possible generalisation: polynomial automata with only one nonlinear variable

On Learning Polynomial Recursive Programs

ALEX BUNA-MARGINEAN, Department of Computer Science, University of Oxford, UK
VINCENT CHEVAL, Department of Computer Science, University of Oxford, UK
MAHSA SHIRMOHAMMADI, Université Paris Cité, CNRS, IRIF, France
JAMES WORRELL, Department of Computer Science, University of Oxford, UK

We introduce the class of P-finite automata. These are a generalisation of weighted automata, in which the weights of transitions can depend polynomially on the length of the input word. P-finite automata can also be viewed as simple tail-recursive programs in which the arguments of recursive calls can non-linearly refer to a variable that counts the number of recursive calls. The nomenclature is motivated by the fact that over a unary alphabet P-finite automata compute so-called P-finite sequences, that is, sequences that satisfy a linear recurrence with polynomial coefficients. Our main result shows that P-finite automata can be learned in polynomial time in Angluin's MAT exact learning model. This generalises the classical results that deterministic finite automata and weighted automata over a field are respectively polynomial-time learnable in the MAT model.

CCS Concepts: • **Theory of computation → Quantitative automata**; **Active learning**.

Additional Key Words and Phrases: Weighted automata, Exact learning, Holonomic sequences, P-finite sequences, Automata learning

ACM Reference Format:
Alex Buna-Marginean, Vincent Cheval, Mahsa Shirmohammadi, and James Worrell. 2023. On Learning Polynomial Recursive Programs. 1, 1 (October 2023), 30 pages.

1 INTRODUCTION

A central problem in computational learning is to determine a representation of a function through information about its behaviour on specific inputs. This problem encapsulates one of the main challenges in the analysis and verification of systems and protocols—namely, inferring an abstract model of a black-box system from a specification or a log of its behaviour.

In the case of functions represented by automata, one of most influential and well-known formalisations of the learning problem is the *minimally adequate teacher* (MAT) model, introduced by Dana Angluin [Angluin 1987]. In this framework a learning problem is specified by a semantic class of functions and a syntactic class of representations (e.g., the class regular languages, represented by deterministic finite automata) and the goal of the learner is to output a representation of a given *target function* by making *membership* and *equivalence* queries to a teacher. In a membership query the algorithm asks the teacher the value of the target function on a specific argument, whereas in an equivalence query the algorithm asks whether its current hypothesis represents the target function and, if not, receives as counterexample an argument on which the hypothesis and target differ. This framework is sometimes referred to as *active learning*, since the learner actively gathers information rather than passively receiving randomly chosen examples, as in Valiant's PAC learning model. Another difference with the PAC model is that in the latter the hypothesis

Authors' addresses: Alex Buna-Marginean, alex.bunamarginean@spc.ox.ac.uk, Department of Computer Science, University of Oxford, Oxford, UK; Vincent Cheval, vincent.cheval@cs.ox.ac.uk, Department of Computer Science, University of Oxford, Oxford, UK; Mahsa Shirmohammadi, mahsa@irif.fr, Université Paris Cité, CNRS, IRIF, Paris, France; James Worrell, jbw@cs.ox.ac.uk, Department of Computer Science, University of Oxford, Oxford, UK.

output by the learner is only required to be approximately correct, while in the MAT model it should be an exact representation of the target function.

In the MAT model, we say that a given learning algorithm runs in polynomial time if its running time is polynomial in the shortest representation of the target concept and the length of the longest counterexample output by the teacher. The running time is, by construction, an upper bound on the total number of membership and equivalence queries. Among other contributions [Angluin 1987] introduced the L^* algorithm: a polynomial-time exact learning algorithm for regular languages, using the representation class of deterministic finite automata. The L^* algorithm essentially tries to discover and distinguish the different Myhill-Nerode equivalence classes of the target language. By now there are several highly optimized implementations of the basic algorithm, including in the LearnLib26 and Libalf packages [Bollig et al. 2010; Isberner et al. 2015].

For many applications, such as interface synthesis, network protocols, and compositional verification, deterministic finite-state automata are too abstract and inexpressive to capture much of the relevant behaviour. This has motivated various extensions of Angluin's L^* algorithm to more expressive models, such as non-deterministic, visibly pushdown, weighted, timed, register, and nominal automata [Bollig et al. 2009; Howar et al. 2019; Michaliszyn and Otop 2022; Moerman et al. 2017]. The current paper considers an extension of weighted automata. The class of weighted automata over a field was introduced by Schützenberger [Schützenberger 1961] and has since been widely studied in the context of probabilistic automata, ambiguity in non-deterministic automata, and formal power series. A weighted automaton is a non-deterministic finite automaton whose transitions are decorated with constants from a weight semiring. Here we focus on the case that the weight semiring is the field $\mathbb{Q}$ of rational numbers. Although weighted automata over a field are strictly more expressive and exponentially more succinct than deterministic automata, the class remains learnable in polynomial time in the MAT model [Beimel et al. 1999]. By contrast, subject to standard cryptographic assumptions [Angluin and Kharitonov 1995] non-deterministic finite automata are not learnable in the MAT model with polynomially many queries.

Contributions of this paper. We introduce and study a generalisation of weighted automata, which we call *P-finite automata*, in which each transition weight is a polynomial function of the length of the input word. Over a unary alphabet, whereas weighted automata represent C-finite sequences (sequences that satisfy linear recurrences with constant coefficients), P-finite automata represent so-called P-finite sequences (those that satisfy linear recurrences with polynomial coefficients). P-finite sequences are a classical object of study in combinatorics and the complexity analysis of algorithms [Kauers and Paule 2011]. P-finite automata can thus be considered as a common generalisation of P-finite sequences and $\mathbb{Q}$-weighted automata. In Section 2 we also view weighted and P-finite automata as simple tail-recursive programs.

The main results of the paper involve two different developments of the problem of learning $\mathbb{Q}$-weighted automata, respectively involving more general and more specific representation classes.

- The most important contribution concerns a generalisation of the algorithm of [Beimel et al. 1999] for learning $\mathbb{Q}$-weighted automata. We give a polynomial-time learning algorithm for the class of P-finite automata in the MAT model. As a stepping stone to this result we show that the equivalence problem for P-finite automata is solvable in polynomial time.
- In a second direction we consider the special case of the learning problem for $\mathbb{Q}$-weighted automata in which the target function is assumed to be integer valued. Clearly the algorithm of [Beimel et al. 1999] can be applied in this case, but its final output and intermediate equivalence queries may be $\mathbb{Q}$-weighted automata. On the other hand, it was shown in [Fliess 1974] that if a $\mathbb{Q}$-weighted automaton gives an integer weight to every word then it has

a minimal representation that is a $\mathbb{Z}$-weighted automaton. Thus, in the case of an integer-valued target it is natural to ask for a learning algorithm that uses $\mathbb{Z}$-weighted automata as representation class. We give such an algorithm, running in polynomial time, and show how it can be implemented using division-free arithmetic. The heart of this construction is to give a polynomial-time procedure to decide whether a $\mathbb{Q}$-weighted automaton is $\mathbb{Z}$-valued and, if yes, to output an equivalent minimal $\mathbb{Z}$-weighted automaton.

Related Work. In the case of a unary alphabet, P-finite automata are closely related to the matrix representations of P-finite sequences considered in [Reutenauer 2012]. Over general alphabets P-finite automata can be seen as a very special case of the polynomial automata of [Benedikt et al. 2017]. However, while determining equivalence of P-finite automata is in polynomial time, checking equivalence of polynomial automata is non-primitive recursive. The key difference is that in the case of P-finite automata one works with modules over univariate polynomial rings, which are principal ideal domains, rather than general polynomial rings, which are merely Noetherian. The former setting yields much smaller bounds on the length of increasing chains of modules (compare, e.g., Proposition 3.1 herein with [Benedikt et al. 2017, Theorem 2]). but not $\mathbb{Z}[x]$

The problems of learning automata with weights in principal ideal domains (such as the ring $\mathbb{Z}$ of integers and the ring $\mathbb{Q}[x]$ of univariate polynomials with rational coefficients) was investigated in [van Heerdt et al. 2020]. That paper relies on the fact that finitely generated modules over principal ideal domains are Noetherian for the termination of the learning algorithm. The methods of the paper do not address the question of the query and computational complexity of the learning problem. The paper also leaves open the question of learning minimal representations of a given target function. Here we give a method that runs in polynomial time in the case of automata with weights in $\mathbb{Z}$ and $\mathbb{Q}[x]$ and that learns minimal representations.

2 OVERVIEW

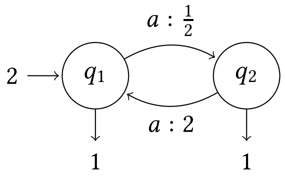
Linear Tail-Recursive Programs.

The weighted extensions of automata that are currently studied in the literature are able to model simple classes of tail-recursive programs, including linear recurrences. Consider Program 1, which reads a string of a's letter-by-letter from the input, and computes the function $f_1 : \{a\}^* \to \mathbb{Z}$ such that

$$f_1(a^k) = \begin{cases} 2 & k \equiv 0 \pmod 2 \\ 1 & \text{otherwise.} \end{cases}$$

Program 1: A linear tail-recursive program computing f_1.

```
def prog($y_1, y_2$)=
    match read() with
        | None -> return $y_1 + y_2$
        | Some a -> prog($2y_2, \frac{1}{2}y_1$)
def main()= prog(2,0)
```



The above program can be modelled by a *weighted automaton* with two states q_1 and q_2, as depicted on the right above. The states q_i represent the output of f, based on the congruence classes modulo 2. Intuitively speaking, the weight of a word is the sum of the weights of all runs of the automaton over the word, where the weight of a run is the product of weights of its starting state, of each transition taken along the run, and of its last state. For the automaton of Program 1 the non-zero initial weights are shown by incoming arrows to the states, whereas final weights are shown by outgoing arrows; each transition is also labelled by the letter a and its weights.

