to evaluate F on all of N^n, hence we have learned E nonproperly. A counterexample $(a_1, \ldots, a_n, v)$ (a vector in $N^n \times N$) provides a counterexample $(b_1(a_1), \ldots, b_k(a_n), v)$

(a vector in $M^{kn} \times N$) on which which G and H differ, as required by the learning algorithm that is trying to learn G.

Since we are answering all queries consistently with G, the algorithm must in polynomial time either produce a hypothesis that is equivalent to G on all of M^n, or else stop because it receives a Yes answer to an Equivalence query, indicating equivalence over N^n. In either case, F is learned in polynomial time. $\square$

The last general result says that learning expressions over a monoid is never harder (up to constant factors) than learning the class of boolean functions computed by boolean programs over the monoid.

Theorem 3. *For the three combinations of query types (Evaluation queries only, Equivalence queries only, or both) the following is true. If boolean functions computed by programs over a monoid M are polynomial-time learnable, then expressions over M are polynomial-time learnable (nonproperly).*

Proof. Let k be $|M|$ and denote the elements of M by $a_1, \dots, a_k$. Let $E(x_1, \dots, x_n)$ be a target expression over M. We encode E into k boolean functions. For each variable x_i, introduce k boolean variables $y_{i1}, \dots, y_{ik}$. Intuitively, y_{ij} means "input x_i is assigned a_j in E". So for an assignment $(c_1, \dots, c_n) \in M^n$, let $code(c_1, \dots, c_n)$ be the assignment

$$\underbrace{0, \dots, 0}_{c_1-1}, 1, \underbrace{0, \dots, 0}_{k-c_1}, \dots, \underbrace{0, \dots, 0}_{c_n-1}, 1, \underbrace{0, \dots, 0}_{k-c_n}$$

in $\{0,1\}^{kn}$. A set of boolean functions $\{h_a \mid a \in M\}$ is said to be *collectively equivalent* to E if for every $(c_1, \dots, c_n)$ there is exactly one h_a which is 1 on $code(c_1, \dots, c_n)$, and is precisely the h_a such that $E(c_1, \dots, c_n) = a$. We can evaluate E on any assignment if we have access to such a set of functions as oracles. Thus, if we learn any set of functions collectively equivalent to E we have learned E nonproperly.

We define a specific set of functions $\{f_a \mid a \in M\}$ collectively equivalent to E and with the additional property that we can evaluate any f_a on *any* assignment from an Evaluation query to E. We use the following notation: for $a \in M$ and $b \in \{0,1\}$, $b \wedge a$ is a if $b = 1$ and e (the identity of M) if $b = 0$. For each a in M, let $f_a(y_{11}, \dots, y_{nk})$ be the boolean function "the value of $E(c_1, \dots, c_n)$ is a, where $c_i = (y_{i1} \wedge a_1) \cdot_M \cdots \cdot_M (y_{ik} \wedge a_k)$". The definition gives an immediate way of evaluating f_a on any assignment from an Evaluation query to E, and it is easy to check that $\{f_a \mid a \in M\}$ is collectively equivalent to E.

We learn E as follows. Let A be an algorithm that learns boolean functions computed by programs over M. We assume w.l.o.g. that A always asks at least one Equivalence query. Run a copy A_a of A for each a in M; we will make sure we answer the queries from A_a according to f_a.

When A_a asks a Membership query about f_a, translate it to an Evaluation query for E, using the definition of f_a.

When some A_a asks an Equivalence query, suspend it until all A_a are asking Equivalence queries. Collect the queries into a set $\{h_{a_1}, \ldots, h_{a_k}\}$ and ask this set as an Equivalence query, meaning "do the functions in this set collectively compute the same function as E?". If the answer is Yes, stop: we have learned E. If the answer is a counterexample $(w, E(w))$, then on $code(w)$, either $h_{E(w)}$ says 0 when it should say 1, or for some $a \neq E(w)$, h_a says 1 when it should say 0. In either case, we can pass $code(w)$ as a counterexample for h_a to one of the A_a algorithms. In time roughly k times the running time of A on f_a, we collect a set of h_a that receives a Yes answer.

Finally, observe that there is a program computing f_a with $k|E|$ instructions: for each occurrence of x_i in e, add to the program the list of k instructions $(y_{i1}, e, a_1) \ldots (y_{ik}, e, a_k)$. This list produces precisely $(y_{i1} \wedge a_1) \cdots (y_{ik} \wedge a_k)$, so the concatenation for all positions of E produces exactly $E(c_1, \ldots, c_n)$, where $c_i = \prod_j (y_{ij} \wedge a_j)$. Taking $\{a\}$ as accepting set we obtain a program that computes f_a.

Therefore, if A runs in time $f(n, m)$ (where m is the shortest program computing its target), we learn E in time $O(k \cdot f(n, k|E|))$. This is a polynomial in n and $|E|$ if f is a polynomial. $\square$

To conclude these general results, note that they remain true if we move to the PAC setting [Val84], where Equivalence queries are replaced with a source of random examples and the worst-case, exact learning condition is replaced with a probabilistic, approximate one.

4 Abelian Groups

Abelian or commutative groups are the simplest groups from an algebraic point of view. Quite naturally, they are easiest from the learning point of view, in the sense that they are learnable with a linear number of either Evaluation or Equivalence queries.

Theorem 4. *a) Expressions over an Abelian group G are properly learnable in polynomial time with $n+1$ Evaluation queries.*

b) Expressions over an Abelian group G are properly learnable in polynomial time with $O(n)$ Equivalence queries.

Proof. Both a) and b) rely on the fact that every expression over Abelian groups can be rewritten into a normal form. Let G be Abelian, and E be an expression over G using variables $x_1, \ldots, x_n$. Using commutativity, it is clear that E is equivalent to an expression of the form $c + \alpha_1 \cdot x_1 + \cdots + \alpha_n \cdot x_n$, where $c \in G$, $0 \leq \alpha_i < |G|$, and $\alpha_i \cdot x_i$ stands for $\underbrace{x_i + \cdots + x_i}_{\alpha_i}$. So learning E consists of finding c and the n coefficients α_i.

a) Learning with Evaluation queries: Ask $Eval(0^n)$, where 0 is the identity of G. The answer is the constant c.

To find the α_i, recall that every finite Abelian group G is of the form $Z_{q_1} \times \cdots \times Z_{q_k}$. Let $g \in G$ be the k-tuple $(g_1, \ldots, g_k)$, where g_j is a generator of Z_{q_j}. For each $i \in \{1 \ldots n\}$, ask $Eval(w_i)$, where $w_i \in G^n$ is the n-tuple that contains g in the ith position and 0 everywhere else. Let v_i be the answer. Then α_i is the smallest number such that $c + \alpha_i \cdot w_i = v_i$; finding it takes constant time for fixed G.

b) Learning with Equivalence queries: We first show how to learn expressions over Z_q, where q is a prime power. Then we show how to extend this to general Abelian groups.

Suppose that after m queries we have gathered counterexamples $(w^{(1)}, v^{(1)})$, $\ldots$, $(w^{(m)}, v^{(m)})$, where $w^{(i)} \in Z_q^n$ and $v^{(i)} \in Z_q$, meaning $E(w^{(i)}) = v^{(i)}$ for every $i = 1 \ldots m$. We argue that it is enough to show how to solve the case in which the constant c in expression E is 0. First, note that we can assume that the constant c in expression E is known; this is because we can run $|G|$ times the following algorithm, once for each value of c, until it succeeds. If c is known we can simulate that it is 0 as follows: add a c at the front of every Equivalence query, and replace every counterexample $(w, E(w))$ received with $(w, E(w) - c)$. This way, we simulate learning the expression $(-c)E$, in which the constant is indeed 0.

Let $w_j^{(i)}$ be the jth component of $w^{(i)}$. Find any set of coefficients β_1,

..., β_n satisfying the set of equations

$$\sum_{j=1}^{n} \beta_j \cdot w_j^{(i)} = v^{(i)} \pmod q, \qquad i = 1 \ldots m. \tag{1}$$

We discuss later how to find these in polynomial time. Build the hypothesis $H(x) = \sum_{j=1}^{n} \beta_j \cdot x_j$ and ask it for equivalence with E. We make the following claim:

Claim Let w belong to the subgroup of Z_q^n generated by $\{w^{(1)}, \ldots, w^{(m)}\}$ ("w is linearly dependent on $\{w^{(1)}, \ldots, w^{(m)}\}$"). Then $H(w) = E(w)$.

Proof (of Claim). Assume $w = \sum_{i=1} c_i\, w^{(i)}$. Then

$$\begin{aligned}
H(w) &= \sum_{j=1}^{n} \beta_j\, w_j = \sum_{j=1}^{n} \beta_j \left(\sum_{i=1}^{m} c_i\, w_j^{(i)} \right) = \sum_{i=1}^{m} c_i \left(\sum_{j=1}^{n} \beta_j\, w_j^{(i)} \right) \\
&= \sum_{i=1}^{m} c_i\, v^{(i)} = \sum_{i=1}^{m} c_i\, E(w^{(i)}) \\
&= \sum_{i=1}^{m} c_i \left(\sum_{j=1}^{n} \alpha_j\, w_j^{(i)} \right) = \sum_{j=1}^{n} \alpha_j \left(\sum_{i=1}^{m} c_i\, w_j^{(i)} \right) = \sum_{j=1}^{n} \alpha_j\, w_j = E(w).
\end{aligned}$$

■ (Claim)

Suppose that the teacher returns a counterexample (w, v), that is, $H(w) \neq E(w) = v$. By the claim, w is linearly independent of the previous counterexamples. Then the following claim bounds the number of Equivalence queries:

Claim The longest sequence of linearly independent elements has length $c \cdot n$ for some $c \leq \log q$.

Proof (of Claim). See for example Lemmas 4.1 and 4.2 in [BTW98]. In fact, it is possible to prove [Lan92] that the longest independent chain in $Z_{p^\alpha}^n$ (p prime) has length at most $\alpha \cdot n$. ■ (Claim)

It remains to show that we can, in polynomial time, find the β_js, i.e. solve the system of equations (1). This is shown, for example, in [GR02] but we sketch their argument here for completeness.

