Learning Expressions and Programs over Monoids

Ricard Gavaldà*
Univ. Politècnica de Catalunya

Pascal Tesson †
McGill University

Denis Thérien†
McGill University

July 14th, 2001

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 learning from queries. For a group G, we prove that expressions over G are easily learnable if G is nilpotent and impossible to learn efficiently (under cryptographic assumptions) if G is nonsolvable. We present some partial results for solvable groups, and point out a connection between their efficient learnability and the

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

†School of Computer Science, McGill University. 3480 University, Montréal, Québec H3A 2A7 Canada. Email: {`ptesso,denis`}`@cs.mcgill.ca`. Supported by NSERC, FCAR, and the von Humboldt Foundation.

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 resources. 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]. To avoid replicating this study, 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, or Equivalence queries, or both.

Note that this problem is not immediately comparable to the problem of learning the class of boolean circuits corresponding to programs over monoid M. This is because, on the one hand, the problem might be harder because the function class is richer. On the other hand, the answers to the queries provide finer information than in the boolean setting, namely, elements of the monoid, and this could help in learning.

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 *wreath product* operation.

Along the paper, we often say "a monoid M is learnable" for the sake of brevity, meaning "expressions over monoid M are learnable", and similarly for a class of monoids. When we consider learning programs instead of expressions, we always say so explicitly.

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.
- Solvable groups formed as extensions of a group Z_p by a group Z_q (for any p and q) are polynomial-time learnable from Evaluation and Equivalence queries in the form of Multiplicity Automata and, for prime p, learnable in probabilistic polynomial time from Evaluation queries alone.
- A slightly different 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 no claims are made regarding computation time). We show that, in fact, these or similar results will hold for any group for which we can prove a certain type of lower bound on its computation power; in other words, such lower bounds on computation power provide upper bounds on learning complexity.

- 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]. For some algorithmic problems on monoids, it is known that feasibility depends essentially on membership to **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) iff the monoid belongs to **DA** [RTT98]. Our results are:

- Expressions over **DA** monoids are identifiable from polynomially many Evaluation queries (though possibly not 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 the same is true for any monoid outside of **DA**, because it is also known 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, the class of idempotent commutative monoids is learnable by Equivalence queries. 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.

We also study the task of learning functions represented as programs over monoids, 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 any larger one. 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, programs over solvable groups compute the functions in CC^0 (polynomial-size circuits made of mod_q gates for a fixed q), and nonsolvable groups (or monoids) compute all functions in NC^1 [Bar89, BT88]. Formally, we do not know how to translate neither positive nor negative results in our model to the boolean case. 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 be learnable in a strong sense, in fact a situation quite in analogy with the results in this paper. We believe they are interesting as they provide a new perspective on learning problems near the borderline of current knowledge.

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. 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 $(M, \cdot)$. The operation is denoted $+$ (additive notation) instead of $\cdot$ when the monoid is an Abelian group. When there are several monoid in play we often write $a \cdot_M b$ (resp. $a +_G b$) to indicate that a and b are multiplied by the operation of monoid M (resp. that of the Abelian group G).

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, in general, the identity of N does not have to be that of M (unlike in groups).

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 $\phi : M \to N$ 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 with respect to $\cdot$. 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 $m^t = m^{t+1}$.

2.2 Expressions and Programs over Monoids

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

An *expression* over monoid M and variables $x_1, \dots, x_n$ is a string over the alphabet $M \cup \{x_1, \dots, x_n\}$. 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 $(w_1, w_2, \dots, w_n)$ in M^n, replace with w_i each occurrence of each variable x_i in the expression, then multiply out the resulting string of monoid elements to obtain a single monoid element. 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.

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.

In the literature, programs have been used mostly to compute boolean functions [Bar89, BT88]. In these *boolean programs*, 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. We will be here 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 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* M^n to M fixed by a *teacher* in an adversary way. The function is assumed to be representable as an expression over M and variables $x_1, \ldots, x_n$. Not all variables are necessarily used. The learning algorithm is initially given n and some upper bound m on the length of a shortest expression for the target.[1] Monoid M is fixed and known

[1]Symbols n and m will always have this meaning in the paper, i.e., number of variables and an upper bound on the length of shortest expression for the target.

both to the teacher and the learning algorithm.

The learning algorithm must produce an expression (or some other representation of the target function, as discussed later) equivalent to the target one on all of M^n.

To achieve learning, teacher and learner exchange information on the target function following some protocol; some specific protocols will be defined later.