Program 2: Scheme of linear tail-recursive programs

```
def prog($\boldsymbol{y}$)=
    match read() with
        | None -> return $\boldsymbol{y\beta}$
        | Some a -> prog($\boldsymbol{y}\mu(a)$)
        | Some b -> prog($\boldsymbol{y}\mu(b)$)
        ⋮
def main()= prog($\boldsymbol{\alpha}$)
```

Formally, a $\mathbb{Q}$-weighted automaton $\mathcal{A} = (\boldsymbol{\alpha}, \mu, \boldsymbol{\beta})$ of dimension n over an alphabet Σ is defined by the initial weight vector $\boldsymbol{\alpha} \in \mathbb{Q}^{1\times n}$, a transition function $\mu : \Sigma \longrightarrow \mathbb{Q}^{n\times n}$ and the final weight vector $\boldsymbol{\beta} \in \mathbb{Q}^{n\times 1}$. The semantics of $\mathcal{A}$, denoted by $[\![\mathcal{A}]\!] : \Sigma^* \longrightarrow \mathbb{Q}$, maps each word $w = \sigma_1 \cdots \sigma_k$ to its weights computed as $\boldsymbol{\alpha}\mu(\sigma_1)\ldots\mu(\sigma_k)\boldsymbol{\beta}$. The automaton of Program 1 is formally defined as

$$\boldsymbol{\alpha} := \begin{bmatrix} 2 & 0 \end{bmatrix} \qquad \mu(a) := \begin{bmatrix} 0 & \frac{1}{2} \\ 2 & 0 \end{bmatrix} \qquad \boldsymbol{\beta} := \begin{bmatrix} 1 \\ 1 \end{bmatrix}.$$

The automaton of Program 1 is a unary(-alphabet) automaton over $\mathbb{Q}$; it is well-known that unary weighted automata over a field coincide with linear recurrence sequences [Berstel and Reutenauer 2010] over the field. Recall that a rational sequence $\{u_i\}_{i=1}^{\infty}$ is a linear recurrence sequence of order d if it satisfies a recurrence relation of the form

$$u_n = c_{d-1}u_{n-1} + \ldots + c_1 u_{n-d+1},$$

where $c_i \in \mathbb{Q}$ and $c_1 \neq 0$. The Fibonacci sequence, for example, is given by $F_0 = F_1 = 1$ and $F_k = F_{k-1} + F_{k-2}$ for all $k \geq 2$. The corresponding Fibonacci automaton is defined by

$$\boldsymbol{\alpha} := \begin{bmatrix} 1 & 1 \end{bmatrix} \qquad \mu(a) := \begin{bmatrix} 1 & 1 \\ 1 & 0 \end{bmatrix} \qquad \boldsymbol{\beta} := \begin{bmatrix} 1 \\ 0 \end{bmatrix}.$$

The automaton computes the k-th Fibonacci number as the weight of the input a^{k-1} through its semantics $\boldsymbol{\alpha}\mu(a)^{k-1}\boldsymbol{\beta}$.

In the general setting, a recursive program computing a function $f : \Sigma^* \longrightarrow \mathbb{Q}$ can be realised by a $\mathbb{Q}$-weighted automaton if its so-called Hankel matrix has finite rank [Berstel and Reutenauer 1988]. This characterization encompasses a rich class of linear tail-recursive programs, where all assignments are linear updates of the form $\boldsymbol{y} \leftarrow \boldsymbol{y}M$, where $\boldsymbol{y} := (y_1, \ldots, y_n)$ is a tuple of variables and $M \in \mathbb{Q}^{n\times n}$. See Program 2 for a schematic illustration of such linear recursive programs. A~~n~~ $\mathbb{Q}$-weighted automaton for such programs is defined accordingly as $(\boldsymbol{\alpha}, \mu, \boldsymbol{\beta})$ over the alphabet Σ.

Before we proceed, we note that in weighted automata the weight growth of each word w is bounded by $c^{|w|}$ for a fixed positive constant $c \in \mathbb{Z}$. In the next section, we will see that, in our proposed extension of weighted automata, the weight growth of each word w can be of magnitude $(c_1|w|)^{c_2|w|}$ where $c_1, c_2 \in \mathbb{Z}$ are fixed positive constants. $n^{O(n)}$

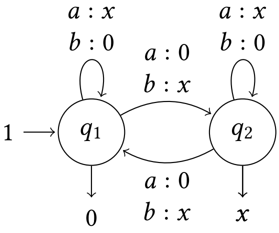
Polynomial Tail-Recursive Programs.

Our proposed P-recursive programs will have a program counter x, that initially is set to zero and monotonously increases by one after each input letter, in order to store the length of the word. The updates on each variable y_i is now in the form $y_i \leftarrow \sum_{j=1}^{n} P_j(x)y_j$ where $P_1, \ldots, P_n \in \mathbb{Q}[x]$ are univariate polynomials with rational coefficients in indeterminate x. Program 3 computes the following function $f_2 : \{a,b\}^* \longrightarrow \mathbb{Z}$ defined by

$$f_2(w) = \begin{cases} (|w|+1)! & \text{if } w \text{ contains an odd number of } b\text{'s,} \\ 0 & \text{otherwise.} \end{cases}$$

Program 3: A P-recursive program computing f_2.

```
def prog(y1, y2, x)=
    match read() with
      | None -> return x y2
      | Some a -> prog(x y1, x y2, x + 1)
      | Some b -> prog(x y2, x y1, x + 1)
def main()= prog(1, 0, 1)
```



We show that such *P*-recursive programs can be realised by our proposed extension of weighted automata, which we call *P-finite automata.* This extension can be thought of as a symbolic weighted automata where transition weights, as well as final weights, are parameterized by an indeterminate x. Along the execution of a P-finite automaton over an input word, the value of indeterminate x stores the length of the input read so far.

In the P-finite automaton representing Program 3 there are two states corresponding to the variables y_1 and y_2. As is the case for weighted automata, the weight of a word is the sum of the weight of all runs of the automaton over the word, where the weight of a run is the product of weights of its starting state, of each transition taken along the run, and of its last state. The main difference is that the transition and final weights change in every step, as the value of x gets updated after every new input letter. For instance, the weight of ab is 3! computed by

$$\underbrace{1}_{\text{initial weight of } q_1} \cdot \overbrace{1}^{\text{weight of } q_1 \xrightarrow{a:1} q_1} \cdot \overbrace{2}^{\text{weight of } q_1 \xrightarrow{b:2} q_2} \cdot \underbrace{3}_{\text{final weight of } q_2 : 3}$$

Formally, a P-finite automaton $\mathcal{A} = (\boldsymbol{\alpha}, \mu, \boldsymbol{\beta}(x))$ of dimension n over an alphabet Σ is defined by the initial weight vector $\boldsymbol{\alpha} \in \mathbb{Q}^n$, a transition function $\mu : \Sigma \longrightarrow \mathbb{Q}[x]^{n \times n}$, and the final weight vector $\boldsymbol{\beta}(x) \in \mathbb{Q}[x]^n$. In the sequel, for simplicity of notations we use $\mu(\sigma, k)$ instead of $\mu(\sigma)(k)$, with $\sigma \in \Sigma$ and $k \in \mathbb{N}$. The semantics of $\mathcal{A}$, denoted by $[\![\mathcal{A}]\!] : \Sigma^* \longrightarrow \mathbb{Q}$, maps each word $w = \sigma_1 \cdots \sigma_k$ to

$$[\![\mathcal{A}]\!](w) := \boldsymbol{\alpha}\mu(\sigma_1, 1) \dots \mu(\sigma_k, k)\boldsymbol{\beta}(k+1) \,.$$

Although the initial vector $\boldsymbol{\alpha}$ is a vector of rationals, one can also look at it as a vector of polynomials (similar to the final vector $\boldsymbol{\beta}(x)$) that is always evaluated on 0 as it would lead to an equivalent semantics $\boldsymbol{\alpha}(0)\mu(\sigma_1, 1) \dots \mu(\sigma_k, k)\boldsymbol{\beta}(k+1)$. The automaton of Program 3 is formally defined as

$$\boldsymbol{\alpha} := \begin{bmatrix} 1 & 0 \end{bmatrix} \qquad \mu(a) := \begin{bmatrix} x & 0 \\ 0 & x \end{bmatrix} \qquad \mu(b) := \begin{bmatrix} 0 & x \\ x & 0 \end{bmatrix} \qquad \boldsymbol{\beta} := \begin{bmatrix} 0 \\ x \end{bmatrix} .$$

Unary P-finite automata coincide with monic *P*-recursive sequences. A rational sequence $\{u_i\}_{i=1}^{\infty}$ is a (monic) *P*-recursive sequence of order d if it satisfies a recurrence relation of the form

$$u_n = P_{d-1}u_{n-1} + \dots P_1 u_{n-d+1},$$

where $P_i \in \mathbb{Q}[x]$ and $P_1 \neq 0$. Another example of monic *P*-recursive sequences comes from the famous recurrence for the number of involutions, found by Heinrich August Rothe in 1800. An

Program 4: Scheme of P-recursive programs

```
def prog($\boldsymbol{y}, x$)=
    match $read()$ with
        | $None$ -> return $\boldsymbol{y}\boldsymbol{\beta}(x)$
        | $Some\ a$ -> prog($\boldsymbol{y}M_a(x), x+1$)
        | $Some\ b$ -> prog($\boldsymbol{y}M_b(x), x+1$)
def main()= prog($\boldsymbol{\alpha}, 1$)
```

involution on a set $\{1, 2, \ldots, k\}$ is a self-inverse permutation. The number of involutions, including the identity involution, is given by $I_0 = I_1 = 1$ and $I_k = I_{k-1} + (k-1)I_{k-2}$ for $k \geq 2$. The corresponding P-finite automaton is defined by

$$\boldsymbol{\alpha} = \begin{bmatrix} 1 & 1 \end{bmatrix} \qquad \mu(a) = \begin{bmatrix} 1 & 1 \\ x & 0 \end{bmatrix} \qquad \boldsymbol{\beta} = \begin{bmatrix} 1 \\ 0 \end{bmatrix}.$$

The P-finite automaton computes the number of involutions of $\{1, \ldots, k\}$ as the weight of the input a^{k-1} through its semantics $\boldsymbol{\alpha} \prod_{i=1}^{k-1} \mu(a, i)\boldsymbol{\beta}(k)$. See Program 4 for a schematic illustration of a class of polynomial tail-recursive programs that can be realized by a P-finite automata.