We can rewrite our system in matrix form as $W\vec{\beta} = \vec{v}$ where the entry W_{ij} is just $w_j^{(i)}$. When q is a prime, we can carry out standard Gaussian elimination. When q is composite Z_q is not a field, but one can still diagonalize W using polynomially many *elementary operations*, namely interchanging of two rows or two columns (corresponding to reordering equations or variables), and, for $i \neq j$ addition of a multiple of row i to row j or addition of a multiple of column i to column j (the latter corresponding to a change of variables $\beta_i = \beta_i' + a\beta_j$).

We can easily find a solution to the diagonalized system of equations and then obtain the β_i's by "undoing" the changes of variables.

Finally, we show how to learn expressions over $G = Z_{q_1} \times \cdots \times Z_{q_k}$, where $q_i = p_i^{\alpha_i}$ and the p_i are prime. Recall that all finite Abelian groups are of this form.

We use the same trick as in Proposition 1, part (1): Run k copies of the learning algorithm, where an Equivalence query is a k-tuple of hypotheses over $Z_{q_1}, \ldots, Z_{q_k}$. The number of queries is $\alpha_1 \cdot n + \cdots + \alpha_k \cdot n \leq n \cdot \log \prod_{i=1}^{k} p_i^{\alpha_i} = n \cdot \log |G| = O(n)$.

In fact, instead of solving k systems of equations for the β_js independently, one can make a single system of equations and find a solution modulo $\prod_{i=1}^{k} q_i$. This way, every time we produce a single expression over G, i.e., we are learning properly. $\square$

Note that the algorithms in this proof run in time polynomial in $|G|$ and n, even if G is given in the input by its multiplication table.

5 Nilpotent groups

Let a and b be elements of a group G and let $[a, b] = a^{-1}b^{-1}ab$ denote the commutator of a and b, so that $ab = ba[a, b]$. These are the commutators of weight 2. Commutators of weight 3 are $[a, [b, c]]$ and $[[a, b], c]$ and commutators of weight k are defined inductively in the obvious way. We say that G is nilpotent class-k iff all commutators of weight $k + 1$ are the identity, and observe that any commutator of any weight involving the identity is itself the identity.

It is clear that nilpotent class-1 groups are exactly the Abelian groups. And indeed several properties of nilpotent groups are natural generalizations of those for Abelian groups. For example, it can be shown that n-variable boolean functions computable by programs over nilpotent class-k groups can always be represented (in the sense of [BST90]) by polynomials of degree k (with coefficients in an appropriate ring). Expressions over nilpotent groups are learnable from Evaluation queries alone.

As in the Abelian case, the learning algorithm is based on the fact that expressions can be rewritten in a normal form, although the transformation is more involved in this case. To define the normal form, we need the following definition: a commutator tree over a set of variables V is either an element of V, or an expression of the form $[E_1, E_2]$ where each of E_1 and E_2 is either a group constant or a commutator tree over V. Note that we choose the definition carefully so that variables are commutator trees but group constants are not. Note also that, using $[x, y] = x^{|G|-1}y^{|G|-1}xy$, every commutator tree can be rewritten as an equivalent expression, possibly with an exponential blowup in length. In a nilpotent class-k group all commutator trees of weight more than k are trivial, i.e., they always evaluate to the identity.

Lemma 5. *Let G be a nilpotent class-k group. Any expression $E(x_1, \ldots, x_n)$ over G can be rewritten in the form $c \cdot E_1 \cdot E_2 \cdot \cdots \cdot E_p$ of length polynomial in n, where:*

- *c is an element of G,*
- *each E_i is of the form $C_{i,1}^{\alpha_{i,1}} \cdot \cdots \cdot C_{i,l_i}^{\alpha_{i,l_i}}$, where $C_{i,1}, \ldots, C_{i,l_i}$ are (in some predefined order) all nontrivial commutator trees using exactly all variables in the set S_i, and*
- *$S_1, \ldots, S_p$ are all nonempty subsets of $\{x_1, \ldots, x_n\}$ of cardinality at most k, in lexicographical order. In particular, if $i < j$, then there is some variable in $S_j - S_i$.*

Proof. Let E be an expression over G. The rewriting is as follows:

First, using that $ab = ba[a, b]$, move all occurrences of constants to the beginning of the expression. For example, applying this process to $E = x_1\, b\, x_2\, a\, x_1\, c\, x_3\, a$ gives

$$\begin{aligned}
E &= x_1\, b\, x_2\, a\, x_1\, c\, x_3\, a \\
&= x_1\, b\, x_2\, a\, x_1\, ca\, x_3\, [x_3, a] \\
&= x_1\, b\, x_2\, aca\, x_1\, [x_1, ca]\, x_3\, [x_3, a] \\
&= x_1\, baca\, x_2\, [x_2, aca]\, x_1\, [x_1, ca]\, x_3\, [x_3, a] \\
&= baca\, x_1\, [x_1, baca]\, x_2\, [x_2, aca]\, x_1\, [x_1, ca]\, x_3\, [x_3, a]
\end{aligned}$$

This leaves a single constant c in front of E, and introduces commutator trees along E containing constants and at most one variable.

Then, do the same for all occurrences of variable x_1, then for all commutators of weight at most k containing only variable x_1. This gives a expression of the form $cx_1^{\alpha_1}E'E"$, where E' is formed by commutator trees that contain variable x_1 and possibly constants, and $E"$ by commutator trees containing at least one variable other than x_1. In general, consider all sets $S_1, \ldots, S_p$ of at most k variables in lexicographical order. This means that all proper subsets of S_i appear in the ordering before S_i. For each set S_i, consider all commutator trees of weight at most k containing constants, at least one occurrence of each variable in S_i, and no variables outside S_i. Call C_i this set of commutator trees. Note that when $S_i = \{x_j\}$, then x_j is one of these commutator trees. Order C_i arbitrarily, then move each commutator in C_i in turn to the front of the expression. Moving a commutator to the front only creates commutators with larger weights, so once a commutator has been moved it never reappears in the remaining part of the expression.

This gives the expression E in a normal form. Note that p, the number of sets S_i, is $\sum_{i=0}^{k} \binom{n}{i} \leq n^k$ and that the number of commutators in C_i is at most $T_k \cdot (k + |G|)^k$, where T_k is the number of trees with k leaves. Therefore, the length of this expression is polynomial in n (for fixed k). $\square$

Theorem 6. *Expressions over a nilpotent class-k group G are properly learnable with $(|G|n)^k$ Evaluation queries and $n^k|G|^{O(|G|^k)}$ time.*

Proof. For convenience, in this proof we refer to commutator trees as simply commutators. Let C_i be the set of commutators using exactly the variables in S_i, as in the proof of Lemma 5. We describe an algorithm that finds a target E from Evaluation queries. Assume w.l.o.g. that E is in the normal form given by Lemma 5.

First, ask $Eval(e^n)$, where e is the identity. The answer is the constant c in front of E.

Now proceed inductively. Suppose that $E = AE_iB$, where A and B are expressions containing the commutators for sets preceding and following S_i, respectively, and E_i contains commutators in C_i. Assume inductively that we have an expression A' that is equivalent to A, in the sense that $A(w) = A'(w)$ for every $w \in G^n$. We will build an expression E_i' such that $A'E_i'$ is equivalent to expression AE_i in the same sense. Expression E_i' is simply the sequence of commutators in C_i, each with an appropriate exponent.

To do this, let $w \downarrow S_i$ be the vector obtained from $w \in G^n$ by setting to the identity all components corresponding to variables not in S_i. Observe that, for any $w \in G^n$, we have $E_i(w) = E_i(w \downarrow S_i)$, because the commutators in E_i do not depend on variables outside S_i. We also have $B(w \downarrow S_i) = e$, because all commutators in B read some variable outside S_i so they evaluate to e (even the commutators consisting of one variable).

By definition, for every u we have $E(u) = A(u)E_i(u)B(u)$, or $E_i(u) = A^{-1}(u)E(u)B^{-1}(u)$, and then

$$\begin{aligned} E_i(w) &= E_i(w \downarrow S_i) = A^{-1}(w \downarrow S_i)E(w \downarrow S_i)B^{-1}(w \downarrow S_i) \\ &= (A'(w \downarrow S_i))^{-1} \cdot E(w \downarrow S_i) \cdot e. \end{aligned}$$

This gives a way to evaluate E_i on an arbitrary input if we can evaluate A' and ask Evaluation queries on all vectors of the form $w \downarrow S_i$.

From this, an expression E_i' equivalent to E_i on all vectors $w \downarrow S_i$ can be found by exhaustive search. Since the order of commutators in E_i is known, it is enough to find a sequence of exponents for all commutators in C_i that provides the correct values on all $w \downarrow S_i$ points.

As for complexity: The set of queries of the algorithm is the set of all vectors in G^n containing at most k nonidentities. There are less than $|G|^k n^k$ of these. Concerning execution time, the only expensive step is finding the set of consistent exponents for a set S_i. There are $|C_i| \leq T_k \cdot (k + |G|)^k$ exponents to try, where T_k is the number of trees with k leaves; each exponent can take $|G|$ values, and there are n^k sets S_i to try. Thus the time cost is $O(n^k \cdot |G|^{O(|G|^k)})$. □

For Equivalence queries, an approach like the one in Theorem 4, part (b), seems more difficult since it would involve solving polynomial equations of constant degree over the ring Z_q, which is NP-complete even for $q = 2$ [Val79].

If we allow non-proper learning, that is, if the output is not required to be an expression over G, we can give an algorithm whose running time has a better dependence on k and $|G|$.

Theorem 7. *Expressions over a nilpotent class-k group G are nonproperly learnable with $(|G|n)^k$ Evaluation queries in time $(|G|n)^{O(k)}$.*

Proof. We have shown in the proof of Theorem 6 that the set $Q_E = \{\langle v, E(v)\rangle \mid v \text{ has weight } \leq k\}$ completely determines the value of the expression E over G. The learning algorithm simply builds Q_E from Evaluation queries and outputs it.

This representation has the property that it allows us to evaluate E in polynomial time. Indeed, assume that E is in the normal form $c \cdot E_1 \cdot \cdots \cdot E_p$ given by Lemma 5. We have shown in the proof of that lemma that for any i and w, we have $E_i(w) = (A(w \downarrow S_i))^{-1} \cdot E(w \downarrow S_i)$, where $A(u) = E_1(u) \cdot \cdots \cdot E_{i-1}(u)$. This gives a polynomial-time recursion for evaluating $E(w)$ on any $w \in G^n$. $\square$

Note that k always satisfies $k \leq \log |G|$. So the algorithm implicit in the previous theorem runs in time quasipolynomial in $|G|$, even when the multiplication table of $|G|$ is given as part of the input.