Resources used by the algorithm are measured as a function of n and m. The class of expressions over M is learnable in time $t(n, m)$ in a given learning protocol 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 is *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" may be different in each learning protocol, 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 monoid M is fixed and known to the learning algorithm, and thus we regard $|M|$ as a constant. We often present algorithmic schemes to learn whole classes of monoids 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. A stronger notion of "polynomial-time learnability" of a monoid class would ask for an algorithmic scheme depending only polynomially on $|M|$. In an even stricter sense, we could ask for an algorithm that receives the multiplication table of M as part of the input, with the promise that M belongs to the monoid class, and running in time polynomial in $|M|$, n, and m; this would be truly "uniformly" learning of class, in the sense that the algorithm does not rely on hardwired information for each monoid.

As for interaction between teacher and learner, we consider two standard query types in Angluin's model: Evaluation 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 a hypothesis h, representing a function $A \mapsto B$ in some way; 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 $f(a)$.

One more distinction is necessary for Equivalence queries. As mentioned, we assume that the target f is computed by an expression over M. If the hypothesis of an Equivalence-query algorithm are always issued in the form of an expression over M, the algorithm is called *proper*; in general, properness means that queries belong to the same syntactic class of functions being learned. Otherwise, hypotheses may belong to a different and possibly richer class, and the algorithm is called *nonproper*. Therefore, when stating a result involving Equivalence queries, we have to make clear the hypothesis class used. An important requirement on any such hypothesis class is that it can be evaluated in polynomial-time, i.e., there is an algorithm that evaluates a given hypothesis on a given input in polynomial time.

Before investigating the learnability of specific classes of monoids, let us consider a quite general question. There are several constructions for building monoids from other monoids. The most natural ones are direct product, submonoid, and homomorphic image. 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 models of query learning (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 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 (nonproperly) learnable from Equivalence queries.*

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

Proof. (Sketch) For item (1), it is easy to simulate Evaluation queries to M and N independently; to simulate Equivalence queries, run algorithms for M and N in parallel and pair the hypotheses issued by both to get a counterexample for at least one of the two. For item (2), run the learning algorithm for M but assume that the teacher has a target function f mapping N^n to N. The algorithm will issue queries h mapping M^n to M, but the teacher is concerned only on the value of h on inputs in N^n. Precisely, on any query h, the teacher must return a Yes answer whenever $h(x) = f(x)$ for all $x \in N^n$, and otherwise it must return a counterexample $(x, u) \in N^n \times N$ with $u = f(x) \neq h(x)$; in this case, while necessarily $u = f(x) \in N$, it may happen that $h(x) \in M \setminus N$, but that is no problem for the M-learning algorithm.

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 $N^n \mapsto N$, is not able to answer these. In fact, the query is ill-posed because the answer may be different for two expressions computing different functions on M^n but the same one on N^n. 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 [GR99]. A submonoid N of 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 models of query learning (Evaluation queries only, Equivalence queries only, or both) the following is true. If expressions over monoid M are learnable and N is an inducible submonoid of M, then expressions over N are 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. 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 $M - N$.

To learn F, run the learning algorithm for M. We will simulate target G given access to 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 in a non-proper sense. 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. ∎

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.

3 Abelian Groups

Abelian or commutative groups are the simplest 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 3 *a) Expressions over an Abelian group G are learnable with $n+1$ Evaluation queries.*

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

Proof. Both items a) and b) rely on the same fact: that expressions over Abelian groups can be transformed easily 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 $\alpha_0 + \alpha_1 \cdot x_1 + \cdots + \alpha_n \cdot x_n$, where $\alpha_0 \in G$, $0 \leq \alpha_i < |G|$, and $\alpha_i \cdot x_i$ stands for $\underbrace{x_i + \cdots + x_i}_{\alpha_i}$. So learning E consists of finding α_0 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 α_0.

To find the α_i, recall that every Abelian group G is of the form $Z_{q_1} \times \cdots \times Z_{q_k}$. Let $g \in G$ be the k-tuple containing k copies of 1. Then, for each i, ask $Eval(g_i)$, where $g_i \in G^n$ is the n-tuple that contains g in the ith position and 0 everywhere else. Let v_i be the answer. Then choose α_i smallest such that $\alpha_0 + \alpha_i \cdot g_i = v_i$; this 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 this is extended to general Abelian.

Suppose that after m queries we have gathered counterexamples $(w^{(1)}, v^{(1)})$, ... , $(w^{(m)}, v^{(m)})$, where $w^{(i)} \in Z_q^n$ and $v^{(i)} \in Z_q$, meaning $E(w^{(i)}) = v^{(i)}$ for every $i = 1 \ldots m$. W.l.o.g. we assume that the constant c in expression E is 0 (we can do this, for example, introducing a dummy term $c \cdot x_0$ and setting $x_0 = 1$ in all counterexamples).