P-Solvable Loops and Extensions

The model of P-recursive programs (or P-finite automata) bears similarities with the notion of *P*-solvable loops [Kovács 2008] and its extensions [Humenberger et al. 2017a,b]. The latter are studied in the context of program analysis and invariant synthesis in particular.

The class of P-solvable loops is subsumed by that of *linear* tail-recursive programs, as P-solvable loops allow only linear updates of program variables [Kovács 2008]. We have also the class of *extended P-solvable loops* [Humenberger et al. 2017a,b], in which the sequence of values assumed by a program variable is a sum of *hypergeometric sequences.* A hypergeometric sequence $(u_n)_{n=0}^{\infty}$ is one that satisfies a polynomial recurrence $u_n = r(n)u_{n-1}$ for all $n \geq 1$, where $r(x) \in \mathbb{Q}(x)$ is a rational function. The class of extended P-solvable loops is thus incomparable with P-finite automata. On the one hand, hypergeometric recurrences allow multiplication by rational functions (such as $r(x)$ above), not just polynomials. On the other hand P-finite automata over a unary alphabet can define sequences that are not sums of hypergeometric sequences (see [Reutenauer 2012, Section 10]).

```
while $true$ do
    $a := 2(x+1)(x+\frac{3}{2})a$
    $b := 4(x+1)b$
    $c := \frac{1}{2}(x+\frac{3}{2})c$
    $x := x+1$
```

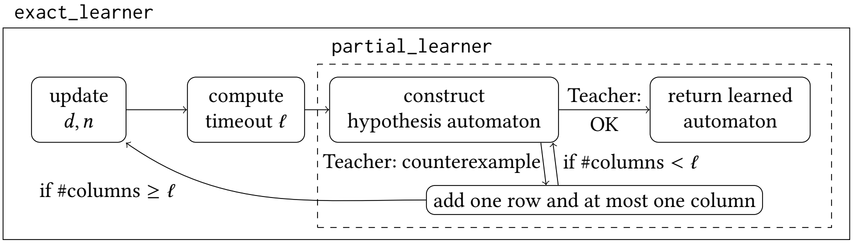
```
def prog($a, b, c, x$)=
    match $read()$ with
        | $None$ -> return $(a, b, c)$
        | $Some\ \_$ -> prog($2(x+1)(x+\frac{3}{2})a, 4(x+$
          $1)b, \frac{1}{2}(x+\frac{3}{2})c, x+1$)
```

The program shown above on the left is an example of an extended P-solvable loop, taken from [Humenberger et al. 2017b]. The corresponding P-recursive program is shown on the right. Since the focus of [Humenberger et al. 2017b] is on invariant generation they consider loops that run forever. In our P-recursive programs any input letter invokes the recursive call.

Learning Algorithm

The high-level structure of the algorithm for learning *P*-finite automata is shown in the diagram below. The algorithm consists of a main procedure `exact_learner` and a subroutine `partial_learner`.

It is also not assumed to know *a priori* an upper bound n on the number of states of the target automaton nor a degree bound d on the polynomials appearing therein. Hence the procedure `exact_learner` searches through pairs of possible values of d and n and for each such pair it calls `partial_learner` that tries to learn a target automaton subject to these bounds.



The subroutine `partial_learner` can be seen as a generalisation of the algorithm of [Beimel et al. 1999] for learning $\mathbb{Q}$-weighted automata. As in [Beimel et al. 1999], the basic data structure, which we call the *table*, is a finite fragment of the Hankel matrix of the target function $f : \Sigma^* \to \mathbb{Q}$. Formally speaking, the table is a finite matrix whose rows and columns are labelled by words and such that the entry with index $(u, v) \in \Sigma^* \times \Sigma^*$ is $f(uv)$. We will later on denote this table by $H(\mathcal{R}, C)$ where $\mathcal{R}$ and C are the sequences of words labelling the rows and columns of the table. The table is used to construct a hypothesis automaton. This involves making membership queries to interpolate polynomials that label the state-to-state transitions of the automaton. Since there is a bound d on the maximum degree of the polynomials, the process of interpolation is reduced to solving a system of linear equations.

Once constructed, the hypothesis automaton is passed to the teacher. If the hypothesis is correct, the algorithm terminates and returns the hypothesis automaton; if it is incorrect, the counterexample given by the teacher is used to augment the table by adding a new row and at most one column (using membership queries to fill in the missing table entries). After augmenting the table, a new hypothesis automaton can be constructed.

For any given run of `partial_learner`, since the degree bound d may not be sufficient to learn the target automaton, there is a timeout ℓ (a function of d and n) on the run of `partial_learner`. If the timeout is reached then the run is abandoned and control returns to `exact_learner`.

Going back to the case of $\mathbb{Q}$-weighted automata, the termination (and polynomial-time bound) of the learning algorithm of [Beimel et al. 1999] relies on the classical result of Carlyle and Paz that a function $f : \Sigma^* \to \mathbb{Q}$ is recognisable by a $\mathbb{Q}$-weighted automaton if and only if its Hankel matrix has finite rank. The idea is that every unsuccessful equivalence query results in the rank of the table increasing by one, and so the number of equivalence queries is at most the rank of the Hankel matrix of the target function. Such a result is not available in the case of P-finite automata.

The termination proof and polynomial complexity bound for `exact_learner` rely on an analysis of the timeout. For this we associate with a run of `partial_learner` an increasing chain of submodules over the polynomial ring $\mathbb{Q}[x]$, whose length is the number of equivalence queries. Since $\mathbb{Q}[x]$ is a Noetherian ring, such a chain must have finite length. We give a novel fine-grained analysis of the maximum length of an increasing chain of modules over $\mathbb{Q}[x]$ to guarantee that we will learn the target automaton within the timeout if the degree bound parameter is sufficiently large. (This analysis even allows us to obtain a polynomial bound on the overall computational and query complexity of our learning algorithm.) We highlight that in contrast to the case of $\mathbb{Q}$-weighted automata, the length of this chain of modules depends on the length of the counterexamples returned by the teacher. As an intermediate result, we use this analysis to show that equivalence of P-finite automata is decidable in polynomial time.

even PID

Our analysis of increasing chains of modules over $\mathbb{Q}[x]$ applies equally well to $\mathbb{Z}$. We use the version for $\mathbb{Z}$ to give a polynomial-time algorithm to decide whether or not the function recognised by a given $\mathbb{Q}$-weighted automaton is $\mathbb{Z}$-valued. We then observe that such an algorithm can be used to reduce the problem of learning $\mathbb{Z}$-weighted automata to that of learning $\mathbb{Q}$-weighted automata.

3 BACKGROUND ON MODULE THEORY

Let R be a commutative ring with unity. An *ideal* of R is an additive subgroup $I \subseteq R$ such that $ra \in I$ for all $r \in R$ and $a \in I$. The ring R is said to be a *principal ideal domain* (PID) if every ideal I is generated by a single element, that is, there is some $a \in R$ such that $I = \{ra : r \in R\}$. We will mainly work with $\mathbb{Z}$ and $\mathbb{Q}[x]$, which are both PIDs.

An R*-module* M is an abelian group together with a scalar multiplication $(\cdot) : R \times M \to M$ such that,

- $r \cdot (m_1 + m_2) = rm_1 + rm_2$,
- $(r_1 + r_2) \cdot m = r_1 \cdot m + r_2 \cdot m$,
- $(r_1 r_2) \cdot m = r_1 \cdot (r_2 \cdot m)$, $1_R \cdot m = m$,

for all scalars $r, r_1, r_2 \in R$, and for all elements $m, m_1, m_2 \in M$. A key example of an R-module is R^n, where $n \in \mathbb{N}$, in which addition and scalar multiplication act pointwise.

Let M be an R-module. A *submodule* of M is a subgroup that is closed under scalar multiplication. A subset $\{\boldsymbol{v}_i : i \in I\} \subseteq M$ is said to be *linearly independent* if an R-linear combination $\sum_{i \in I} r_i \boldsymbol{v}_i$ is only zero if all the r_i are zero. We write $\langle \boldsymbol{v}_i : i \in I \rangle_R$ for the *R-span* of the $\boldsymbol{v}_i$, defined by

$$\langle \boldsymbol{v}_i : i \in I \rangle_R := \left\{ \sum_{i \in I} r_i \cdot \boldsymbol{v}_i : r_i \in R \right\}.$$

commutative rings are IBN

We say that $\{\boldsymbol{v}_i : i \in I\}$ *generates* M if $M = \langle \boldsymbol{v}_i : i \in I \rangle_R$. If the $\boldsymbol{v}_i$ are, in addition, linearly independent, then we say that $\{\boldsymbol{v}_i : i \in I\}$ is a *basis* of M. If R is a PID then all submodules M of R^n have a basis and all bases have the same cardinality. called "rank"; in general for IBN rings R

A key difference between modules and vector spaces is that one can have proper inclusions between modules of the same rank. For example, we have that $15\mathbb{Z} \subsetneq 3\mathbb{Z} \subsetneq \mathbb{Z}$ are all rank-1 submodules of $\mathbb{Z}$. However it remains true that all finitely generated R-modules M are *Noetherian*: every strictly increasing chain of submodules of M is finite. A crucial ingredient in the analysis of our algorithms is an upper bound on the length of strictly increasing chains of modules in R^n. For this, we use the Smith Normal Form.