6 Solvable Groups

In this section we establish learnability results for restricted classes of solvable groups. For a group G and subsets $A, B \subseteq G$, we denote as $[A, B]$ the subgroup generated by the commutators $[a, b]$ with $a \in A$ and $b \in B$. The *derived series* of G is the chain of subgroups $G = G_0 \supseteq G_1 \supseteq \ldots$ where $G_{i+1} = [G_i, G_i]$ and a group is *solvable* (depth-k) if this chain ends at the trivial group (after k steps).

Although in this section we are aiming mostly at solvable depth-2 groups, we derive results for more general monoids. We need the following definition.

Definition 8. *Let M and H be monoids and α be a function $M \times H \mapsto H$. For convenience, we denote $\alpha(m, h)$ as simply h^m. We say that α* is an action *of M on H if for all $m, m_1, m_2 \in m$ and all $h, h_1, h_2 \in H$ it satisfies the following four conditions:*

1. $(h_1 \cdot_H h_2)^m = h_1^m \cdot_H h_2^m$
2. $h^{m_1 \cdot_M m_2} = (h^{m_1})^{m_2}$
3. $h^{e_M} = h$
4. $(e_H)^m = e_H$.

Given finite monoids M and H and an action α of M on H, we can form a *semidirect product of H by M*, denoted by $H \rtimes M$, which consists of pairs (h, m) with $h \in H$ and $m \in M$ and multiplication given by

$$(h_1, m_1) \cdot (h_2, m_2) = (h_1 \cdot_H h_2^{m_1}, m_1 \cdot_M m_2).$$

One can verify that this operation defines a monoid with identity (e_H, e_M). Any such monoid is called an *extension* of H by M. Observe that the direct product is the special case in which the action of M on H is trivial, i.e. $h^m = h$ for all m, h. When M and H are groups, $H \rtimes M$ is a group with $(h, m)^{-1} = ((h^{-1})^{m^{-1}}, m^{-1})$. In this case, it is easy to see that the set $\{(h, e_M) \mid h \in H\}$ forms a normal subgroup isomorphic to H, and that $(H \rtimes M)/H$ is isomorphic to M.

Our results in this section concern monoids which are semidirect products of Z_p by another monoid M, and especially the case in which M is an Abelian group. Note that in this latter case $Z_p \rtimes M$ is a solvable depth-2 group. For example the group S_3 of permutations on three points, is a semidirect product of Z_3 by Z_2.

The actions of a monoid M on Z_p have a fairly simple structure which we state next.

Theorem 9. *Let Z_p be a cyclic group and M any monoid, and denote the operation on Z_p additively. Any action of M on Z_p can be written as $h^m = c_m h$ for all $m \in M$, $h \in Z_p$ where the $0 \leq c_m < p$ are integers and $c_m h$ is shorthand for $\underbrace{h + h + \ldots + h}_{c_m \text{ times}}$.*

Proof. In the additive notation, let 1 be the generator of Z_p. For an action of M on Z_p and for each $m \in M$, let c_m be such that $0 \leq c_m < p$ and $1^m = c_m 1$. Any element h of Z_p can be written as $c_h 1$ for some integer $0 \leq c_h < p$. Using condition (1) in the definition of an action (Definition 8) we obtain

$$h^m = (\underbrace{1 + \cdots + 1}_{c_h \text{ times}})^m = \underbrace{1^m + \cdots + 1^m}_{c_h \text{ times}} = \underbrace{1 + \cdots + 1}_{c_m c_h \text{ times}} = c_m(c_h 1) = c_m h,$$

as required. $\square$

From now on, we view elements of a monoid N as above as pairs in $Z_p \times M$. For an element $x \in N$, we use notations $Z_p(x)$ and $M(x)$ to denote the first and second elements of the pair associated to x, that is, we identify x with the pair $(Z_p(x), M(x))$.

Theorem 9 can be interpreted as follows:

Corollary 10. *Let N be any extension of Z_p by M, for some action. Then there is a function $c : M \mapsto Z_p$ such that for all $x, y \in N$*

$$M(x \cdot_N y) = M(x) \cdot_M M(y)$$

and

$$Z_p(x \cdot_N y) = Z_p(x) +_{Z_p} c(M(x)) \cdot Z_p(y).$$

To learn expressions over these monoids, we use the fact that their Z_p components are computable by *multiplicity automata*. Multiplicity automata over rings (MA for short) are an important generalization of classical automata. They were first used in the context of learning by Bergadano and Varricchio [BV96], who gave a polynomial-time algorithm for learning MA over fields from Evaluation and Equivalence queries. Later, Bshouty, Tamon, and Wilson [BTW98] extended the algorithm to work over a large class of rings instead of fields, including all finite integer rings; note that MA are called *matrix functions* in [BTW98]), and that the hypothesis class of their algorithm is not, strictly speaking, MA but a closely related model that they call *decision programs.*

The algorithm for MA has been used to learn several other classes of functions [BBTV97, BBB+00]. In particular, [BBTV97] uses the MA learning algorithm to learn some classes of boolean circuits with modular gates and boolean permutation branching programs of width at most 4. These results are probably related to the connection between some solvable groups and MA that we find here.

We give here a working definition of MA; for more systematic presentations see [BBB+00]. An MA over an alphabet Σ and a ring K is a nondeterministic finite automaton where each transition triple (q, a, q'), with $a \in \Sigma$, is additionally labelled by an element in K. To each path in the automaton we associate the value in K given by the product of all the labels along the path. An MA A computes a function $\Sigma^\star \mapsto K$, which we also denote by A, in the following way: for each input $w \in \Sigma^\star$, $A(w)$ is the sum of the values of all nondeterministic paths from initial states to final states defined by input w on the MA A. A particular case of MA that we use here is when $\Sigma = K$ and the MA is evaluated on inputs of a fixed length n, so that A is a function $K^n \mapsto K$.

Theorem 11. *Let N be a monoid that is an extension of Z_p by M. Then there is a function $c : M \mapsto Z_p$ such that for every expression $E(x_1, \ldots, x_n)$ over N there is a multiplicity automaton A over Z_p of size $O(n|E|)$ such that for all $a_1, \ldots, a_n \in N$,*

$$Z_p(E(a_1, \ldots, a_n)) = A(Z_p(a_1), \ldots, Z_p(a_n), c(M(a_1)), \ldots, c(M(a_n))).$$

Furthermore, for all vectors $u_1, \ldots, u_n \in Z_p$ and $v_1, \ldots, v_n \in Z_p$ such that some v_i is not in the range of c,

$$A(u_1, \ldots, u_n, v_1, \ldots, v_n) = 0.$$

Proof. From Corollary 10 it is easy to verify inductively that, for any $s_1, \ldots, s_m$ in N, we have

$$Z_p(s_1 \cdot_N \cdots \cdot_N s_m) = \sum_{i=1}^{m} \left(\prod_{j=1}^{i-1} c(M(s_j)) \right) Z_p(s_i),$$

where both the sum and product are over Z_p.

Then the sum above is a polynomial $P(x_1, \ldots, x_n, y_1 \ldots, y_n)$ on the $2n$ variables $x_1, \ldots, x_n, y_1, \ldots, y_n$, where $x_i = Z_p(s_i)$ and $y_i = c(M(s_i))$. It is easy to transform a polynomial into an equivalent MA; this results in an MA satisfying the first condition of the theorem for the expression $E = s_1 \ldots s_m$.

To satisfy the second condition, we have to ensure that the MA vanishes on any input $(x_1, \ldots, x_n, y_1, \ldots, y_n)$ such that some y_i is not in the range of c. Define a function $R : Z_p \mapsto Z_p$ by $R(v) = 1$ if v is in the range of c and $R(v) = 0$ otherwise. Then the expression

$$Q(x_1, \ldots, x_n, y_1 \ldots, y_n) = \sum_{i=1}^{m} \left(\prod_{j=1}^{i-1} y_j \right) x_i \left(\prod_{j=1}^{n} R(y_j) \right)$$

equals $P(x_1, \ldots, x_n, y_1 \ldots, y_n)$ on all vectors such that all y_i are in the range of c, and is 0 everywhere else. Reordering each term in the sum so that all occurrences of a variable appear together, it is easy to see that Q can also be transformed into an equivalent MA of size $O(n|E|)$. This MA satisfies the conditions in the theorem.

Note: It is not necessarily true that Q is a small polynomial, depending on R. For the special case of S_3, the function $R(v)$ is simply v^2, so Q is still a polynomial with $|E|$ terms. □

We can now state the main result in this section.

Theorem 12. *Suppose that expressions over a monoid M are learnable in polynomial time using Evaluation and Equivalence queries. Let N be an extension of Z_p by M, for some p. Then expressions over N are also learnable in polynomial time using Evaluation and Equivalence queries.*

Proof. Fix a target expression $E = E(x_1, \ldots, x_n)$ over N. Consider first the M-component of E. Let E_M be an expression over M obtained from E by replacing each constant (a, m) occurring in E with m, and interpreting variables over M. By Corollary 10 and an easy induction we have that $M(E((a_1, m_1), \ldots, (a_n, m_n))) = E_M(m_1, \ldots, m_n)$.

Let A be the MA provided by Theorem 11, computing the Z_p component of E. We learn E if we learn both the M and Z_p components of E. To do this, run simultaneously the algorithm for learning expressions over M, with target E_M, and the algorithm for learning MA in [BTW98], with target A. Queries are answered as follows.

An Evaluation query $(m_1, \ldots, m_n)$ to E_M is answered by asking to E the Evaluation query $((a_1, m_1), \ldots, (a_n, m_n))$ (where the a_i are any values in Z_p) and returning the M-component of the answer. An Evaluation query $(u_1, \ldots, u_n, v_1, \ldots, v_n)$ to A has answer 0 if any of the v_i is not in the range of the function c provided by Corollary 10 for N. Otherwise, pick a preimage $w_i \in M$ by c for each v_i. Build the assignment $((u_1, w_1), \ldots, (u_n, w_n)) \in N^n$ and ask it as an Evaluation query to the teacher for E. If the answer is (u, w), return u to the algorithm. In both cases, the value returned to the algorithm is precisely $A(u_1, \ldots, u_n, v_1, \ldots, v_n)$, by the properties of A stated in Theorem 11.