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 [GR99] 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, 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, in this case it is not difficult to produce every time a single expression over G, i.e., to make the algorithm proper: instead of solving k systems of equations for the β_js independently, make a single system of equations and find a solution modulo $\prod_{i=1}^{k} q_i$. ∎

Note that the algorithms in this proof run in polynomial time in $|G|$ in addition to n, and that in fact they run in polynomial time even if the multiplication table of G is a part of the input.

4 Nilpotent groups

Let a and b be elements of a group G and let $[a, b] = aba^{-1}b^{-1}$ denote the commutator of a and 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 of class-k iff all commutators of weight $k + 1$ are the identity, and observe that any commutator of any weight involving the identity it itself the identity.

It is clear that nilpotent groups of class 1 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 functions that are realizable by programs over nilpotent groups of class k 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 to 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 are careful in this definition so that group constants are not commutator trees, but variables in V are. Note also that, using $[x, y] = xyx^{|G|-1}y^{|G|-1}$, every commutator tree can be rewritten as an equivalent expression, possibly with an exponential blowup. 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 4 *Let G be a nilpotent class-k group. Then 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 $\sum_{i=0}^{k}\binom{n}{k} \le n^k$ 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). ∎

Theorem 5 *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 4. 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 4.

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 no more than the sequence of commutators in C_i with an appropriate exponent each.

To do this, let $w \downarrow S_i$ be the vector obtained from $w \in G^n$ by changing 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, and that $B(w \downarrow S_i) = e$, because all commutators in B read some variable outside S_i so they evaluate to e.

By definition, we have $E(u) = A(u)E_i(u)B(u)$ for any u, or $E_i(u) = A^{-1}(u)E(u)B^{-1}(u)$. So all in all we have $E_i(w) = E_i(w \downarrow S_i) = A^{-1}(w \downarrow S_i)E(w \downarrow S_i)B^{-1}(w \downarrow S_i) = (A'(w \downarrow S_i))^{-1} \cdot E(w \downarrow S_i) \cdot e$. 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 that in Theorem 3, part (b), seems difficult since it would involve solving polynomial equations over cyclic groups.

The algorithm in Theorem 5 seems to have a running time that is doubly exponential in k, the nilpotency class of G. It is a moderately interesting problem to reduce this dependence to a single exponential. We observe next that this reduction is possible if the algorithm is not required to output its result as an expression.

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

Proof. We have shown in the proof of Theorem 5 that the set $Q_E = \{\langle v, E(v)\rangle \mid v \text{ has weight } \leq k\}$ completely determines the value of an 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, E be in the normal form $c \cdot E_1 \cdot \dots \cdot E_p$ given by Lemma 4. 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 $w \in G^n$. ∎

5 Solvable Groups

In this section we will present some partial results on learnability in solvable groups. For any two subsets A and B of a group G, denote by $[A, B]$ the subgroup generated by all commutators $[a, b]$, with $a \in A, b \in B$. We can then form the descending chain of subgroups $G_0, G_1, \ldots$ by setting $G_0 = G$ and $G_i = [G_{i-1}, G_{i-1}]$. The group G is solvable iff this chain goes to the trivial subgroup.

In order to present our results concerning solvable groups, we have to recall the following notion. Let G be a group, and suppose H is a normal subgroup of G. Then G is said to be an extension of H by the quotient group G/H and it admits the following representation.

Let $R = \{r_1, \ldots, r_k\}$ be representatives for the k cosets of G (=elements of G/H). Then each element x of G can be written uniquely as a product $h \cdot r$ for $h \in H$ and $r \in R$, namely, r is the representative of Hx and $h = xr^{-1}$ is uniquely given by the fact that $Hr = Hx$. Thus, there is a bijection between G and $H \times (G/H)$. Observe that

$$(h_1 r_1) \cdot_G (h_2 r_2) = (h_1(r_1 h_2 r_1^{-1}))(r_1 r_2) = (h_1(r_1 h_2 r_1^{-1} u_{1,2}))r_3$$

where $r_3 \in R$ is the representative of the coset of $r_1 r_2$ and $u_{1,2}$ is such that $r_1 r_2 = u_{1,2} r_3$. In other words, product in G is isomorphic to the following product on pairs in $H \times (G/H)$:

$$(h_1, A) \cdot (h_2, B) = (h_1 \cdot_H f_A(h_2, B), A \cdot_{G/H} B).$$

We will call functions $f_A : H \times (G/H) \mapsto H$ "twists"; a twist couples the first components of the group to the second components. Observe that the direct product of two groups corresponds precisely to the case when $f_A(h, B)$ is simply h.

Our results in this section concern solvable groups which are extensions of Z_p by Z_q, where Z_p and Z_q are any cyclic groups. Note that, for example, the group S_3 of permutations on three points is an extension of Z_3 by Z_2.