Let R be either $\mathbb{Z}$ or $\mathbb{Q}[x]$ and let $M = \langle \boldsymbol{v}_1, \ldots, \boldsymbol{v}_m \rangle_R$ be a finitely generated R-module of rank r. $\leq m, n$ Using the Smith Normal Form [Smith 1861], one can show that from the set $\{\boldsymbol{v}_1, \ldots, \boldsymbol{v}_m\}$ of generators of M, we can compute, in polynomial time, a R-basis $\boldsymbol{f}_1, \ldots, \boldsymbol{f}_n$ of R^n and elements $d_1, \ldots, d_r \in R$ such that $d_1 \boldsymbol{f}_1, \ldots, d_r \boldsymbol{f}_r$ is an R-basis of M. To be specific, the matrix

$M = \langle d_1 f_1, \ldots, d_r f_r \rangle$

$$A := \begin{bmatrix} \boldsymbol{v}_1 & \ldots & \boldsymbol{v}_m \end{bmatrix}$$

can be put in Smith Normal Form, that is, A can be written as $S\tilde{A}T$ where $S \in R^{n \times n}$ and $T \in R^{m \times m}$ are invertible matrices, and

$$\tilde{A} = diag(d_1, \ldots, d_r, 0, \ldots, 0)$$

is a diagonal matrix such that $d_i \mid d_{i+1}$ for all $1 \leq i < r$. Define $D_i(A)$ to be the greatest common divisor of the $i \times i$ minors of A for $i = 0, \ldots, r$, (so that $D_0(A) = 1$). It is known [Newman 1997] that for all $1 \leq i \leq r$ we have

$$d_i = \frac{D_i(A)}{D_{i-1}(A)}. \tag{1}$$

In the above-mentioned decomposition of A as $S\tilde{A}T$, the columns of S are in fact the R-basis $\boldsymbol{f}_1, \dots, \boldsymbol{f}_n$ of R^n, and $S\tilde{A}$ is a matrix with columns $d_1\boldsymbol{f}_1, \dots, d_r\boldsymbol{f}_r$, representing an R-basis of M.

Proposition 3.1. *Let $n, k \in \mathbb{N}$ and $M_0 \subsetneq M_1 \subsetneq \cdots \subsetneq M_k$ be a strictly increasing chain of submodules of $\mathbb{Z}^n$, all having the same rank $r \leq n$. Assume that M_0 is generated by a collection of vectors whose entries have absolute value at most B. Then $k \leq r \log B + \frac{r}{2} \log r$.*

Proof Sketch. By assumption, there are vectors $\boldsymbol{v}_1, \dots, \boldsymbol{v}_m \in \mathbb{Z}^n$ that generate M_0 and whose entries have absolute value at most B. Using Smith Normal Form, there exists a basis $\boldsymbol{f}_1, \dots, \boldsymbol{f}_n$ of $\mathbb{Z}^n$ and positive integers $d_1, \dots, d_r$, such that $d_1\boldsymbol{f}_1, \dots, d_r\boldsymbol{f}_r$ is a basis of M_0. Furthermore, by Equation (1) it follows that $d_1 \cdots d_r$ is the greatest common divisor of all $r \times r$ minors of the $n \times m$ matrix with columns $\boldsymbol{v}_1, \dots, \boldsymbol{v}_m$. By Hadamard's inequality it follows that $d_1 \cdots d_r \leq B^r r^{r/2}$.

Let M be the module generated by $\boldsymbol{f}_1, \dots, \boldsymbol{f}_r$. Since the modules $M_0, \dots, M_k$ all have rank r, they are all contained in M. Recall that the index $[M : M_0]$ of a subgroup M_0 in the group M, is the number of cosets of M_0 in M. Observe that the index $[M : M_0]$ is $d_1 \cdots d_r$. We also have $[M_{k+1} : M_k] \geq 2$ for $k = 0, \dots, n-1$ since M_k is a proper submodule of M_{k+1}. It follows that $n \leq \log(d_1 \cdots d_r) \leq r \log B + \frac{r}{2} \log r$. □

Remark 3.2. The bound in Proposition 3.1 is tight: consider $\boldsymbol{e}_1 = \begin{bmatrix} 1 & 0 & \cdots & 0 \end{bmatrix}^\top \in \mathbb{Z}^n$ and some positive integer b. The strictly increasing chain of modules of rank 1:

$$\left\langle 2^b \boldsymbol{e}_1 \right\rangle_{\mathbb{Z}} \subsetneq \left\langle 2^b \boldsymbol{e}_1, 2^{b-1}\boldsymbol{e}_1 \right\rangle_{\mathbb{Z}} \subsetneq \cdots \subsetneq \left\langle 2^b \boldsymbol{e}_1, 2^{b-1}\boldsymbol{e}_1, \cdots, \boldsymbol{e}_1 \right\rangle_{\mathbb{Z}}$$

PIDs ⇒ GCD

has length b, which is the bound given by Proposition 3.1.

PIDs

In the next Proposition, we generalize Proposition 3.1 to ~~the~~ PIDs. for ~~which~~ there exists a well-defined greatest common divisor function. A detailed proof can be found in Appendix A.

Proposition 3.3. *Let $n, k \in \mathbb{N}$. Let R be a PID and $M = \langle \boldsymbol{v}_1, \dots, \boldsymbol{v}_m \rangle$ be a R-submodule of R^n. Let A be the $n \times m$ matrix whose i-th column is $\boldsymbol{v}_i$. Let $M \subsetneq M_1 \subsetneq M_2 \subsetneq \ldots \subsetneq M_k$ be a strictly increasing chain of R-submodules of R^n, all having the same rank $r \leq n$. Then k is bounded by the number of (not necessarily distinct) prime factors of $D_r(A)$.*

counting multiplicity?

4 $\mathbb{Z}$-WEIGHTED AUTOMATA

In this section, we start by giving a procedure to decide in polynomial time whether a $\mathbb{Q}$-weighted automaton computes an integer-valued function. For every "yes" instance our procedure returns an equivalent $\mathbb{Z}$-weighted automaton and for every "no" instance it returns a word whose weight is non-integer. This algorithm can be regarded as an effective (and computationally efficient) version of the well-known fact that $\mathbb{Q}$ is a Fatou extension of $\mathbb{Z}$ [Berstel and Reutenauer 2010, Chapter 7]. As a corollary of the above procedure, we give a polynomial-time reduction of the exact learning problem for $\mathbb{Z}$-automata to the exact learning problem for $\mathbb{Q}$-automata. (One can similarly reduce the exact learning problem for automata with weights in the ring $\mathbb{Q}[x]$ to that for automata with weights in the quotient field $\mathbb{Q}(x)$.)

→ if $f \in \mathbb{Z}[[X]]$ is $\mathbb{Q}$ rat. $= P/q$, $P, q \in \mathbb{Q}[X]$
then it is $\mathbb{Z}$ rat $= P/q$, $P, q \in \mathbb{Z}[X]$

4.1 $\mathbb{Z}$-valuedness of $\mathbb{Q}$-automata

Let $\mathcal{A} = (\boldsymbol{\alpha}, \mu, \boldsymbol{\beta})$ be a $\mathbb{Q}$-*weighted automaton* of dimension n over alphabet Σ. Here $\boldsymbol{\alpha} \in \mathbb{Q}^{1 \times n}$, $\mu(\sigma) \in \mathbb{Q}^{n \times n}$ for all $\sigma \in \Sigma$, and $\boldsymbol{\beta} \in \mathbb{Q}^{n \times 1}$. We say that such an automaton $\mathcal{A}$ is $\mathbb{Z}$-*weighted* if all entries of $\boldsymbol{\alpha}, \boldsymbol{\beta}$ and those of the matrices $\mu(\sigma)$ are integers. Let I_n be the $n \times n$ identity matrix. We extend μ to a map $\mu : \Sigma^* \to \mathbb{Q}^{n \times n}$ by writing $\mu(\varepsilon) := I_n$ and $\mu(w\sigma) := \mu(w)\mu(\sigma)$ for all $\sigma \in \Sigma$ and $w \in \Sigma^*$. The semantics of $\mathcal{A}$, that is, the function computed by $\mathcal{A}$, is given by $[\![\mathcal{A}]\!] : \Sigma^* \to \mathbb{Q}$

with $[\![\mathcal{A}]\!](w) := \boldsymbol{\alpha}\mu(w)\boldsymbol{\beta}$. Automata $\mathcal{A}_1, \mathcal{A}_2$ over the same alphabet Σ are said to be *equivalent* if $[\![\mathcal{A}_1]\!] = [\![\mathcal{A}_2]\!]$. An automaton $\mathcal{A}$ is *minimal* if there is no equivalent automaton with fewer states.

Define the *forward reachability set* of $\mathcal{A}$ to be $\{\boldsymbol{\alpha}\mu(w) : w \in \Sigma^*\}$ and define the *backward reachability set* to be $\{\mu(w)\boldsymbol{\beta} : w \in \Sigma^*\}$. The *forward space* and *forward module* of $\mathcal{A}$ are respectively the $\mathbb{Q}$-subspace of $\mathbb{Q}^n$ and $\mathbb{Z}$-submodule of $\mathbb{Q}^n$ spanned by the forward reachability set, *viz.*,

$$\langle \boldsymbol{\alpha}\mu(w) : w \in \Sigma^* \rangle_{\mathbb{Q}} \qquad \text{and} \qquad \langle \boldsymbol{\alpha}\mu(w) : w \in \Sigma^* \rangle_{\mathbb{Z}} \,.$$

The backward space and backward module are defined analogously. The forward space is the smallest (with respect to inclusion) vector space that contains $\boldsymbol{\alpha}$ and is closed under post-multiplication by $\mu(\sigma)$. The forward module is likewise the smallest module that contains $\boldsymbol{\alpha}$ and is closed under post-multiplication by $\mu(\sigma)$. Analogous statements apply to the backward space and backward module.