When one of the two algorithms produces an Equivalence query, wait until the other one does too (as usual, we assume w.l.o.g. that both algorithms ask at least one Equivalence query). Let A' and E' be the queries, referring to A and E_M, respectively. Ask the pair (A', E') as an Equivalence query to the teacher for E, with the meaning "do both A' and E' predict correctly the Z_p and M components of E, respectively?". If the answer is Yes, then the target E has been learned as the pair (A', E'). Otherwise the answer provides a counterexample for one of A' and E', and at least one of the two algorithms can make progress.

Since the answers given to both algorithms are consistent with A and E_M, and both run in polynomial time, this strategy receives a Yes answer in polynomial time. $\square$

The following corollary on a particular type of solvable groups follows by induction.

Corollary 13. *Let $p_1, \ldots, p_k$ be any natural numbers. Define $G_1 = Z_{p_1}$ and, for $1 < i \leq k$, let G_i be a semidirect product of Z_{p_i} with G_{i-1}. Then expressions over G_k are learnable from Evaluation and Equivalence queries.*

Let us make some remarks about the groups in Corollary 13. Certainly they are all solvable and many of them (such as S_3) are not Abelian. It can be shown that in fact all of them are solvable depth-2 and belong to the class of supersolvable groups (A. Miasnikov, personal communication). The computational power of these groups has not been studied in the literature, e.g. no lower bound on the length of boolean programs over these groups has been shown. Given our learning result, they appear as a natural next step for proving lower bounds.

When N is a group of the form $Z_p \rtimes Z_q$, an interesting point is that the algorithm proving Theorem 12 runs in polynomial time even if the multiplication table of N is given as part of the input. It is known [BST90] that the boolean functions computed by polynomial-length programs over direct products of semidirect products of Z_p by Z_q are the boolean combinations of those computed by polynomial-size, depth-2, circuits consisting of a MOD_p gate connected to a number of MOD_q gates. Because of these facts, we view this result as a partial analog of the following result by Bergadano *et al.* [BBTV97]. They show, using also multiplicity automata, that depth-2 boolean circuits consisting of a MOD_p gate (p prime) connected to arbitrary modular gates are learnable from Membership and Equivalence queries. The fact that modular gates with unbounded moduli are allowed in their result is related to polynomiality in the group size in our result.

When p is a prime power, groups $Z_p \rtimes Z_q$ are special cases of semidirect products of p-groups by Abelian groups, which are the largest class of groups with known lower bounds. These lower bounds were proved in [BST90] by showing that all such groups have the following property.

Definition 14. *A solvable group G is* non-narrowing *if there is a polynomial $p(m)$ such that for every program $P : \{0,1\}^n \mapsto G$ of length m and every $a \in G$, if there is an assignment $w \in \{0,1\}^n$ such that $P(w) = a$ then there are at least $2^n/p(m)$ such assignments.*

In particular, for non-narrowing groups, it is impossible to compute the AND function via a boolean program of subexponential length, since such a program is forced to produce a particular group element on exactly one of 2^n boolean assignments.

The next two theorems show that from the non-narrowing property, or in general from lower bounds for the AND function, one can produce an identification strategy with few Evaluation queries. They were obtained through discussions with Cris Moore.

Theorem 15. *If G is non-narrowing, then expressions over G can be identified probabilistically from Evaluation queries in polynomial time.*

Proof. We prove this result for programs $\{0,1\}^n \mapsto G$. The result for expressions follows then from Theorem 3.

Take two boolean programs P_1 and P_2 of length m computing different functions $\{0,1\}^n \mapsto G$. This means that the program $P_1 \cdot P_2^{-1}$ is not the constant identity, where P_2^{-1} is the concatenation of $|G|-1$ copies of P_2. Thus, because G is non-narrowing, P_1 and P_2 differ on a randomly chosen assignment in $\{0,1\}^n$ with probability at least $1/p(|G|m)$. A standard probabilistic argument shows the following: There is a polynomial q such that if $q(n, m, \log(1/\delta))$ assignments $x_1, \ldots, x_q$ are drawn independently and uniformly at random from $\{0,1\}^n$, then with probability $1-\delta$ every two nonequivalent programs of size m over n variables give different values on at least one of the x_i.

The learning algorithm simply draws these many elements of $\{0,1\}^n$ uniformly at random, where m is the upper bound on the length of the target expression that it receives initially, and asks them as Evaluation queries. With probability at least $1-\delta$, all programs not equivalent to the target are proved nonequivalent by at least one answer. In other words, the target function has been identified. □

Theorem 16. *Let $s(m)$ be the largest arity of an AND function that can be computed by a boolean program of length m over a group G. Then expressions over G can be identified from $(|G|n)^{O(s(2|G|m))}$ Evaluation queries.*

Proof. We show how to identify boolean programs with $n^{O(s(2m))}$ Evaluation queries, and the result for expressions follows from the proof of Theorem 3. As usual, by the weight of a boolean assignment $w = (w_1, \ldots, w_n)$ we mean the number of nonzero w_i. Let P_1 and P_2 be two nonequivalent boolean programs of length m over G. It is enough to show that they differ on an input of weight at most $s(2m)$, as there are no more than $n^{O(s(2m))}$ such inputs. The program $P = P_1 P_2^{-1}$ does not produce e_G everywhere. Let w have minimal weight such that $P(w)$ does not produce e_G, and let k be the weight of w. Obtain from P a program Q by fixing to 0 all variables that are 0 in w and taking $\{P(w)\}$ as the accepting set. Then, Q computes the AND function of the remaining k inputs, so $k \leq s(|Q|) \leq s(|P|) \leq s(2m)$, and then P_1 and P_2 disagree on an assignment of weight at most $s(2m)$, as desired. □

This theorem gives identification strategies for non-narrowing groups and for nilpotent groups:

Corollary 17. *(1) Expressions over a non-narrowing group can be identified from $n^{O(\log m)}$ Evaluation queries.*
(2) Expressions over a nilpotent group can be identified from $n^{O(1)}$ Evaluation queries.

Part (1) follows by the fact that computing the AND of n variables requires programs of length $2^{\Omega(n)}$ over a non-narrowing group. Part (2) follows since it is known that boolean programs over a nilpotent group can only compute the AND function of a constant number of variables [BST90]. Part (2) provides a weaker version (identification only) of Theorem 6.

As already mentioned, it is conjectured in [BST90] that the exponential lower bound on the length of boolean programs computing the AND function holds for all solvable groups. A proof of this conjecture would prove that boolean programs (hence, expressions) over any solvable group are learnable from a quasipolynomial number of Evaluation queries alone. It is worth noting that a lower bound of $\Omega(n \log \log n)$ is known on the length of programs computing the AND function over *any* group [BS95].

Given the previous results, we conjecture that, for prime p, expressions over a group G which is an extension of Z_p by Z_q are efficiently learnable from Evaluation queries alone, say probabilistically in polynomial time or deterministically in quasipolynomial time. Note a tempting (but wrong) argument to prove this conjecture. By Theorem 12, expressions over G are learnable from Evaluation and Equivalence queries. But by the argument in Theorem 15, an Equivalence query can be answered correctly with high probability by asking a polynomial number of randomly chosen Evaluation queries, hence, Evaluation queries suffice for learning. The argument is wrong because the algorithm given by Theorem 12 uses Multiplicity Automata as Equivalence queries. These queries may, for all we know, compute functions not computable by any expression over G, and then the probabilistic argument in Theorem 15 does not apply. That is, it could well happen that an Equivalence query is nonequivalent to the target, yet sufficiently close to it that a polynomial number of random Evaluation queries will fail to provide a counterexample.

7 Hardness of Nonsolvable Groups

In this section we show that expressions over nonsolvable groups are not polynomial-time learnable unless NC^1 circuits are polynomial-time learnable too. Hence, under the cryptographic assumptions in [AK95], they are not learnable at all.

We prove this by reducing the problem to that of predicting boolean programs over simple nonabelian groups, which are polynomially equivalent to NC^1 circuits [Bar89]. Conceptually, there are three parts in the reduction: 1) learning expressions over some nonsolvable group implies learning expressions over some simple nonabelian group; this does not follow trivially from the fact that a nonsolvable group contains a simple nonabelian one, because an algorithm for learning a nonsolvable group might conceivably use queries outside the simple nonabelian one to learn it; 2) over a simple nonabelian group, programs and expressions compute the same functions; 3) over a simple nonabelian group, programs simulate boolean programs in a prediction-preserving sense.

For the rest of this section, by "learnable" we mean polynomial-time learnable from Evaluation and Equivalence queries, where the hypothesis class for Equivalence is any one that can be evaluated in polynomial time.

Lemma 19 is a special case of the more general Theorem IX.1.2 in [Str94]. We prove it here to have a self-contained proof.

Lemma 18. *Let G be a simple group and e its identity. Then, for every $a, b \in G$ with $a \neq e$ there is a 1-variable expression $E_{a,b}(x)$ such that $E_{a,b}(a) = b$ and $E_{a,b}(e) = e$.*

Proof. Assume otherwise. Then there is $a \neq e$ and b such that for every expression $E(x)$ such that $E(e) = e$, $E(a) \neq b$. Equivalently, for every tuple $(c_1, c_2, \ldots, c_k, c_{k+1})$ in G^{k+1} such that $c_1 c_2 \ldots c_k c_{k+1} = e$, we have $c_1 a c_2 a \ldots c_k a c_{k+1} \neq b$, i.e., $c_1 a c_2 a \ldots c_k a c_k^{-1} \ldots c_2^{-1} c_1^{-1} \neq b$. Let H be the set of all elements in G that can be written as a product $c_1 a c_2 a \ldots c_k a c_k^{-1} \ldots c_2^{-1} c_1^{-1}$ for some $k \geq 0$ and some tuple $c_1, \ldots, c_k$. Then H is a group (in fact, the subgroup generated by $\{cac^{-1} \mid c \in G\}$), it is normal in G, proper because $b \notin H$, and nontrivial because $a \in H$. Thus G is not simple, a contradiction. □

Lemma 19. *Let G be a simple nonabelian group. Then, for every function $f : G \mapsto G$ there is a 1-variable expression E such that $f(a) = E(a)$ for every $a \in G$.*

Proof. For each nonempty set $S \subseteq G$, every $b \in G$, and every $a \notin S$ we build inductively expressions $E_{a,b,S}(x)$ such that $E_{a,b,S}(a) = b$ and $E_{a,b,S}(c) = e$ for all $c \in S$. If $S = \{c\}$, then $E_{a,b,S}(x)$ is the expression $E_{ac^{-1},b}(xc^{-1})$ built in Lemma 18. Otherwise, $S = \{c\} \cup T$. As G is simple and nonabelian we have $G = [G, G]$, so every $b \in G$ is the product of a number of commutators. Suppose b is the commutator $[\alpha, \beta]$. Then $E_{a,b,S}(x)$ is the expression $[E_{a,\alpha,\{c\}}(x), E_{a,\beta,T}(x)]$. If b is the product of two or more commutators, concatenate the corresponding expressions. It is routine to check that these expressions behave as stated.