The first result is based on the observation that twist functions for these groups have a particularly simple structure.

Theorem 7 *Let G be an extension of Z_p by Z_q. Then the twist functions necessary to build G have the form:*

$$f_A(x, B) = c_A \cdot_{Z_p} x,$$

where $c_A \in Z_p$ is a constant depending on A and $c_A \cdot_{Z_p} x = \underbrace{x + \cdots + x}_{c_A}$.

Proof. Using the notation above, let H be Z_p and G/H be isomorphic to Z_q. Let $[0]$, $[1]$, $\ldots$, $[q-1]$ be the q elements of G/H and choose representatives in the following particular way: the identity of G is the representative of $[0]$. The representative r_1 of $[1]$ is any element in $[1]$. The representative of $[i]$ is $r_i = r_1^i$. This way, the product of representatives r_i and r_j is directly the representative r_{i+j} (i.e., the element $u_{1,2}$ in the description above is the identity).

Then, and moving to additive notation,

$$(h_1 r_1) \cdot_G (h_2 r_2) = (h_1 +_{Z_p} (r_1 h_2 r_1^{-1})) \cdot_G (r_1 +_{Z_q} r_2)$$

so that

$$f_{[i]}(h, B) = r_i \cdot_G h \cdot_G r_i^{-1} = r_1^i \cdot_G h \cdot_G r_1^{-i}.$$

Now, observe that

$$f_{[i]}(0, B) = r_1^i \cdot_G 0 \cdot_G r_1^{-i} = 0$$

since $0 \in H$ is the identity of G, and

$$f_{[i]}(j+1, B) = r_1^i \cdot_G (j+1) \cdot_G r_1^{-i} = (r_1^i \cdot_G j \cdot_G r_1^{-i}) + (r_1^i \cdot_G 1 \cdot_G r_1^{-i})$$

Therefore, $f_{[i]}(j, B) = (r_1^i \cdot_G 1 \cdot_G r_1^{-i}) \cdot_{Z_p} j$ for all $i \in Z_q$, and $j \in Z_p$. Defining $c_{[i]}$ as $(r_1^i \cdot_G 1 \cdot_G r_1^{-i})$, the theorem is proved. ∎

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

Theorem 7 can be interpreted as follows:

Corollary 8 *For any G which is an extension of Z_p by Z_q, there is a function $f : Z_q \mapsto Z_p$ such that for all $x, y \in G$,*

$$Z_q(x \cdot_G y) = Z_q(x) + Z_q(y)$$

and

$$Z_p(x \cdot_G y) = Z_p(x) + f(Z_q(x)) \cdot Z_p(y).$$

The learning algorithm for these solvable groups uses Multiplicity Automata as hypothesis class. 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 by Equivalence and Evaluation (there called Multiplicity) 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 Multiplicity Automata; for more systematic presentations see [BV96, BTW98]. An MA over an alphabet Σ and a ring K is a nondeterministic finite automata where each transition triple (q, a, q') $(a \in \Sigma)$ is additionally labelled by an element in K. To each path in the automata we associate the value in K given by the product of all the labels along the path. The MA computes a function $M : \Sigma^\star \mapsto K$ in the following way: for each input $w \in \Sigma^\star$, $M(w)$ is the sum of the values of all nondeterministic paths from initial states to final states defined by input w on the MA. A 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 $M : K^n \mapsto K$.

Theorem 9 *Let G be a group which is an extension of Z_p by Z_q. Then there is a function $f : Z_q \mapsto Z_p$ such that for every expression $E(x_1, \ldots, x_n)$ over G there is a multiplicity automata M over Z_p of size $O(n|E|)$ such that for all $a_1, \ldots, a_n \in G$,*

$$Z_p(E(a_1, \ldots, a_n)) = M(Z_p(a_1), \ldots, Z_p(a_n), f(Z_q(a_1)), \ldots, f(Z_q(a_n))).$$

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 f,

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

Proof. From Corollary 8 it is easy to verify inductively that, for any $s_1, \dots, s_m \in G$, we have

$$Z_q(s_1 \cdot_G \cdots \cdot_G s_m) = \sum_{i=1}^{m} Z_q(s_i),$$

where sum is over Z_q, and

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

where both sum and product are over Z_p.

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

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

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

equals $P(x_1, \dots, x_n, y_1 \dots, y_n)$ on all vectors such that all y_i are in the range of f, 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 the complexity of function R. For the special case of S_3, the function R can be implemented as a monomial so that Q is still a polynomial with $|E|$ terms. ∎

Then we can prove:

Theorem 10 *For every group G as above, expressions over G are learnable in polynomial time using Evaluation and Equivalence queries. The Equivalence queries and the output of the algorithm are pairs formed by a decision program over the ring Z_p and an expression over Z_q.*

Proof. Fix a target expression $E = E(x_1, \ldots, x_n)$ over G. Corollary 8 provides two expressions computing separately the Z_p and Z_q components of E on any assignment.