Let $F \in \mathbb{Q}^{m_f \times n}$ with $m_f \leq n$ be a matrix whose rows form a basis of the forward space of $\mathcal{A}$. It is known that there are unique $\boldsymbol{\alpha}_f \in \mathbb{Q}^{1 \times m_f}$, $\boldsymbol{\beta}_f \in \mathbb{Q}^{m_f \times 1}$ and $\mu_f(\sigma) \in \mathbb{Q}^{m_f \times m_f}$, for all $\sigma \in \Sigma$, such that:

$$\boldsymbol{\alpha}_f F = \boldsymbol{\alpha} \qquad \mu_f(\sigma)F = F\mu(\sigma) \qquad \boldsymbol{\beta}_f = F\boldsymbol{\beta} \,. \tag{2}$$

Similarly, let $B \in \mathbb{Q}^{n \times m_b}$ with $m_b \leq n$ be a matrix whose columns form a basis of the backward space of $\mathcal{A}$. It is known that there are unique $\boldsymbol{\alpha}_b \in \mathbb{Q}^{1 \times m_b}$, $\boldsymbol{\beta}_b \in \mathbb{Q}^{m_b \times 1}$ and $\mu_b(\sigma) \in \mathbb{Q}^{m_b \times m_b}$, for all $\sigma \in \Sigma$, such that:

$$\boldsymbol{\alpha}_b = \boldsymbol{\alpha}B \qquad B\mu_b(\sigma) = \mu(\sigma)B \qquad B\boldsymbol{\beta}_b = \boldsymbol{\beta} \,.$$

The automaton $\mathcal{A}_f = (\alpha_f, \mu_f, \beta_f)$ is a *forward conjugate* of $\mathcal{A}$, and the automaton $\mathcal{A}_b = (\alpha_b, \mu_b, \beta_b)$ is a *backward conjugate* of $\mathcal{A}$. These automata are equivalent to $\mathcal{A}$, meaning that

$$[\![\mathcal{A}_f]\!] = [\![\mathcal{A}_b]\!] = [\![\mathcal{A}]\!] \,.$$

The procedure to decide $\mathbb{Z}$-valuedness of $\mathbb{Q}$-automata is a variant of the classical minimisation algorithm for $\mathbb{Q}$-weighted automata and it is described in Figure 6. Below, we first work through a subroutine used in the algorithm.

It is classical that given an automaton $\mathcal{A}$ we can compute in polynomial time a $\mathbb{Q}$-basis of the forward vector space that is comprised of vectors in the forward reachability set. An analogous result holds for the backward space [Kiefer 2020; Tzeng 1992]. The forward module need not be finitely generated in general, but it will be finitely generated if the forward reachability set is contained in $\mathbb{Z}^n$.

The procedure `compute_ℤ_generators`, shown in Figure 5, is a polynomial-time algorithm that, for an input $\mathbb{Q}$-automaton, either outputs a finite basis of the forward module of $\mathcal{A}$ or a non-integer vector in the forward reachability set. Intuitively, it builds a set of words W, starting from $\{\varepsilon\}$, by adding words that augment the module $\langle \boldsymbol{\alpha}\mu(u) : u \in W \rangle_{\mathbb{Z}}$. When no such word can be found, the set $\{\boldsymbol{\alpha}\mu(u) : u \in W\}$ will form a generating set for the forward module.

Notice that the procedure is based on a two-pass search: first we search for words that increase the rank of the forward module and then for words that augment the forward module while the rank is stable. This allows us to obtain a polynomial-time running bound through a single application of Proposition 3.1 to the second phase of the search. We do not know if it is possible to obtain a polynomial bound under arbitrary search orders.

Proposition 4.1. *The procedure* `compute_ℤ_generators`*, described in Figure 5, is a polynomial-time algorithm that given a* $\mathbb{Q}$*-automaton* $\mathcal{A} = (\boldsymbol{\alpha}, \mu, \boldsymbol{\beta})$ *of dimension* n *over alphabet* Σ *either outputs a finite set of words* W *generating the forward module of* $\mathcal{A}$*, namely,*

$$\langle \boldsymbol{\alpha}\mu(w) : w \in W \rangle_{\mathbb{Z}} = \langle \boldsymbol{\alpha}\mu(w) : w \in \Sigma^* \rangle_{\mathbb{Z}} \,,$$

Figure 5: Computing generators of the forward module or a counterexample.

```
1 def compute_$\mathbb{Z}$_generators($\mathcal{A}$)=
2     $W := \{\varepsilon\}$
      // Finding words that increase the rank    over $\mathbb{Q}$
3     while there is $(w,\sigma) \in W \times \Sigma$ such that $\boldsymbol{\alpha}\mu(w\sigma) \notin \langle \boldsymbol{\alpha}\mu(u) : u \in W \rangle_{\mathbb{Q}}$ do
4         $W := W \cup \{w\sigma\}$
5         if $\boldsymbol{\alpha}\mu(w\sigma) \notin \mathbb{Z}^n$ then return $w\sigma$
      // Finding words that augment the module
6     while there is $(w,\sigma) \in W \times \Sigma$ such that $\boldsymbol{\alpha}\mu(w\sigma) \notin \langle \boldsymbol{\alpha}\mu(u) : u \in W \rangle_{\mathbb{Z}}$ do
7         $W := W \cup \{w\sigma\}$
8         if $\boldsymbol{\alpha}\mu(w\sigma) \notin \mathbb{Z}^n$ then return $w\sigma$
9     return $W$
```

or else a word $w \in \Sigma^*$ such that $\boldsymbol{\alpha}\mu(w) \notin \mathbb{Z}^n$.

Proof. Write the entries of $\boldsymbol{\alpha}, \boldsymbol{\beta}$ and $\mu(\sigma)$, with $\sigma \in \Sigma$, as fractions over a common denominator and let B be an upper bound of the numerators and denominator of the resulting fractions. Note that the bit size of B is polynomially bounded in the length of the encoding of $\mathcal{A}$.

The first **while**-loop, in Line 3 computes a set of words $W_0 \subseteq \Sigma^{\leq n}$ such that $\{\boldsymbol{\alpha}\mu(w) : w \in W_0\}$ is a $\mathbb{Q}$-basis of the forward space of $\mathcal{A}$. By construction, the dimension of the space spanned by the set $\{\boldsymbol{\alpha}\mu(w) : w \in W_0\} \leq |W_0| \leq n$, which shows that the first **while**-loop terminates after at most n iterations.

Below, we prove that the second **while**-loop, in Line 6, terminates in polynomial time in the length of the encoding of $\mathcal{A}$. Let $W_0, W_1, W_2, \ldots$ be the successive values of the variable W during the second loop. For all $k \in \mathbb{N}$, let M_k be the $\mathbb{Z}$-module $\langle \boldsymbol{\alpha}\mu(w) : w \in W_k \rangle_{\mathbb{Z}}$. Then $M_0 \subsetneq M_1 \subsetneq \cdots$ is a strictly increasing sequence of $\mathbb{Z}$-modules, all having the same rank (namely the size of W_0, that is the dimension of the forward space).

Recall that the length of words in W_0 is at most n. A simple induction on the length of words allows us to show that for all $w \in \Sigma^*$, the entries of $\boldsymbol{\alpha}\mu(w)$ have numerators and denominators bounded by $n^{|w|-1}B^{|w|}$. In particular, we obtain that the entries in $\{\boldsymbol{\alpha}\mu(w) : w \in W_0\}$ are bounded by $n^{n-1}B^n$.

Let $k_0 := n(n - \frac{1}{2}) \log n + n^2 \log B$. Suppose that all modules $M_0, M_1, \cdots$ contain only integer vectors. Then Proposition 3.1 shows that the above sequence modules has length at most k_0. The only other possibility is that for some $k \leq k_0$ we have $M_k \not\subseteq \mathbb{Z}^n$ and hence $\boldsymbol{\alpha}\mu(w) \notin \mathbb{Z}^n$ for some word $w \in W_k$. In either case, the number of iterations of the while loop is at most k_0.

It follows that each set W_k consists of at most $k_0 + n$ words, each of length at most $k_0 + n$. Thus the set of vectors $\{\boldsymbol{\alpha}\mu(w) : w \in W_k\}$ has description length polynomial in $\mathcal{A}$. Each iteration of the while loop involves solving $|W| \cdot |\Sigma|$ systems of linear equations over $\mathbb{Z}$ to determine membership in the module generated by $\{\boldsymbol{\alpha}\mu(w) : w \in W_k\}$. Again, this requires time polynomial in $\mathcal{A}$. Altogether, the algorithm runs in polynomial time.

If the loop terminates by returning $W \subseteq \Sigma^*$ then $\{\boldsymbol{\alpha}\mu(w) : w \in W\}$ contains $\boldsymbol{\alpha}$ and is closed by multiplication on the right by $\mu(\sigma)$ for all $\sigma \in \Sigma$. Thus this module is the forward module of $\mathcal{A}$. □

Figure 6: Computing a $\mathbb{Z}$-weighted automaton from a $\mathbb{Q}$-weighted automaton.

```
1  def compute_ℤ_automaton($\mathcal{A} = (\boldsymbol{\alpha}, \mu, \boldsymbol{\beta})$)=
       // Compute a basis of the backward space
2      $W_B := \{\varepsilon\}$
3      while there is $(w, \sigma) \in W_B \times \Sigma$ s.t. $\mu(\sigma w)\boldsymbol{\beta} \notin \langle \mu(u)\boldsymbol{\beta} : u \in W_B \rangle_{\mathbb{Q}}$ do
4          $W_B := W_B \cup \{\sigma w\}$
       // Build new automaton
5      $B := \begin{bmatrix} \mu(w_1)\boldsymbol{\beta} & \dots & \mu(w_m)\boldsymbol{\beta} \end{bmatrix}$ where $W_B = \{w_1, \dots, w_m\}$
       // Conjugate $\mathcal{A}$ with matrix $B$ to obtain $\mathcal{A}'$
6      $\mathcal{A}' := (\boldsymbol{\alpha}', \mu', \boldsymbol{\beta}')$ s.t. $\boldsymbol{\alpha}' = \boldsymbol{\alpha}B$, $B\boldsymbol{\beta}' = \boldsymbol{\beta}$ and $B\mu'(\sigma) = \mu(\sigma)B$, for all $\sigma \in \Sigma$
7      match compute_ℤ_generators($\mathcal{A}'$) with
8          | $w \in \Sigma^*$ ->                                   // $\boldsymbol{\alpha}'\mu'(w) \notin \mathbb{Z}^m$
9              take $i \in \{1, \dots, m\}$ such that $(\boldsymbol{\alpha}'\mu'(w))_i \notin \mathbb{Z}$
10             return $ww_i$
11         | $W \subseteq \Sigma^*$ ->                 // Generators of forward space in $\mathbb{Z}^m$
12             $B_F$ := generate $\mathbb{Z}$-basis of $\langle \boldsymbol{\alpha}'\mu'(w) \mid : w \in W \rangle_{\mathbb{Z}}$  // Using Smith Normal Form
13             $F := \begin{bmatrix} \boldsymbol{v}_1 \\ \dots \\ \boldsymbol{v}_\ell \end{bmatrix}$ where $B_F = \{\boldsymbol{v}_1, \dots, \boldsymbol{v}_\ell\}$
               // Conjugate $\mathcal{A}'$ with matrix $F$ to obtain $\mathcal{A}''$
14             $\mathcal{A}'' := (\boldsymbol{\alpha}'', \mu'', \boldsymbol{\beta}')$ s.t. $\boldsymbol{\alpha}''F = \boldsymbol{\alpha}'$, $\boldsymbol{\beta}'' = F\boldsymbol{\beta}'$ and $\mu''(\sigma)F = F\mu'(\sigma)$, for all $\sigma \in \Sigma$
15             return $\mathcal{A}''$
```