Now the expression in the statement of the lemma is the concatenation of all expressions $E_{a,f(a),G\setminus\{a\}}(x)$ for all $a \in G$. $\square$

Theorem 20. *Let G be a simple nonabelian group. Then, for every program P over G there is an expression E, of length linear in that of P, computing the same function from G^n to G.*

Proof. Replace each instruction (i, f) in P with the expression computing $f(x_i)$ given by Lemma 19. $\square$

Theorem 21. *The following are equivalent:*

(a) For any nonsolvable group G, expressions over G are not learnable.

(b) For any nonsolvable group G, programs over G are not learnable.

(c) For any simple nonabelian group G, programs over G are not learnable.

(d) For any simple nonabelian group G, expressions over G are not learnable.

Proof. (a) implies (b), because we are not limiting the hypothesis class used for Equivalence queries. (b) implies (c) as every simple nonabelian group is nonsolvable. (c) implies (d) by Theorem 20. We prove that (d) implies (a) by contradiction. Suppose G is nonsolvable yet expressions over G are learnable. We prove next that expressions over the commutator subgroup $[G, G]$ are also learnable. By induction, expressions over the simple nonabelian subgroup appearing in the normal series of G are also learnable, contradicting (d).

To prove that expressions over $H = [G, G]$ are learnable if G is, it is enough to show that H is an inducible subgroup of G and appeal to Proposition 2. Elements in H are those that can be obtained as a product (hence, finite product) of commutators. Let k be such that all elements of H can be written as a product of at most k commutators. Then the expression $E(x_1, y_1, \ldots, x_k, y_k) = [x_1, y_1] \cdot \cdots \cdot [x_k, y_k]$ has exactly range H when its inputs range over G^{2k}. Here $[x_i, y_i]$ stands for the expression $x_i y_i x_i^{|G|-1} y_i^{|G|-1}$. $\square$

Theorem 22. *Let G be a simple nonabelian group. If programs over G are learnable, then boolean programs over G are learnable, hence NC^1 circuits are learnable.*

Proof. Let P be a boolean program over G computing a function $\{0,1\}^n \mapsto \{0,1\}$. Let T be the set of elements of G that are interpreted as 1 when output by G. Fix a nonidentity element $a \in G$. Build a (nonboolean) program Q by replacing each instruction (i, f) in P with the instruction (i, g), where $g(a) = f(1)$ and $g(x) = f(0)$ for all $x \neq a$. Let $E(x)$ be the expression, given by Theorem 20, such that $E(x) = a$ for $x \in T$ and $E(x) = e$ for $x \in G \setminus T$. The program $E(Q(x_1, \ldots, x_n))$ computes a function $G \mapsto \{e, a\}$ identical to that computed by P using input a instead of 1 and any other element instead of 0. Its length is linear in that of P, and the theorem follows easily. $\square$

The following is immediate from the preceding two theorems.

Corollary 23. *Unless NC^1 circuits are learnable, expressions over nonsolvable groups are not learnable.*

The same negative result can be obtained for the more powerful model of PAC-prediction with Evaluation queries, in fact the one used to prove the results in [AK95].

8 Aperiodic Monoids

We now turn our attention to the case of aperiodic or "group-free" monoids. The subclass of aperiodic monoid that tuns out to be central in our investigation is the pseudo-variety called **DA**, introduced by [Sch76]. Several characterizations of **DA** are known; see [TT02]. For example, monoids in **DA** are those M with the property that for any two elements e and s, if $e = e^2$ and $MeM \subseteq MsM$ then $ese = e$. For our purposes, it is more convenient to use the following language-theoretic characterization.

Let u be an arbitrary string in $A^\star$. Write $\alpha(u)$ for the set of letters appearing in u. Assume $a \in \alpha(u)$. Then there is a unique triple (u_0, a, u_1)

such that $u = u_0 a u_1$ and $a \notin \alpha(u_0)$, which we call the a-left decomposition of u. Symmetrically, there is a unique triple (u_0, a, u_1) such that $u = u_0 a u_1$ and $a \notin \alpha(u_1)$, which we call the a-right decomposition of u.

We define the following equivalence relation on A^*, indexed by k, by induction on $k + |\alpha(u)|$.

First, we define that $u \equiv_0 v$ holds for all strings u and v. Second, if $k > 0$, then $u \equiv_k v$ if and only if

1. $\alpha(u) = \alpha(v)$, and
2. for every $a \in \alpha(u)$, if (u_0, a, u_1) and (v_0, a, v_1) are the a-left decompositions of u and v, then $u_0 \equiv_k v_0$ and $u_1 \equiv_{k-1} v_1$, and
3. for every $a \in \alpha(u)$, if (u_0, a, u_1) and (v_0, a, v_1) are the a-right decompositions of u and v, then $u_1 \equiv_k v_1$ and $u_0 \equiv_{k-1} v_0$.

(Note that this is well-defined, since in (2) we have $|\alpha(u_0)| < |\alpha(u)|$ and in (3) we have $|\alpha(u_1)| < |\alpha(u)|$.)

It is shown in [TW98] that each $\equiv_k$ is a congruence of finite index on A^*, and that $M = A^*/\gamma$ belongs to **DA** iff γ is refined by $\equiv_k$ for some k. Moreover, it can be shown that any $\equiv_k$-class can be described by a regular expression of the form $A_0^* a_1 A_1^* \cdots a_s A_s^*$, where the a_i's are letters, the A_i's are subsets of the alphabet, and the expression is unambiguous, i.e. whenever a word u can be written as $u = u_0 a_1 u_1 \cdots a_s u_s$, with $u_i \in A_i^*$, this can be done in a unique way. When u can be factorized in this way, we will refer to the occurrences of the a_i's defining the factorization as "bookmarks".

Next we show that expressions over monoids in **DA** can be identified from a polynomial number of Evaluation queries. It is not hard to show that learning boolean programs over any monoid which is not a group requires exponentially many Evaluation queries for identification, regardless of computation time (we prove this, essentially, in Theorem 36). Therefore, **DA** witnesses that, in some cases, identification can be exponentially more difficult for boolean programs than for expressions, and, therefore, that the converse of Theorem 3 does not hold.

Theorem 24. *Expressions over a monoid in* **DA** *are learnable from a polynomial number of Evaluation queries (and possibly exponential time).*

Proof. Suppose that the target is an expression $E(x_1, \cdots, x_n)$ over a monoid M in **DA**. Assume that M is a homomorphic image of $M^*/\equiv_k$ and that every

$\equiv_k$-class is described by a regular expression as above with at most s a_i's in the expression. Ask the teacher to evaluate $E(w)$, for every $w \in M^n$ that has at most $2s$ components different from the identity of M: clearly, there are polynomially many such queries. Construct an expression $F(x_1, \cdots, x_n)$ that is consistent with all answers. We claim that $E(z) = F(z)$ for all $z \in M^n$. Indeed, fix z and look at $E(z)$ and $F(z)$ as words over the alphabet M. Define w by replacing by the identity all letters in z that do not contribute a bookmark to $E(z)$ or to $F(z)$; thus w has at most $2s$ components different from the identity, and $E(z) = E(w)$ and $F(z) = F(w)$. But by consistency of F, $E(w) = F(w)$. Therefore, $E(z) = F(z)$. □

We can also show, by a somewhat indirect argument, that expressions over a monoid in **DA** are learnable from Equivalence queries.

Theorem 25. *Expressions over a monoid in* **DA** *are (nonproperly) learnable from Equivalence queries in polynomial time.*

Proof. It is shown in [GT03] that the set of boolean functions computed by boolean programs monoids in **DA** are exactly those computed by decision trees of constant rank. Decision trees of constant rank can be learned from Equivalence queries in polynomial time [Sim95, CB95]. By Theorem 3, expressions over monoids in **DA** are nonproperly learnable from Equivalence queries. □

It is known that there are exactly two minimal monoids outside of **DA**, named U and BA_2. Monoid U is the syntactic monoid of the language $(aa^\star b)^\star$, has 6 elements, and is known to be universal, in the sense that every boolean function is computed by a program over U; this is because programs over U can simulate DNF formulas with polynomial blowup [Thé89]. Monoid BA_2 is the syntactic monoid of $(ab)^\star$, has 6 elements too, and is provably not universal.

Multiplication tables for U and BA_2 are given in Figures 1 and 2. To get an intuition of their behavior, observe that in both cases $aba = a$, $bab = b$, and $b^2 = 0$. The only difference is that $a^2 = a$ in U while $a^2 = 0$ in BA_2.

We show that the problem of learning expressions over U from Evaluation queries is as hard as the problems of learning monotone DNF formulas from Membership queries, which requires exponentially many queries [BCG$^+$96]. Similarly, learning expressions over BA_2 is as hard as learning singleton sets by Membership queries, which is shown by a standard argument to require

	1	a	b	ab	ba	0
1	1	a	b	ab	ba	0
a	a	a	ab	ab	a	0
b	b	ba	0	b	0	0
ab	ab	a	0	ab	0	0
ba	ba	ba	b	b	ba	0
0	0	0	0	0	0	0

Figure 1: Multiplication table for monoid U.