The expression for the Z_q component is a standard expression over Z_q on variables $Z_q(x_1)$, ..., $Z_q(x_n)$. Thus, it can be learned from Evaluation queries (setting all Z_p components to 0) by Theorem 3. Let E' be the result.

The Z_p component of E is learned in the form of the MA M given by Theorem 9. Use the algorithm described in [BTW98] and answer queries as follows.

When the algorithm asks a point $(u_1, \ldots, u_n, v_1, \ldots, v_n)$ as Evaluation query, check first that each v_i is in the range of function f. If this is not the case, return answer 0 to the algorithm. Otherwise, pick up a preimage $w_i \in Z_q$ by f for each v_i. Build the assignment $((u_1, w_1), \ldots, (u_n, w_n)) \in G^n$ 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 $M(u_1, \ldots, u_n, v_1, \ldots, v_n)$, by the properties of M stated in Theorem 9.

When the algorithm produces an Equivalence query N, ask the pair (N, E') for equivalence to the teacher for E. The query is intended to mean whether both N and E' predict correctly components Z_p and Z_q of E. If the answer is YES, then the target E has been identified. Otherwise the answer is a counterexample for the pair (N, E'). Since E' is correct on the Z_q components, the counterexample provides a counterexample for MA N that can be passed to the MA algorithm.

As all answers passed to the learning algorithm are consistent with MA M, in polynomial time the algorithm produces an Equivalence query that is answered YES. ∎

An interesting point is that the algorithm proving Theorem 10 still runs in polynomial time if the multiplication table of G is given as part of the input. Because of this fact, we view this result as a partial analog of the following result by Bergadano *et al.*[BBTV97]. Using also multiplicity automata, they show that depth-two circuits consisting of a MOD$_p$ gate (p prime) connected to arbitrary modular gates are learnable from Membership and Evaluation queries. The fact that modular gates with unbounded moduli are allowed in their result seems related to polynomiality in the group size in our result.

When p is prime, the groups we have been working with in this section can be viewed as special cases of semi-direct products of Abelian groups by p-groups. This larger family has been studied in [BST90] where it was shown that any group of that form could not possibly compute the AND function via a program of subexponential length. Moreover, these groups were also shown to have the following property.

Say that a solvable group G is non-narrowing if there is a polynomial $p(m)$ such that for all program P and all $a \in G$, if there is an assignment $w \in G^n$ such that $P(w) = a$ then there are at least $|G|^n / p(length(P))$ such assignments. This property gives an identification strategy by polynomially many Evaluation queries (not necessarily a polynomial-time algorithm as we do not know how to efficiently obtain a hypothesis from the answers).

The remaining results in this section were obtained through discussions with Cris Moore.

Theorem 11 *If G is non-narrowing, then programs (hence, expressions) over G can be identified probabilistically from Evaluation queries in polynomial time.*

Proof. Take two programs P_1 and P_2 of size m computing different functions $G^n \mapsto G$. This means that the program $P_1 \cdot P_2^{-1}$ is not the constant identity. Thus, because G is non-narrowing, P_1 and P_2 differ on a randomly chosen assignment in G^n with probability at least $1/p(2m)$. By a standard counting argument, there is a polynomial $q(n, m, 1/\log(\delta))$ such that if S is chosen uniformly at random from G^n and has cardinality $q(n, m, 1/\log(\delta))$ then every two nonequivalent programs of size m over n variables give different values on at least one element of S, with probability at least $1 - \delta$.

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

With the same argument, it is easy to show that an Equivalence query to the groups above can be simulated with high probability by polynomially many random Evaluation queries. We can combine this observation with the Equivalence and Evaluation query algorithm in Theorem 10.

Corollary 12 *Any solvable group which is an extension of Z_p (p prime) by Z_q is learnable from Evaluation queries in probabilistic polynomial time.*

We finally observe that an exponential lower bound on the length of programs over G that compute the AND function translates into a quasipolynomial upper bound on the number of Evaluation queries needed to identify a program over that group.

Theorem 13 *If programs over G cannot compute the AND function in subexponential length, then programs over G can be identified from $n^{O(\log m)}$ Evaluation queries.*

Proof. Let P_1 and P_2 be two length m programs over G that compute distinct functions. The theorem will follow if we show that they must differ on an input of weight at most $O(\log m)$, as there are $n^{O(\log m)}$ such inputs. The program $Q = P_1 P_2^{-1}$ is not everywhere the identity: let w have weight k and be of minimal weight such that $Q(w)$ is not the identity. WLOG assume that $w = 1^k 0^{n-k}$. Obtain from Q a program Q' by fixing all bits in positions greater than k to 0. This new program computes the AND function of k bits and has thus length $2^{\Omega(k)}$. ∎

Note that it is conjectured in [BST90] that the exponential lower bound on the AND function holds for all solvable groups. A proof of this conjecture would prove that programs over any solvable group are learnable in quasipolynomial time from Evaluation queries alone.