The procedure to compute an equivalent $\mathbb{Z}$-automaton from a $\mathbb{Q}$-automaton is illustrated in Figure 6. It starts by computing a $\mathbb{Q}$-basis of the backward space and by building an equivalent $\mathbb{Q}$-automaton $\mathcal{A}'$, where each entry of a forward reachability vector is an evaluation of the function computed by $\mathcal{A}$, that is $\boldsymbol{\alpha}'\mu'(u) = \begin{bmatrix} [\![\mathcal{A}]\!](uw_1) & \dots & [\![\mathcal{A}]\!](uw_m) \end{bmatrix}$. We then apply compute_ℤ_generators(to either deduce the existence of a word such that $[\![\mathcal{A}]\!](ww_i) \notin \mathbb{Z}$ for some $i \in \{1, \dots, m\}$, or to obtain a generator of the forward reachability set consisting of integer vectors. Form these generators, an equivalent $\mathbb{Z}$-automaton is built.

Theorem 4.2. *The procedure* compute_ℤ_automaton*, described in Figure 6, is a polynomial-time algorithm that given a $\mathbb{Q}$-weighted automaton $\mathcal{A}$ of dimension n over Σ, either outputs an equivalent $\mathbb{Z}$-automaton (that is in fact minimal as a $\mathbb{Q}$-weighted automaton), or a word w such that $[\![\mathcal{A}]\!](w) \notin \mathbb{Z}$.*

Proof. The procedure is a variant of the classical minimisation algorithm for weighted automata over fields.

The first step is to compute a basis $\{u_1, \dots, u_m\}$ of the backward space of $\mathcal{A}$. Lines 2-4 correspond to Tzeng's procedure and, as noted previously, this is done in polynomial time. The matrix $B \in \mathbb{Q}^{n \times m}$ has columns corresponding to the vectors in the above-mentioned basis, that are $\mu(u_i)\boldsymbol{\beta}$.

The next step defines a new m dimensional $\mathbb{Q}$-automaton $\mathcal{A}'$ that is a conjugate of $\mathcal{A}$, so that $[\![\mathcal{A}]\!] = [\![\mathcal{A}']\!]$. From the fact that the columns of B form a basis of the backward space of $\mathcal{A}$ it can be seen that $\mathcal{A}'$ is well-defined. Furthermore, for all $w \in \Sigma^*$ we have $\boldsymbol{\alpha}'\mu'(w) = \boldsymbol{\alpha}\mu(w)B$, so, the i-th entry of $\boldsymbol{\alpha}'\mu'(w)$ has the form $\boldsymbol{\alpha}\mu(ww_i)\boldsymbol{\beta} = [\![\mathcal{A}]\!](ww_i)$. Thus, the forward reachability set of $\mathcal{A}'$ consists exclusively of integer vectors when $[\![\mathcal{A}]\!]$ is integer-valued.

```
def $\mathbb{Q}$_equivalence_oracle($\mathcal{H}$)=
    match compute_$\mathbb{Z}$_automaton($\mathcal{H}$) with
        | $w \in \Sigma^*$ -> return Some($w$)                // A counterexample as $f_{\mathcal{H}}(w) \notin \mathbb{Z}$
        | $\mathcal{H}'$ -> return equivalence_oracle($\mathcal{H}'$)   // $\mathbb{Z}$-automaton equivalent to $\mathcal{H}$
```

Applying Proposition 4.1, the computation of `compute_`$\mathbb{Z}$`_generators(`$\mathcal{A}'$`)` yields either a word $w \in \Sigma^*$ such that $\boldsymbol{\alpha}\mu(w) \notin \mathbb{Z}^m$ or else a set W of words generating the forward reachability set of $\mathcal{A}'$. In the former case, there exists $i \in \{1, \ldots, m\}$ such that $(\boldsymbol{\alpha}\mu(w))_i \notin \mathbb{Z}$ and so $[\![\mathcal{A}]\!](ww_i) \notin \mathbb{Z}$. In the latter case, we use the Smith Normal Form to generate a $\mathbb{Z}$-basis B_F of $\langle \boldsymbol{\alpha}'\mu'(w) : w \in W \rangle_{\mathbb{Z}}$. As B_F is comprised of the $\mathbb{Z}$-vectors $\boldsymbol{v}_1, \ldots, \boldsymbol{v}_\ell$, the ℓ dimensional automaton $\mathcal{A}''$ is a conjugate automaton of $\mathcal{A}'$, that is $[\![\mathcal{A}']\!] = [\![\mathcal{A}'']\!]$. Note that $\mathcal{A}''$ is a well-defined $\mathbb{Z}$-automaton by the fact that the rows of F form a $\mathbb{Z}$-basis forward module of $\mathcal{A}'$, which entails that Equation (2) has a solution $\alpha_f, \mu_f(\sigma), \beta_f$ in integers. We conclude by noting that ℓ is the dimension of the forward space of $\mathcal{A}'$ as well as the rank of the forward module. It follows that $\mathcal{A}''$ is a minimal $\mathbb{Q}$-weighted automaton. □

4.2 Exact Learning

In this subsection, we describe how the exact learning problem for $\mathbb{Z}$-weighted automata can be reduced to the exact learning problem for $\mathbb{Q}$-weighted automata. Such a reduction is non-trivial since the equivalence oracle in the former setting is more restrictive: it requires a $\mathbb{Z}$-weighted automaton as input rather than a $\mathbb{Q}$-weighted automaton. The key to the reduction is thus a procedure $\mathbb{Q}$`_equivalence_oracle` that implements an equivalence oracle for $\mathbb{Q}$-weighted automata using an equivalence oracle for $\mathbb{Z}$-weighted automata. This procedure inputs a $\mathbb{Q}$-weighted automaton $\mathcal{H}$ and returns either Some(w) or None:

- In the first case, it returns Some(w) with w being a counterexample, witnessing that $[\![\mathcal{A}]\!](w) \neq [\![\mathcal{H}]\!](w)$. This counterexample is given by `compute_`$\mathbb{Z}$`_automaton(`$\mathcal{H}$`)` in case $[\![\mathcal{H}]\!]$ is not integer valued and otherwise it is given by `equivalence_oracle`.
- In the second case the procedure returns None, meaning that $\mathcal{H}$ is equivalent to $\mathcal{A}$.

Theorem 4.3. *There is a procedure that learns the target $\mathbb{Z}$-weighted automaton $\mathcal{A}$, by outputting a minimal $\mathbb{Z}$-weighted automaton equivalent to $\mathcal{A}$, which runs in polynomial time in the length of the encoding of $\mathcal{A}$ and in the length of the longest counterexample given by the teacher.*

Proof. Denote by s the size of the encoding of the target automaton $\mathcal{A}$. As is the case for $\mathbb{Q}$-weighted automata learning, the algorithm maintains the invariant that the dimension of the hypothesis automata $\mathcal{H}$ constructed during the learning procedure is less than s. By Theorem 4.2, the procedure `compute_`$\mathbb{Z}$`_automaton(`$\mathcal{H}$`)` runs in time polynomial in s. This implies that the built-in $\mathbb{Q}$`_equivalence_oracle(`$\mathcal{H}$`)` also runs in time polynomial in s. We know that there is a procedure $\mathcal{L}$ that learns $\mathbb{Q}$-weighted automata, and runs in time polynomial in s and in the length of the longest counterexample given by the teacher [Beimel et al. 1999]. As such, $\mathcal{L}$ only calls such equivalence oracle a polynomial number of times. Therefore, using $\mathbb{Q}$`_equivalence_oracle` as an oracle for $\mathcal{L}$ yields a polynomial time procedure that outputs a $\mathbb{Q}$-weighted automaton $\mathcal{H}$ equivalent to $\mathcal{A}$. We conclude by calling `compute_`$\mathbb{Z}$`_automaton(`$\mathcal{H}$`)` which runs, as already mentioned, in time polynomial in s. □

5 P-FINITE AUTOMATA

Recall that a P-finite automaton of dimension n over Σ is a tuple $\mathcal{A} = (\boldsymbol{\alpha}, \mu, \boldsymbol{\beta}(x))$ where $\boldsymbol{\alpha} \in \mathbb{Q}^{1\times n}$ is the initial vector, $\mu : \Sigma \to \mathbb{Q}[x]^{n\times n}$ is the transition function and $\boldsymbol{\beta}(x) \in \mathbb{Q}[x]^{n\times 1}$ is the final vector. We write $\mu(\sigma, k)$ to stand for $\mu(\sigma)(k)$ for all $\sigma \in \Sigma$ and $k \in \mathbb{N}$. We extend μ to a map $\mu : \Sigma^* \to \mathbb{Q}[x]^{n\times n}$ by writing $\mu(\varepsilon)(x) := I_n$ and

$$\mu(w\sigma, x) := \mu(w, x)\, \mu(\sigma, x + |w|)$$

for all $\sigma \in \Sigma$ and $w \in \Sigma^*$. Hence, the semantics of $\mathcal{A}$, defined as

$$[\![\mathcal{A}]\!](w) = \boldsymbol{\alpha}\mu(\sigma_1, 1) \ldots \mu(\sigma_k, k)\boldsymbol{\beta}(k+1)$$

for all $w = \sigma_1 \ldots \sigma_k \in \Sigma^*$, can be simply written $[\![\mathcal{A}]\!](w) = \boldsymbol{\alpha}\mu(w, 1)\boldsymbol{\beta}(|w| + 1)$. The semantics of $\mathcal{A}$ is also called the function computed by $\mathcal{A}$. We also denote by $\boldsymbol{e}_1, \boldsymbol{e}_2, \ldots, \boldsymbol{e}_n$ the standard basis.