	1	a	b	ab	ba	0
1	1	a	b	ab	ba	0
a	a	0	ab	0	a	0
b	b	ba	0	b	0	0
ab	ab	a	0	ab	0	0
ba	ba	0	b	0	ba	0
0	0	0	0	0	0	0

Figure 2: Multiplication table for monoid BA_2.

an exponential number of queries too. Since we will use this "singleton argument" several times with minor variations, let us sketch it once. Take a class of subsets of a domain D containing all singleton sets $\{w\}$ for $w \in D$, and any Membership query learning algorithm for the class. Suppose that the algorithm asks less than $|D|-1$ Membership queries on all targets. Then, there are two singleton sets $\{w_1\}$ and $\{w_2\}$ such that neither w_1 nor w_2 are asked as Membership queries when the answer No is repeatedly given to the algorithm. Yet these No answers are consistent with both $\{w_1\}$ and $\{w_2\}$ being the target. Therefore, at least one of these two sets, when chosen as the target, will not be identified correctly by the algorithm, regardless of computation time.

Theorem 26. *Expressions over the monoid U are not learnable from subexponentially many Evaluation queries, even using unbounded computation time. The same holds for the monoid BA_2.*

Proof. (Monoid U) We show that for every monotone DNF formula F there is an expression over U that is strongly equivalent to it, in the sense that it computes exactly the same boolean function modulo a renaming of constants 0 and 1 to elements of U. The result follows because identifying monotone DNF formulas requires exponentially many Membership queries [BCG+96].

A construction is given in [Thé89] that, for each monotone DNF formula F, produces an expression P of length polynomial in $|F|$ such that for all y over $\{1, a\}$, $P(y) = 0$ iff y encodes a true assignment of F, replacing 1 with 0 and a with 1; otherwise $P(y) = ab$. The problem now is that a learning strategy can also ask Evaluation queries with assignments containing the other monoid elements (b, ab, ba, and 0). We make sure that any such Evaluation query can be answered immediately as 0 without any teacher.

Consider the expression $E(x) = axbxab$. We have $E(1) = E(a) = ab$, and $E(b) = E(ab) = E(ba) = E(0) = 0$. So the new expression $Q(y) = P(y)E(x_1)E(x_2)\cdots E(x_n)$ maps any assignment not in $\{1, a\}^n$ to 0, and behaves as P on assignments over $\{1, a\}$.

(Monoid BA_2) We show that expressions over BA_2 of length $3n$ can encode all singletons in $\{a, b\}^n$, hence $2^n - 1$ Evaluation queries are required to learn them by the singleton argument. For each variable x_i, consider the two expressions $(ab)x_ib$ or $ax_i(ab)$. If x_i is 0, 1, ab, or ba then both expressions evaluate to 0. If x_i is a, the first one is ab and the second one is 0, and if x_i is b, then the first one is 0 and the second one is ab.

Now consider the set of 2^n expressions $T_1T_2....T_n$ where each subexpression T_i is chosen to be either abx_ib or ax_iab. By the argument above, any of these expressions evaluates to 0 on all assignments containing at least one 0, 1, ab, or ba, and also to 0 on all assignments in $\{a, b\}^n$ but one; namely, on the assignment that sets $x_i = a$ for all i for which we chose abx_ib and $x_i = b$ otherwise. On this unique assignment it evaluates to $(ab)^n = ab$. Thus, these 2^n expressions encode the 2^n singletons from $\{a, b\}^n$. $\square$

Theorem 24 says that an easy-to-find, polynomial-size set of Evaluation queries identifies uniquely any target over a monoid in **DA**. But the argument does not give any obvious way to use the answers to these queries to evaluate the target on other inputs. For two incomparable subclasses of **DA**, commutative aperiodic and idempotent R-trivial, we know how to reconstruct efficiently an expression for the target from these answers. Recall that a monoid is idempotent if $a = a^2$ for every element a, and observe that every idempotent monoid is trivially aperiodic.

Theorem 27. *Expressions over aperiodic commutative monoids are properly learnable in polynomial time from n Evaluation queries.*

Proof. We use the following property of aperiodic commutative monoids: Let $g_1, \ldots, g_s$ be generators of the monoid such that $g_i^{t_i} = g_i^{t_i+1}$ for all i; the t_i exist because of aperiodicity. By commutativity, each $x \in M$ can be represented as $g_1^{k_1} \cdots g_s^{k_s}$, for $0 \leq k_i \leq t_i$. This can be used to show that for any c, x, and y in M, if $cx = cy$ then either $x = y$ or $c^2x = cx$ and $c^2y = cy$.

Let M be an aperiodic commutative monoid satisfying the equation $m^t = m^{t+1}$ and let e denote its identity. Any expression over M can be assumed to have the form $E(x_1, \ldots, x_n) = cx_1^{\alpha_1} \ldots x_n^{\alpha_n}$, where $0 \leq \alpha_i \leq t$. Asking $Eval(e^n)$ will return the constant c. With $(t+1)|M|$ queries, find an element m which maximizes the cardinality of the set $\{cm^i : i \geq 0\}$; let this cardinality be $r+1$. For each i, ask $Eval(e^{i-1}me^{n-i})$; let the returned answer be cm^{β_i} for some $\beta_i \leq r$. By aperiodicity, if $\beta < r$, then $cm^\beta \neq cm^{\beta+k}$ for all $k > 0$. We thus have $cm^{\alpha_i} = cm^{\beta_i}$ for each i. This implies that for every element s, we also have $cs^{\alpha_i} = cs^{\beta_i}$ for each i. Indeed, if $\beta_i < r$ then $\alpha_i = \beta_i$, and if $\beta_i = r$ then the result follows because the choice of m forces $cs^r = cs^{r+1}$.

We now claim that the expression E is equivalent to $cx_1^{\beta_1} \ldots x_n^{\beta_n}$. Let $m_1, \ldots, m_n$ be an arbitrary assignment to the variables: as explained above, we have $cm_1^{\alpha_1} = cm_1^{\beta_1}$. Assume inductively that $cm_1^{\alpha_1} \ldots m_i^{\alpha_i} = cm_1^{\beta_1} \ldots m_i^{\beta_i}$. Since we have $cm_{i+1}^{\alpha_{i+1}} = cm_{i+1}^{\beta_{i+1}}$, by commutativity, the property stated at the beginning of the proof, and the inductive hypothesis, we can deduce that

$$\begin{aligned} cm_1^{\alpha_1} \ldots m_i^{\alpha_i} m_{i+1}^{\alpha_{i+1}} &= cm_1^{\alpha_1} \ldots m_i^{\alpha_i} \cdot cm_{i+1}^{\alpha_{i+1}} = cm_1^{\beta_1} \ldots m_i^{\beta_i} \cdot cm_{i+1}^{\beta_{i+1}} \\ &= cm_1^{\beta_1} \ldots m_i^{\beta_i} m_{i+1}^{\beta_{i+1}}. \end{aligned}$$

The case $i = n$ proves that our proposed expression is equivalent to the original one.

Note that the algorithm runs in polynomial time even when M is given in the input as its multiplication table. $\square$

For the next theorem, we consider a subclass of **DA** consisting of *idempotent R-trivial monoids*, also known as $\mathbf{R_1}$. A monoid M is idempotent R-trivial if it satisfies $x^2 = x$ and $xyx = xy$ for all $x, y \in M$ In particular, these identities imply that the product $m_1 \cdot \ldots \cdot m_t$ of t elements of M is equal to $m_{i_1} \ldots m_{i_s}$ with $s \leq |M|$ and where m_{i_j} is the first occurrence from left to right of this particular monoid element.

Theorem 28. *Expressions over idempotent R-trivial monoids are properly learnable in polynomial time from Evaluation queries.*

Proof. Let M be an idempotent R-trivial monoid, and E an expression over M. By the property of idempotent R-trivial monoids explained above, we can assume that any constant or variable appearing in E appears only once, i.e. $E = E_0 c_1 E_1 \dots c_k E_k$ where the c_i's are distinct constants (hence $k < |M|$) and the E_i's are constant-free subexpressions over disjoint sets of variables.

If the expression is constant-free, i.e. $E = E_0$ in the notation above, we can proceed as follows: evaluate $E(w_i)$ for $i = 1, \dots, n$, where $w_i = e^{i-1} m e^{n-i}$ for any $m \neq e$. This tells us which variables appear in the expression, namely x_i appears in E iff $E(w_i) = m$. If the monoid is commutative and idempotent, this information determines the expression. Otherwise, let x and y be two elements that do not commute, and $s = xy$ and $t = yx$. We have $s \neq t$, $st = xyyx = xyx = xy = s$ and $ts = yxxy = yxy = yx = t$. Fix these elements s and t.

For every pair of variables x_i, x_j both appearing in E, we evaluate $E(w_{ij})$, where w_{ij} has s in position i, t in position j, and the identity everywhere else. Depending on whether the returned value is s or t, we know in which order these two variables appear in the expression. Sorting the result of these comparisons will yield the target expression.

For the general case, we can assume the constants c_i are known, since we can run the following algorithm over all possible choices of $c_1, \dots, c_k$. There are $O(1)$ possibilities altogether. Once we have generated $O(1)$ competing hypotheses, we can use the set of Evaluation queries described in the proof of Theorem 24 to discard all those not equivalent to the target.

Suppose first that E_0 is empty, i.e. that the expression to be learned begins with a non-trivial constant c_1. The ideal $N = c_1 M$ forms a proper submonoid of M (with c_1 being the identity): moreover, for any assignment $(m_1, \dots, m_n)$ in M^n, the value in M of $E(m_1, \dots, m_n)$ is equal to the value $E(c_1 m_1, \dots, c_1 m_n)$ in the monoid N. We can thus argue that E can be learned, by induction on the cardinality of the underlying monoid.