6 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 the learning algorithm for nonsolvable 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 the section, by "learnable" we mean polynomial-time learnable from Evaluation and Equivalence queries, where the hypothesis class for Equivalence is any polynomial-time evaluatable one.

Lemma 15 is a special case of more general Theorem IX.1.2 in [Str94]. We prove it here to have a self-contained proof.

Lemma 14 *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 $E(a) = b$ and $E(e) = e$.*

Proof. Deny. 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 $(c_1, c_2, \ldots, c_k, c_{k+1})$ of constants in G 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 expressable as a product of the form $c_1 a c_2 a \ldots c_k a c_k^{-1} \ldots c_2^{-1} c_1^{-1}$ 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 15 *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 14. Otherwise, $S = \{c\} \cup T$. 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 several 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 $a \in G$. ∎

Theorem 16 *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 15. ∎

Theorem 17 *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 class of hypothesis used for Equivalence queries. (b) implies (c) as every simple nonabelian group is nonsolvable. (c) implies (d) by Theorem 16. 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 $x_i y_i x_i^{|G|-1} y_i^{|G|-1}$. ∎

Theorem 18 *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 16, 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 size is linear in that of P. From here the learning result follows easily. ∎

The following is immediate from the preceding two theorems.

Corollary 19 *Unless NC^1 circuits are learnable, expressions over nonsolvable groups are not learnable.*

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

7 Aperiodic Monoids

We now turn our attention to the case of aperiodic or "group-free" monoids.

Theorem 20 *Expressions over a monoid in* **DA** *are learnable from a polynomial number of Evaluation queries and unbounded computation time.*

Proof. Let u be an arbitrary string. 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 $|\alpha(u_1)| < |\alpha(u)|$.)

In [TW98], it is shown 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 so factorized, we will refer to the occurrences of the a_i's defining the factorization as "bookmarks".

Suppose now that we have an expression $E(x_1, \cdots, x_n)$ over the monoid M that belongs to **DA**. Assume that M is a homomorphic image of $M^*/\equiv_k$ and that any $\equiv_k$-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 element of M: clearly, there are polynomially many such queries. Construct an expression $F(x_1, \cdots, x_n)$ which is consistent with all answers that are returned. We claim that $E(w) = F(w)$ for all $w \in M^n$. Suppose this is not the case for some w. Look at $E(w)$ and $F(w)$ as words over the alphabet M. Define z by replacing by the identity all letters in w that do not contribute a bookmark to $E(w)$ or to $F(w)$: thus z has at most $2s$ components different from the identity. Because of consistency, $E(z) = F(z)$: but since $E(z) = E(w)$ and $F(z) = F(w)$, we get a contradiction. ∎

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 because Boolean programs over it can simulate DNF formulas [Thé89]. Monoid BA_2 is the syntactic monoid of $(ab)^\star$, has 6 elements too, and is provably not universal. Multiplication tables for both are given in Figures 1 and 2.

	1	a	b	c	d	0
1	1	a	b	c	d	0
a	a	a	c	c	a	0
b	b	d	0	b	0	0
c	c	a	0	c	0	0
d	d	d	b	b	d	0
0	0	0	0	0	0	0

Figure 1: Multiplication table for monoid U.

	1	a	b	c	d	0
1	1	a	b	c	d	0
a	a	0	c	0	a	0
b	b	d	0	b	0	0
c	c	a	0	c	0	0
d	d	0	b	0	d	0
0	0	0	0	0	0	0

Figure 2: Multiplication table for monoid 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 easily seen to require an exponential number of queries too.

Theorem 21 *Expressions over monoid U are not learnable from subexponentially many Evaluation queries, even using unbounded computation time. The same holds for monoid BA_2.*

Proof. (Monoid U) We show that for each 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].

We use the names for the elements of U as in [Thé89]. There, a construction is given that, for each monotone DNF formula F, produces an expression P 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 $P(y) = c$. The problem now is that a learning strategy can also ask Evaluation queries with assignments containing the other monoid elements (b, c, d, and 0). We make sure that any such Evaluation query can be answered immediately as 0 without any teacher.

Consider the expression $E(x) = axbxc$. We have $E(1) = E(a) = c$, and $E(b) = E(c) = E(d) = E(0) = 0$. 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 with assignments over $\{1, a\}$.