In this section, we tackle the zeroness, equivalence, and exact learning problems for P-finite automata. The equivalence problem is the problem of deciding whether two automata compute the same function, while the zeroness problem aims to check whether the input automaton computes the zero function. In Section 5.1 we observe that the zeroness and equivalence problems for P-finite automata are polynomial-time interreducible and we show that zeroness can be solved in polynomial time. Meanwhile, in Section 5.2 we show that the P-finite automata can be exactly learned in polynomial time in the MAT model.

5.1 Equivalence

We can reduce the equivalence problem to the zeroness problem. Indeed, two automata $\mathcal{A}_1$ and $\mathcal{A}_2$ are equivalent if and only if the difference automaton $\mathcal{A}_-$ (such that $[\![\mathcal{A}_-]\!] = [\![\mathcal{A}_1]\!] - [\![\mathcal{A}_2]\!]$) computes the zero function. We refer to Appendix B for details.

Proposition 5.1. *The equivalence problem of P-finite automata is polynomial-time reducible to the zeroness problem.*

5.1.1 Backward module. Below, we fix a P-finite automaton $\mathcal{A} = (\boldsymbol{\alpha}, \mu, \boldsymbol{\beta}(x))$ of dimension n over Σ. The *backward function* associated to $\mathcal{A}$, denoted by $B_{\mathcal{A}}$, is the function $B_{\mathcal{A}} : \Sigma^* \to \mathbb{Q}[x]^n$ given by

symbolic semantics

$$B_{\mathcal{A}}(u)(x) = \mu(u, x)\boldsymbol{\beta}(x + |u|)\,.$$

The *backward module* is the $\mathbb{Q}[x]$-submodule of $\mathbb{Q}[x]^n$ defined as $\mathcal{B}_{\mathcal{A}} = \langle B_{\mathcal{A}}(w) : w \in \Sigma^* \rangle_{\mathbb{Q}[x]}$.

Consider the P-finite automaton of Program 3, one can show that the backward module $\mathcal{B}_{\mathcal{A}_1}$ of this automaton is defined as:

$$\left\langle \begin{bmatrix} 0 \\ x \end{bmatrix}, \begin{bmatrix} 0 \\ xp_k(x) \end{bmatrix}, \begin{bmatrix} xp_k(x) \\ 0 \end{bmatrix} : k \in \mathbb{N} \right\rangle_{\mathbb{Q}[x]},$$

where $p_k(x) := \prod_{i=1}^{k}(x + i)$. By a simple computation, we have that

$$\mathcal{B}_{\mathcal{A}_1} = \left\langle \begin{bmatrix} 0 \\ x \end{bmatrix}, \begin{bmatrix} x(x+1) \\ 0 \end{bmatrix} \right\rangle_{\mathbb{Q}[x]}.$$

We remark that the backward function can be defined recursively as $B_{\mathcal{A}}(\varepsilon) = \boldsymbol{\beta}(x)$, and for all $\sigma \in \Sigma$ and $w \in \Sigma^*$,

$$B_{\mathcal{A}}(\sigma w) = \mu(\sigma, x)\, B_{\mathcal{A}}(w)(x+1),$$

where $B_{\mathcal{A}}(w)(x+1)$ is obtained by substituting $x+1$ for x in the vector $B_{\mathcal{A}}(w)$. By definition, the result of the computation of a P-finite automaton $\mathcal{A}$ on a word w is $[\![\mathcal{A}]\!](w) = \boldsymbol{\alpha}B_{\mathcal{A}}(w)(1)$.

$$\mu(uv)(x) = \mu(u)(x) \cdot \mu(v)(x + |u|)$$

$\mu_a = \left[\;..(x)...\; \right] \sigma$ $\sigma(x) = x+1$ $\mu_{uv} = \mu^{(u)} \cdot \sigma^{|u|} \cdot \mu(v) \sigma^{|u+v|}$

Figure 7: Finding a generating set for the backward module of a P-finite automaton

```
def generators_backward_module($\mathcal{A}$)=
    // $\mathcal{A}$ a P-finite automaton over $\Sigma$
    $W := \{\varepsilon\}$
    // Finding words that increase the rank
    while there is $(w, \sigma) \in W \times \Sigma$ such that $B_{\mathcal{A}}(\sigma w) \notin \langle B_{\mathcal{A}}(u) : u \in W \rangle_{\mathbb{Q}(x)}$ do
        $W := W \cup \{\sigma w\}$
    // Finding words that augment the module
    while there is $(w, \sigma) \in W \times \Sigma$ such that $B_{\mathcal{A}}(\sigma w) \notin \langle B_{\mathcal{A}}(u) : u \in W \rangle_{\mathbb{Q}[x]}$ do
        $W := W \cup \{\sigma w\}$
    return $\{B_{\mathcal{A}}(u) : u \in W\}$
```

From backward module to zeroness. Formally speaking, the zeroness problem asks, given an automaton $\mathcal{A}$ over Σ, whether $[\![\mathcal{A}]\!](w) = 0$ for all words $w \in \Sigma^*$. The following proposition describes how we can decide zeroness by inspecting a finite generating set of the backward module.

Proposition 5.2. *Let $\mathcal{A} = (\boldsymbol{\alpha}, \mu, \boldsymbol{\beta}(x))$ be a P-finite automaton of dimension n. Let $S \subseteq \mathbb{Q}[x]^n$ be a finite generating set for the backward module $\mathcal{B}_{\mathcal{A}}$. We have $[\![\mathcal{A}]\!] \equiv 0$ if and only if all $\boldsymbol{v} \in S$ satisfy $\boldsymbol{\alpha v}(1) = 0$.*

$\langle S \rangle = \mathcal{B}_A$

Proof. Let $\mathcal{A}$ be over Σ. The proof is straightforward by unfolding the definitions of backward function, backward module, and $f_{\mathcal{A}}$:

$$\begin{aligned} \forall w \in \Sigma^*, [\![\mathcal{A}]\!](w) = 0 &\iff \forall w \in \Sigma^* : \boldsymbol{\alpha} B_{\mathcal{A}}(w)(1) = 0 \\ &\iff \forall v \in \mathcal{B}_{\mathcal{A}}, \boldsymbol{\alpha} v(1) = 0 && \text{By the definition of } \mathcal{B}_{\mathcal{A}} \\ &\iff \forall v \in S, \boldsymbol{\alpha} v(1) = 0 && \text{Since } S \text{ is a generating set of } \mathcal{B}_{\mathcal{A}} \end{aligned}$$

□

The previous proposition indicates that, in order to verify zeroness, it is enough to check if $\boldsymbol{\alpha}$ is orthogonal to a generating set of the backward module.

5.1.2 Computing a generating set for $\mathcal{B}_{\mathcal{A}}$. Our algorithm for computing a generating set of the backward module is displayed in Figure 7. It bears a strong resemblance to our algorithm for computing generators for the backward and forward modules in $\mathbb{Z}$-weighted automata (Figure 5). The main distinction lies in the soundness proof, which is more involved due to the necessity to work with $\mathbb{Q}[x]$-modules. In particular, we will need the following corollary of Proposition 3.3.

Corollary 5.3. *Let $n, k \in \mathbb{N}$ and $M_0 \subsetneq M_1 \subsetneq \ldots \subsetneq M_k$ be a strictly increasing chain of submodules of $\mathbb{Q}[x]^n$, all having the same rank $r \leq n$. Assume that M_0 is generated by a collection of vectors whose entries have degree at most d. Then $k \leq d \cdot r$.*

Proof. From Proposition 3.3, it follows that k is bounded by the number of prime factors of $D_r(A)$ where A is the matrix whose columns contain generators of M_0. Since the number of prime factors of a univariate polynomial is at most its degree, k is bounded by $\deg(D_r(A))$. This can also be upper-bounded by the maximum degree of all $r \times r$ minors of A, which, by the triangle inequality and the determinant formula involving permutations, is at most $d \cdot r$. □

We are now ready to present the polynomial-time membership of the equivalence problem of P-finite automata.

Theorem 5.4. *The procedure* `generators_backward_module` *in Figure 7, on an input P-finite automaton $\mathcal{A}$, terminates and outputs a set B of vectors such that $\mathcal{B}_{\mathcal{A}} = \langle B \rangle_{\mathbb{Q}[x]}$. The procedure executes in polynomial time in the length of encoding of $\mathcal{A}$.*

Proof. Let $\mathcal{A} = (\boldsymbol{\alpha}, \mu, \boldsymbol{\beta}(x))$ be an automaton of dimension n and over alphabet Σ. Write $W_1, W_2, \ldots$ for the successive instantiations of the variable W during the execution of the function `generators_backward_module(`$\mathcal{A}$`)`. Since $W_1 = \{\varepsilon\}$, and for all $i > 0$, $W_i = W_{i-1} \cup \{\sigma w\}$ for some $\sigma \in \Sigma$ and $w \in W_{i-1}$, it follows that the maximum length of words in W_i is at most the size of W_i.

The first **while**-loop, in Line 3, terminates after at most n iterations since the backward module, being a submodule of $\mathbb{Q}[x]^n$, has rank at most n. The second **while**-loop, in Line 5, terminates by virtue of $\mathbb{Q}[x]^n$ being Noetherian. Below, we write

$$W_\ell = \{w_1, \ldots, w_\ell\} \qquad \text{and} \qquad W_m = \{w_1, \ldots, w_m\}$$

for some $\ell \leq n$, for the instantiations of W upon exiting the first and second **while**-loops, respectively.