If $c_1 s c_2 \dots c_k = s c_1 c_2 \dots c_k$ for every $s \in M$, we can assume that no variables appear in E_0, i.e. we are in the case just discussed. Otherwise, suppose there exists some s such that $c_1 s c_2 \dots c_k \neq t = s c_1 c_2 \dots c_k$. For each i, we ask the evaluation of $E(w_i)$ where w_i is the identity everywhere except that the i^{th} component is equal to s. We claim that $t \neq c_1 \dots c_j s c_{j+1} \dots c_k$ for any j: Otherwise we get $c_1 s c_2 \dots c_k = c_1 s c_1 \dots c_k = c_1 c_1 \dots c_j s c_{j+1} \dots c_k = c_1 \dots c_j s c_{j+1} \dots c_k = t$, contradicting the original hypothesis. These n evaluations can thus be used to discover which variables appear in E_0. The subexpression $c_1 E_1 \dots c_k E_k$ over the remaining variables can be seen as taking place over the proper submonoid $c_1 M$ and can thus be learned by recurrence on the cardinality of the monoid. It remains to order the variables appearing in E_0. If $stc_1 \dots c_k = tsc_1 \dots c_k$ for all $s, t \in M$, the order is irrelevant: otherwise the order can be learned by using the same strategy as in the constant-free case discussed before. $\square$

Recall that all of **DA** is nonproperly learnable from Equivalence queries (Theorem 25). Next we show that the class of idempotent commutative monoids is *properly* learnable from Equivalence queries. It is easy to show that this class is precisely the intersection of the two classes considered in the previous theorems, aperiodic commutative monoids and idempotent R-trivial monoids.

We use the following property of idempotent commutative monoids. A monoid M is idempotent commutative iff there exists a partial order $\leq$ on M such that every two elements x, y have the element $x \cdot y$ as a greatest common lower bound (in other words, M forms a semilattice).

Theorem 29. *Expressions over idempotent commutative monoids are properly learnable in polynomial time from n Equivalence queries.*

Proof. If M is idempotent and commutative, we can assume that any expression over M is of the form $E = cx_{i_1} \dots x_{i_r}$ with $c \in M$ and $1 \leq i_1 < \dots < i_r \leq n$. We can assume that the constant c is known since the learner can run this algorithm for each constant c until it succeeds. Thus, the learner only needs to identify the set S of variables occurring in E.

As a first step, the learner assumes that $S = \{x_1 \dots, x_n\}$ and therefore proposes the expression $cx_1 \dots x_n$. If this initial guess is incorrect, the teacher returns a counterexample $(m_1, \dots, m_n)$ where $a = cm_{i_1} \dots m_{i_r} \neq cm_1 \dots m_n = b$. Since every term in the left-hand side product also appears in the right-hand side product we have $a > b = cm_1 \dots m_n$ in the semilattice. Therefore, there is at least one m_i such that $m_i \geq a$ does not hold and we can conclude that $x_i \notin S$. Continuing in this way, the learner produces the target expression after at most n queries. $\square$

To conclude the section, we show that the idempotency condition is necessary in the previous theorem within the realm of aperiodic monoids, at least for proper learning.

Theorem 30. *Expressions over a non-idempotent aperiodic monoid cannot be properly learned with a subexponential number of proper Equivalence queries.*

Proof. If M is aperiodic but non-idempotent, it must contain an element m of threshold 2, i.e. $m \neq m^2 = m^3$.

We will show that a polynomial number of proper Equivalence queries are insufficient for the learner to distinguish between the following set S of $2^{n/2}$ expressions:

$$\prod_{i=1}^{n/2} x_{2i-b_i},$$

where the b_i's are bits. That is, we only consider expressions such that exactly one of x_{2i}, x_{2i-1} occurs in the expression, all variables appear in order and no variable is repeated. Observe that all such expressions depend on different sets of variables, hence compute different functions.

For an expression E over M, denote by $\tau(E)$ the set of variables occurring in E. Suppose first that the hypothesis h proposed by the learner is such that both x_{2i} and x_{2i-1} are in $\tau(E)$ for some i (resp. neither x_{2i} nor x_{2i-1} are in $\tau(E)$). Then let $w \in M^n$ be such that $w_j = m$ for $j = 2i$ or $2i - 1$ and $w_j = 1$ otherwise. For all E's in S, we have $E(w) = m$ whereas $h(w) = m^2$ (resp. $h(w) = 1$). Such equivalence queries are thus useless for the learner.

Let h now be an Equivalence query such that for all i, exactly one of x_{2i}, x_{2i-1} occurs in h. Let w be such that $w_i = 1$ if $x_i \in \tau(h)$ and $w_i = m$ otherwise. We have that $h(w) = 1$, but that $E(w) = m^2$ for any $E \in S$ such that $\tau(E)$ and $\tau(h)$ differ in at least two positions. There are only $n/2 + 1$ expressions E such that $\tau(E)$ and $\tau(h)$ do *not* differ in at least two positions. Therefore, after k such queries, at least $2^{n/2} - k \cdot (n/2+1)$ candidate expressions remain in S, so $\Omega(2^{n/2}/n)$ queries are needed for learning. $\square$

In fact, one can prove something slightly stronger using the same ideas: if M is a monoid which is not the union of its subgroups, then expressions over M can not be learned with a subexponential number of Equivalence queries. Indeed, for such a monoid, we can find $m \in M$ such that $m \neq m^2 = m^{q+2}$ for some q. It is not hard to adapt the proof above to this case; we omit the details since the focus of the paper is mostly on groups and aperiodics.

9 Learning Programs over Monoids

In this section we consider learning tasks where we assume that the target is representable as a program, but not necessarily as an expression over a monoid. Clearly, if programs over some monoid M are learnable, then expressions over M are also learnable (using the same hypothesis class in both cases). Here we investigate cases in which the converse is also true.

We can prove such a converse for all classes of groups for which we obtained any results in the previous sections. Curiously, the argument is particular (and seemingly different) for each of the classes of groups we consider. It would be interesting to find a general argument showing that if expressions over a group G are learnable, then so are programs over G, and in particular knowing whether this implication is true for the group classes whose learnability remains unknown, namely, solvable groups that are neither nilpotent neither extensions of some Z_m by some Z_n.

We prove a couple of general results first.

Proposition 31. *If expressions over any monoid M are learnable from Equivalence queries, then so are programs over M.*

Proof. Let $P = (i_1, f_1), \ldots, (i_m, f_m)$ be a program over M. For each f mapping M to M and each i, $1 \leq i \leq n$, define variable $x_{i,f}$. Let E be the expression $x_{i_1,f_1} \cdots x_{i_m,f_m}$. We show how to learn P by simulating the Equivalence learning algorithm learning target E.

When the algorithm produces hypothesis $H = x_{j_1,f_1} \ldots x_{j_l,f_l}$, build the program $Q = (j_1, f_1) \ldots (j_l, f_l)$ and ask it as an Equivalence query to the teacher for P (constants in H, if any, translate to constant functions f_i). If the answer is Yes, then Q is equivalent to P and we are done. Otherwise, the answer is a counterexample $(a_1, \ldots, a_n, v)$ such that $P(a_1, \ldots, a_n) = v \neq Q(a_1, \ldots, a_n)$. Build an assignment b to the $x_{i,f}$ variables by setting $b_{i,f} = f(a_i)$. It is direct to show that $E(b) = P(a_1, \ldots, a_n) = v \neq Q(a_1, \ldots, a_n) = H(b)$, so (b, v) is a valid counterexample for H that can be given to the Equivalence learner. $\square$

The simulation approach above fails, in general, for Evaluation queries. Indeed, from an arbitrary Evaluation query, i.e., an arbitrary assignment to the expression variables $x_{i,f}$, it may be impossible to obtain a single consistent value for the program variables x_i. In fact, by Theorem 36 and the comments after it, no such simulation is possible.

Proposition 32. *If programs over M are learnable (by any of Evaluation queries, Equivalence queries, or both), and N is a submonoid of M, then programs over N are learnable (with the same type(s) of queries).*

Proof. It is easy to see that every submonoid of M is inducible by programs: to induce N, use an instruction (i, f) where $f(m) = m$ for all $m \in N$ and $f(m)$ is the identity of M for all $m \in M \setminus N$. Continue the proof as in Proposition 2, part (1). $\square$

Theorem 33. *a) Programs over an Abelian group G are learnable with $|G|n + 1$ Evaluation queries.*

b) Programs over an Abelian group G are learnable with $O(n)$ Equivalence queries.

Proof. Part a): If G is Abelian, given any program we can reorder and merge instructions so that there is at most one instruction reading each variable. Furthermore, we can assume that $f(e) = e$ (where e is the identity) in all instructions (i, f) but the first one. This is because each such f can be multiplied by $f^{-1}(e)$, and the product of all $f(e)$ moved to a constant instruction to the first place. That is, w.l.o.g. we can assume that the program over G is of the form $(1, f_c)(1, f_1)(2, f_2)(3, f_3) \ldots (n, f_n)$, where f_c is the constant c function and $f_i(e) = e$ for all i. The only task is to find functions f_c, f_1, ... , f_n.

Query for Evaluation the all-identity vector; the answer is the constant c. Then, for each i and each $a \in G \setminus \{e\}$, query the vector having a in the ith position and e everywhere else. Let $b_{i,a}$ be the answer. Clearly, $f_i(a) = b_{i,a} - c$. So these $|G|n + 1$ queries completely determine f_c, f_1, ... , f_n.

Part b) follows from Proposition 31 and part b) of Theorem 4. $\square$

Theorem 34. *Programs over a nilpotent class-k group G are properly learnable with $(|G|n)^k$ Evaluation queries and $n^k|G|^{O(|G|^k)}$ time.*

Proof. Let P be a program over G. First, ensure that for all instructions (i, f) in P, either f is a constant function or $f(e) = e$. To do this, decompose any instruction (i, f) that does not satisfy this condition in two consecutive instructions. The first one is (i, f') where f' is the constant function $f(e)$, and the second one is $(i, f(e)^{-1} \cdot f)$. Second, copying the proof of Lemma 5, rewrite P in the form

$$(1, c)E_1 \ldots E_p,$$

where

- c is a constant function

- expression E_i is a product of nontrivial commutator trees (in some predefined order) over the set $\{(j, f) \mid j \in S_i,\ f : G \mapsto G,\ f(e) = e\}$,
- every commutator tree in E_i reads all variables in S_i, and
- $S_1, \ldots, S_p$ are all sets of at most k variables in lexicographical order.

Now we can replicate the rest of the proof of Theorem 6 too. A program Equivalent to P is built inductively. At step i, we observe that the behavior of subprogram E_i is completely determined by the value of E on vectors of weight at most k, and the value of the prefix of P preceding E_i. Since E_i computes a function of at most k variables, it is possible to reconstruct (in time dependent on G and k, but independent of n) a program equivalent to E_i. Details are left to the reader. □

The following is the analog of Theorem 12 for groups.