(Monoid BA_2) View BA_2 as the syntactic monoid of the language $(ab)^\star$ over $\{a, b\}$. Its 6 elements are 1, 0, a, b, ab, and ba. Element 1 stands for the empty string (the identity of the monoid), and element 0 stands for any word with two consecutive as or two consecutive bs (as cannot be extended to a word in the language).

Consider the pattern $T_1T_2....T_n$ where each subexpression T_i is either abx_ib or ax_iab. This gives 2^n expressions computing 2^n different functions.

Fix one such expression E as the unknown target. Consider any assignment w to $x_1, \ldots, x_n$. If the assignment contains a 0, then clearly $E(w) = 0$. If it contains 1, ab, or ba we get at least two consecutive a's or two consecutive b's so that again $E(w) = 0$. Otherwise, w contains only a's and b's. If w contains b's exactly in the variables surrounded by a's and a's exactly in the variables surrounded by b's, then $E(w) = (abab)^n = ab$. For any other w, $E(w) = 0$. So each E of that special form is 0 on all but one of these 2^n inputs. The standard adversary argument shows that $2^n - 1$ Evaluation queries are needed to learn the expression. ∎

Theorem 20 says that an easy-to-find, polynomial-size set of Evaluation queries identifies uniquely any target over a **DA** monoid. 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 monoid element a, and that every idempotent monoid is trivially aperiodic.

Theorem 22 *Expressions over aperiodic commutative monoids can be learned by n Evaluation queries.*

Proof. Let M be an aperiodic commutative monoid satisfying the equation $m^t = m^{t+1}$ and let 1 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 $Eval(1^n)$ will return the constant c. In $O(|M|^2)$ steps, we can then 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, we then ask $Eval(1^{i-1}m1^{n-i})$. Let the returned answer be cm^{β_i} for some $\beta_i \leq r$. 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 $cm_{i+1}^{\alpha_{i+1}} = cm_{i+1}^{\beta_{i+1}}$ and commutativity, we can deduce that $cm_1^{\alpha_1} \ldots m_i^{\alpha_i} m_{i+1}^{\alpha_{i+1}} = cm_1^{\alpha_1} \ldots m_i^{\alpha_i} m_{i+1}^{\beta_{i+1}}$. Using the inductive equality on the right-hand-side, we see that the induction step goes through. The case $i = n$ proves that our proposed expression is equivalent to the original one.

Note that the algorithm runs in polynomial time when M is given as input by its multiplication table. ∎

For the following two theorems we need some definitions. For a monoid M, define a pre-order $\leq_R$ by $x \leq_R y$ iff x and y generate the same right ideal, that is $xM \subseteq yM$. Elements x and y are in the same *R-class* if $x \leq_R y$ and $y \leq_R x$, i.e., if $xM = yM$. A monoid is called *R-trivial* if each R-class contains exactly one element, or, in other words, if $xM = yM$ implies $x = y$. Symmetrically, we can define the pre-order $\leq_L$, L-classes and L-trivial monoids using Mx instead of xM. Pre-order $\leq_J$ is defined by $x \leq_J y$ iff $MxM \subseteq MyM$, and other definitions are analogous. A monoid is *J-trivial* if $MxM = MyM$ implies $x = y$.

Theorem 23 *Expressions over idempotent R-trivial monoids are learnable in polynomial time from Evaluation queries.*

Proof. Since idempotent R-trivial monoids satisfy the equation $xyx = xy$, we can assume that any constant or variable appearing in an expression 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 set 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 $w_i = 1^{i-1} m 1^{n-i}$ for any $m \neq 1$. This will tell us what 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, there exist $s \neq t$ which are in the same J-class, and these must satisfy $st = t$ and $ts = s$. For every pair of variables x_i, x_j that both appear 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 if the returned value is s or t, we will 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 hypothesis, we can use the set of Evaluation queries described in the proof of Theorem 20 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. Else, 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_1M and can thus be learned by recurrence on the cardinality of the monoid. It remains to order the variables appearing in E_0. If $stc_1 \ldots c_k = tsc_1 \ldots c_k$ 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 earlier. ∎

Next we show that the intersection of both aperiodic commutative and idempotent R-trivial is, in addition, learnable from Equivalence queries. It is easy to check that this intersection is precisely idempotent commutative monoids.