Theorem 35. *For every group G which is an extension of Z_p by Z_q, programs over G are learnable in polynomial time using Evaluation and Equivalence queries.*

Proof. Let P be a program over G. We learn independently the Z_q and Z_p components of P. By definition of extension, P behaves on the Z_q component as some program Q over Z_q. This program can be learned from Evaluation queries by Theorem 33, part (a).

Next we show how to learn the Z_p component of P. Let the instructions in P be $(j_1, f_1)(j_2, f_2) \ldots (j_m, f_m)$. Let s_i stand for the value of $f_i(x_{j_i})$, that is, the result of P is the product in G of $s_1, \ldots, s_m$. Note that $Z_q(s_i)$ depends on both $Z_p(x_i)$ and $Z_q(x_i)$, and likewise for $Z_p(s_i)$.

As shown in the proof of Theorem 11, the Z_p component of P on some input is

$$\sum_{i=1}^{m} \left(\prod_{j=1}^{i-1} f(Z_q(s_j)) \right) Z_p(s_i)$$

where both sum and product are modulo p. Replace each s_i with its definition $f_i(x_{j_i})$, then reorder the products so that all occurrences of each variable x_i appear together. Permuting the names of indices i and j for convenience, we obtain an expression of the form

$$\sum_{j=1}^{m} \prod_{i=1}^{n} g_{i,j}(Z_p(x_i), Z_q(x_i)), \tag{2}$$

where the $g_{i,j} : Z_p \times Z_q \mapsto Z_p$ are a set of unknown binary functions. This expression is a sum of products of binary functions. It is known only that all sums of products of *unary* functions can be computed by MA [BBB+00], so we are not done yet. In the following, and using the particular structure of the functions $g_{i,j}$, we show how to reencode the inputs so that this expression can be computed by an MA over Z_p.

Let u_i and v_i be variables that stand for $Z_p(x_i)$ and $Z_q(x_i)$, respectively. Then reencode v_i in base p: introduce variables $v_{i,1}, \ldots, v_{i,\ell}$, with $\ell = \lceil \log_p q \rceil$ to stand for this encoding. Note that only encodings of values $0, \ldots, q-1$ make sense; encodings $q, \ldots, 2^\ell - 1$ are illegal.

We now describe a small MA $M_{i,j}$ having inputs $v_{i,1}, \ldots, v_{i,\ell}, u_i$, that computes $g_{i,j}(Z_p(x_j), Z_q(x_j)) = g_{i,j}(u_i, v_i)$ whenever $v_{i,1}, \ldots, v_{i,\ell}$ encodes a value $v_i \in \{0, \ldots, q-1\}$, and is 0 otherwise. MA $M_{i,j}$ has an initial state s and a final state t. From s, it branches deterministically after reading each $v_{i,k}$, using edges with weight 1. Each tuple of values of $v_{i,1}, \ldots, v_{i,\ell}$ leads to a distinct state, so this part of $M_{i,j}$ is a p-ary tree of depth ℓ. From each of these p^ℓ states, there is a transition to final state t. The weight of this transition is $g_{i,j}(u_i, v_i)$ if the tuple $v_{i,1}, \ldots, v_{i,\ell}$ encodes a value $v_i \in \{0, \ldots, q-1\}$, and 0 otherwise. It is easy to check that $M_{i,j}$ computes as desired.

Let M be a nondeterministic automaton whose initial state branches on $j = 1, \ldots, m$. Its jth branch consists of the concatenation of automata $M_{i,j}$ for all $i = 1, \ldots, n$, following expression (2). In analogy with Theorem 11, M has the following properties:

- if, for every i, the tuple $v_{i,1}, \ldots, v_{i,\ell}$ encodes a value $v_i \in \{0, \ldots, q-1\}$, then $M(v_{1,1} \ldots v_{1,\ell}\, u_1 \ldots v_{n,1} \ldots v_{n,\ell}\, u_n)$ is $Z_p(P((u_1, v_1), \ldots, (u_n, v_n)))$;
- otherwise, $M(v_{1,1} \ldots v_{1,\ell}\, u_1 \ldots v_{n,1} \ldots v_{n,\ell}\, u_n)$ is 0.

The rest of the proof is as that of Theorem 12.

Note that the size of M is $O(m \cdot n \cdot |M_{i,j}|)$. Since $|M_{i,j}| \leq p^\ell = p^{\lceil \log_p q \rceil} \leq pq$, this is $O(m \cdot n \cdot |G|)$, and in fact the algorithm runs in polynomial time even if G is part of the input. Actually, $|M_{i,j}|$ is $O(q)$ rather than $O(pq)$ in its minimal form. $\square$

We dealt with programs over nonsolvable groups in Corollary 23. For programs over nongroups, we have a strong negative result.

Theorem 36. *If a monoid M is not a group, then programs over M cannot be learned from less than $|M|^n$ Evaluation queries.*

Proof. The argument is the same as for proving that boolean terms, or singleton sets, cannot be learned with few Evaluation queries. We use the fact that M is not a group iff it has at least one idempotent besides the identity e. Let $w \in M$ be such that $e \neq w = w^2$. For all $m \in M$, define the function $g_m : M \mapsto M$ by

- $g_m(m) = e$,
- $g_m(u) = w$ for all $u \in M - \{m\}$.

Consider the set of $|M|^n$ programs of the form $(1, f_1)(2, f_2) \cdots (n, f_n)$, where each f_i is some g_m function. Each such program has the property that it evaluates to e on the assignment that sets each x_i to the value m such that $f_i = g_m$, and it evaluates to w on all other assignments. The singleton argument shows that learning this family of programs plus the constant-w program requires $|M|^n$ Evaluation queries. $\square$

Theorem 36 implies a strong separation of the complexity of learning programs and expressions by Evaluation queries for some monoids. Consider for example the aperiodic monoid AND having two elements, the identity and a zero. By the theorem, learning programs over AND requires 2^n Evaluation queries, yet it is easy to learn expressions over AND using $n + 1$ Evaluation queries.

Acknowledgements

We are indebted to Cris Moore for ideas and discussions that lead to Theorems 15 and 16, and to Alexei Miasnikov for clarifying explanations on semidirect products. We thank José L. Balcázar and the anonymous referees for helpful comments.

References

[AK95] D. Angluin and M. Kharitonov. When won't membership queries help? *Journal of Computer and System Sciences*, 50(2):336–355, 1995.

[Ang87] D. Angluin. Learning regular sets from queries and counterexamples. *Information and Computation*, 75(2):87–106, 1987.

[Ang88] D. Angluin. Queries and concept learning. *Machine Learning*, 2(4):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(1):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(3):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.

[BCG+96] N.H. Bshouty, R. Cleve, R. Gavaldà, S. Kannan, and C. Tamon. Oracles and queries that are sufficient for exact learning. *Journal of Computer and System Sciences*, 52(3):421–433, 1996.

[BMT92] M. Beaudry, P. McKenzie, and D. Thérien. The membership problem in aperiodic transformation monoids. *Journal of the ACM*, 39(3):599–616, 1992.

[BS95] D.A. Mix Barrington and H. Straubing. Superlinear lower bounds for bounded-width branching programs. *Journal of Computer and System Sciences*, 50(3):374–381, 1995.

[BST90] D.A. Mix Barrington, H. Straubing, and D. Thérien. Non-uniform automata over groups. *Information and Computation*, 89(2):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(4):941–952, 1988.

[BTW98] N.H. Bshouty, C. Tamon, and D.K. Wilson. Learning matrix functions over rings. *Algorithmica*, 22(1/2):91–111, 1998.

[BV96] F. Bergadano and S. Varricchio. Learning behaviors of automata from multiplicity and equivalence queries. *SIAM Journal on Computing*, 25(6):1268–1280, 1996.

[CB95] J. Castro and J.L. Balcázar. Simple PAC learning of simple decision lists. In *Proc. 6th Intl. Workshop on Algorithmic Learning Theory (ALT'95)*, number 997 in Lecture Notes in Artificial Intelligence, pages 239–248. Springer-Verlag, 1995.

[Eil76] S. Eilenberg. *Automata, Languages, and Machines*, volume B. Academic Press, 1976.

[GR02] M. Goldmann and A. Russell. The complexity of solving equations over finite groups. *Information and Computation*, 178(1):253–262, 2002.

[GT03] R. Gavaldà and D. Thérien. Algebraic characterizations of small classes of boolean functions. In *Proc. 20th Annual Symposium on Theoretical Aspects of Computer Science (STACS'03)*, number 2607 in Lecture Notes in Computer Science, pages 331–342. Springer-Verlag, 2003.

[Lan92] S. Lang. *Algebra*. Addison-Wesley Pub. Co., 1992.

[RTT98] J.-F. Raymond, P. Tesson, and D. Thérien. An algebraic approach to communication complexity. In *Proc. ICALP'98, Springer-Verlag LNCS*, volume 1443, pages 29–40, 1998.

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

[Sim95] H.U. Simon. Learning decision lists and trees with equivalence-queries. In *Proc. 2nd European Conference on Computational Learning Theory*, number 904 in Lecture Notes in Artificial Intelligence, pages 322–336. Springer-Verlag, 1995.

[Str94] H. Straubing. *Finite Automata, Formal Logic and Circuit Complexity*. Boston: Birkhauser, 1994.

[Thé89] D. Thérien. Programs over aperiodic monoids. *Theoretical Computer Science*, 64(3):271–280, 29 1989.

[TT02] P. Tesson and D. Thérien. Diamonds are forever: the variety DA. In *Algorithms, Automata, and Languages*, pages 475–500. World Scientific, 2002.

[TW98] D. Thérien and T. Wilke. Over words, two variables are as powerful as one quantifier alternation: $\mathrm{FO}^2 = \Sigma_2 \cap \Pi_2$. In *Proc. 30th ACM Symposium on the Theory of Computing*, pages 41–47, 1998.

[Val79] L.G. Valiant. The complexity of enumeration and reliability problems. *SIAM Journal on Computing*, 8(3):410–421, 1979.

[Val84] L.G. Valiant. A theory of the learnable. *Communications of the ACM*, 27(11):1134–1142, 1984.