Theorem 24 *Expressions over idempotent commutative monoids can be learned with a linear number of equivalence queries.*

Proof. We use the fact that all idempotent commutative finite monoids are J-trivial [Eil76]. Let M be an idempotent commutative monoid. An expression over M can be assumed to have the form $cx_{i_1} \ldots x_{i_r}$, where $1 \leq i_1 < \ldots < i_r \leq n$. We suppose that the constant is known since the learner can run the following algorithm with, in turn, every possible value for c until it succeeds. The first step is to propose the expression $cx_1 \ldots x_n$. If possible, the teacher will return a counterexample $(m_1, \ldots, m_n)$, where $d = cm_{i_1} \ldots m_{i_r} \neq cm_1 \ldots m_n = e$. We always have $e \leq_J d$. Clearly $d \leq_J c$: if $d \leq_J m_i$ for each i, we could use idempotence to deduce that $d \leq_J e$. Since M is J-trivial, we would conclude $d = e$, a contradiction. We can thus remove from our proposed expression every x_i such that $d \leq m_i$ does not hold. Continuing in this way, we eventually identify the target expression. ∎

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

Theorem 25 *Expressions over a non-idempotent aperiodic monoid can not be 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 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.

For an M-expression E, 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 $2^{n/2} - k \cdot (n/2+1)$ candidates remain in S. ∎

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 above proof for this case but we will omit these details since the focus of our paper is mostly on groups and aperiodics.

8 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 in which cases 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 group classes considered. It would be interesting to find a general argument showing that if expressions over 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 26 *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. ∎

The simulation approach like 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 31 and the comments after it, no such simulation is possible.

Proposition 27 *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 (in the same protocol).*

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. ∎

Theorem 28 *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): Note first that every program over Abelian G we reorder instructions so that all instructions reading x_1 appear at the beginning, followed by all instructions that read x_2, etc. Then, a block of instructions reading the same variable can be collapsed into a single one by adding all the functions in the block. Furthermore, we can assume that $f(e) = e$ 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 $(1, f_c)(1, f_1)(2, f_2)(3, f_3)\dots(n, f_n)$, where f_c is the constant c function and $f_i(e) = e$ for all i. The only task is then finding functions f_c, f_1, $\dots$, f_n.

Query for Evaluation the all-identities vector; the answer is the constant c. Then, for each i and each $a \in G \setminus \{e\}$, query the vector having a in 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, $\dots$, f_n.

Part b) follows from Proposition 26 and part b) of Theorem 3. ∎

Theorem 29 *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 4, rewrite P in the form

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

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 5 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 10.

Theorem 30 *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. The Equivalence queries and the output of the algorithm are pairs formed by a decision program over the ring Z_p and a program over Z_q with domain $Z_p \times Z_q$.*

Proof. Let P be a program over G. We first sketch how to learn the Z_q component of P from Evaluation queries; this is a simple adaptation of the proof of Theorem 28, part a).

Note that, if we are concerned only with the Z_q component of P's result, we can freely permute instructions and obtain programs with the same Z_q behavior. So, as in Theorem 28, part a), we can rewrite P as $P' = (1, c)(1, f_1)(2, f_2)\ldots(n, f_n)$, where c is a constant function and $f_i(e) = e$ for all i. That is, $Z_q(P(w)) = Z_q(P'(w))$ for all $w \in G^n$. We can also modify the instructions in P' so that $Z_p(P'(w)) = 0$ for all w. Now, it is easy to use Evaluation queries to find out $Z_q(c)$ and $Z_q(f_i(x))$ for all i and $x \in G$, and hence reconstruct P'.

Next we show how to learn the Z_p component of P. Let 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 9, 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 $g_{i,j} : Z_p \times Z_q \mapsto Z_p$ is a set of unknown binary functions. This expression is a sum of products, which reminds of an MA, but not directly computable by an MA over Z_p because 1) it depends on variables $Z_q(x_i)$, which are not in Z_p; 2) MA can apply unary transformations to each of its inputs, but not necessarily binary transformations to pairs of variables; note however that only the pairs to which the $g_{i,j}$ functions are applied are very particular.

We 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.

Then, concatenate $M_{i,j}$ for all $i = 1, \ldots, n$, and add for $j = 1, \ldots, m$, following expression (2). This gives an MA M. In analogy with Theorem 9, M has the following properties:

- if, for all i, $v_{i,1}, \ldots, v_{i,\ell}$ encode 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 10.

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. ∎

We dealt with programs over nonsolvable groups in Corollary 19. For programs over nongroups, we have a strong negative result.

Theorem 31 *If 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 function $g_m : M \to M$ 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. A standard argument shows that learning this family of programs plus the constant-w program requires $|M|^n$ Evaluation queries. ∎

Theorem 31 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 11, 12, and 13. We thank José L. Balcázar for helpful comments.

References

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

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

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

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

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

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

[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*, 52:421–433, 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.

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

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

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

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

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

[GR99] M. Goldmann and A. Russell. The complexity of solving equations over finite groups. In *Proc. 14th IEEE Conference on Computational Complexity*, pages 80–86, 1999.

[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.

[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.

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

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