We first claim that $B_{\mathcal{A}}(w) \in \langle B_{\mathcal{A}}(u) : u \in W_\ell \rangle_{\mathbb{Q}(x)}$ for all words $w \in \Sigma^*$. The proof is by induction on the length of the words. The base case ($|w| = 0$) follows as $\varepsilon \in W_\ell$. For the inductive step ($|w| > 0$), decompose w as $\sigma w'$ for some $\sigma \in \Sigma$ and $w' \in \Sigma^*$. By the induction hypothesis,

$$B_{\mathcal{A}}(w')(x) = \sum_{k=1}^{\ell} \frac{p_k(x)}{q_k(x)} B_{\mathcal{A}}(w_k)(x)$$

for some univariate polynomials $p_k(x)$, $q_k(x) \in \mathbb{Q}[x]$, where $k \in \{1, \ldots, \ell\}$. Recall the recursive definition of the backward function, namely, we have $B_{\mathcal{A}}(\sigma w') = \mu(\sigma, x)\, B_{\mathcal{A}}(w')(x+1)$. Hence,

$$\begin{aligned} B_{\mathcal{A}}(\sigma w')(x) =& \mu(\sigma, x) \sum_{k=1}^{\ell} \frac{p_k(x+1)}{q_k(x+1)}\, B_{\mathcal{A}}(w_k)(x+1) \\ =& \sum_{k=1}^{\ell} \frac{p_k(x+1)}{q_k(x+1)} \mu(\sigma, x) B_{\mathcal{A}}(w_k)(x+1) \\ =& \sum_{k=1}^{\ell} \frac{p_k(x+1)}{q_k(x+1)} B_{\mathcal{A}}(\sigma w_k), \end{aligned}$$

implying that $B_{\mathcal{A}}(w) \in \langle B_{\mathcal{A}}(\sigma u) : \sigma \in \Sigma, u \in W_\ell \rangle_{\mathbb{Q}(x)}$. But then the exit-condition of the first **while**-loop ensures that

$$\langle B_{\mathcal{A}}(\sigma u) : \sigma \in \Sigma, u \in W_\ell \rangle_{\mathbb{Q}(x)} \subseteq \langle B_{\mathcal{A}}(u) : u \in W_\ell \rangle_{\mathbb{Q}(x)},$$

concluding the proof of the claim.

We show a similar result concerning the second **while**-loop termination. We claim that $B_{\mathcal{A}}(w) \in \langle B_{\mathcal{A}}(u) : u \in W_m \rangle_{\mathbb{Q}[x]}$ for all words $w \in \Sigma^*$. Intuitively speaking, once exiting the second **while**-loop, no words in Σ^* that could augment the module can be added. The proof is again by induction on the length of the words. The base case ($|w| = 0$) trivially holds as $\varepsilon \in W_m$. For the inductive step ($|w| > 0$), rewrite w as $\sigma w'$ for some $\sigma \in \Sigma$ and $w' \in \Sigma^*$. By the induction hypothesis,

$$B_{\mathcal{A}}(w')(x) = \sum_{k=1}^{m} p_k(x) B_{\mathcal{A}}(w_k)(x)$$

for some polynomials $p_k(x) \in \mathbb{Q}[x]$ where $k \in \{1, \ldots, m\}$. Following similar reasoning as in the first loop case, we obtain that $B_{\mathcal{A}}(w) \in \langle B_{\mathcal{A}}(\sigma u) : \sigma \in \Sigma, u \in W_m \rangle_{\mathbb{Q}[x]}$. But then, again, the exit-condition of the second **while**-loop ensures that

$$\langle B_{\mathcal{A}}(\sigma u) : \sigma \in \Sigma, u \in W_m \rangle_{\mathbb{Q}[x]} \subseteq \langle B_{\mathcal{A}}(u) : u \in W_m \rangle_{\mathbb{Q}[x]},$$

concluding the proof of the claim.

It remains to show that the execution of `generators_backward_module(`$\mathcal{A}$`)` can be carried out in time polynomial in the length of encoding of $\mathcal{A}$. Recall that, given a word $w = \sigma_1 \ldots \sigma_k$, the backward reachable vector is computed as $B_{\mathcal{A}}(w)(x) = \mu(\sigma_1, x) \ldots \mu(\sigma_k, x + k - 1)\boldsymbol{\beta}(x + k)$. Denote by d and c, respectively, the maximal degree and largest coefficient of the polynomials occurring as entries of $\boldsymbol{\beta}(x)$ and $\mu(\sigma)$, for $\sigma \in \Sigma$. It follows that the degree of polynomial entries of $B_{\mathcal{A}}(w)(x)$ is at most $d(|w| + 1)$. We will argue that the largest coefficient of the polynomials occurring as entries of $B_{\mathcal{A}}(w)(x)$ is at most $n^{|w|}c^{|w|+1}(|w|d)^{(|w|+1)d}$. Indeed, this comes from the observation that the coefficients of the monomial $(x + |w|)^d$ are bounded by $(|w|d)^d$. The length of the encoding of $B_{\mathcal{A}}(w)(x)$ is therefore polynomial in the length of encoding of $\mathcal{A}$ and in $|w|$. Using [Kannan 1985], we deduce that testing whether

$$B_{\mathcal{A}}(\sigma w) \notin \langle B_{\mathcal{A}}(u) : u \in W \rangle_{\mathbb{Q}(x)} \qquad \text{or} \qquad B_{\mathcal{A}}(\sigma w) \notin \langle B_{\mathcal{A}}(u) : u \in W \rangle_{\mathbb{Q}[x]}$$

is polynomial in the length of encoding of $\mathcal{A}$, and in the maximum length of words in W_m, and in the size of W_m. Recall that the maximum length of words in W_m is at most the size of W_m.

We conclude the proof by arguing that the size of W_m is polynomial in the length of encoding of $\mathcal{A}$. As a result of the two claims on the termination of the loops, the two backward modules induced by W_ℓ and W_m have the same rank. Since $\ell \leq n$, the degree of the polynomials in the entries of $B_{\mathcal{A}}(u)$ for $u \in W_\ell$ is at most $d(n+1)$. By Corollary 5.3, the length of the strictly increasing sequence of modules induced by $W_\ell \subsetneq \ldots \subsetneq W_m$ is at most $m - \ell + 1 \leq nd(n + 1)$, implying that the size of W_m is at most $nd(n + 1) + n - 1$. □

By a direct application of Theorem 5.4, Proposition 5.2 and Proposition 5.1, we have:

Theorem 5.5. *The zeroness and equivalence problems for P-finite automata are both in polynomial time. We can furthermore suppose that the polynomial-time procedure for testing equivalence returns a word of polynomial length that witnesses in-equivalence on negative instances.*

5.2 Learning

We first introduce some notation and terminology. Below, we fix $f : \Sigma^* \longrightarrow \mathbb{Q}$ to be a function. The Hankel matrix of f is an infinite matrix with rows and columns indexed by words in Σ^* such that $H(r, c) := f(rc)$, where $H(r, c)$ is the entry of matrix with row index $r \in \Sigma^*$ and column index $c \in \Sigma^*$.

Given two sequences $\mathcal{R}, C$ of words from Σ^*, denote by $H(\mathcal{R}, C)$ the restriction of the Hankel matrix to the respective sets $\mathcal{R}$ of rows and C of columns, that is, if $\mathcal{R} = [r_1, \ldots, r_m]$ and $C = [c_1, \ldots, c_n]$, then $H(\mathcal{R}, C)$ is the $m \times n$ submatrix such that $H(\mathcal{R}, C)_{i,j} = f(r_i c_j)$. Moreover, given two words $r, c \in \Sigma^*$ such that r appears in the sequence $\mathcal{R}$ and c appears in the sequence C, we write $\text{row}_C(r)$ for the associated row and $\text{col}_{\mathcal{R}}(c)$ for the associated column in $H(\mathcal{R}, C)$, namely,

$$\text{row}_C(r) := \begin{bmatrix} f(rc_1) & \ldots & f(rc_n) \end{bmatrix} \qquad \text{and} \qquad \text{col}_{\mathcal{R}}(c) := \begin{bmatrix} f(r_1 c) & \ldots & f(r_m c) \end{bmatrix}^\top.$$

In the sequel, we will call $H(\mathcal{R}, C)$ a *table*.

Assume that the target function f can be computed by a P-finite automaton. Intuitively, our learning algorithm maintains a table from which it constructs a *hypothesis automaton*. Using the equivalence oracle, the algorithm checks whether the hypothesis automaton computes the function f. In case of a negative answer, the witness of non-equivalence is used to augment the table (by augmenting the sets of rows and columns), and the process repeats.

In order to build the hypothesis automaton we require the table to be *closed* in the following sense. Let $\mathcal{R}$ and C be two sequences of words from Σ^* such that $|C| = n$. We say that the table $H(\mathcal{R}, C)$ is *closed* when for each $\sigma \in \Sigma$, there exists a matrix of polynomials $M_\sigma(x) \in \mathbb{Q}[x]^{n\times n}$ such that for all rows $r \in \mathcal{R}$, the equation $\mathrm{row}_C(r\sigma) = \mathrm{row}_C(r)M_\sigma(|r|+1)$ holds. Given such a closed table $H(\mathcal{R}, C)$, we can compute a hypothesis P-finite automaton $(\boldsymbol{\alpha}, \mu, \boldsymbol{\beta})$ of dimension n as follows:

$$\boldsymbol{\alpha} = \mathrm{row}_C(r_1), \qquad \boldsymbol{\beta}(x) = \boldsymbol{e}_1, \qquad \mu(\sigma, x) = M_\sigma(x) \text{ for all } \sigma \in \Sigma. \tag{3}$$

The polynomials in the transition matrix of a hypothesis automaton need to be constructed by interpolation. To this end, we maintain a variable d that represents a degree bound on the polynomials in $M_\sigma(x)$. Specifically, we will say that the table $H(\mathcal{R}, C)$ is *d-closed* when the maximal degree of the polynomials in the $M_\sigma(x)$ are bounded by d. The d-closedness condition allows to set up a linear system of equations where the unknowns $y_{i,j,k}$ are the coefficients of the polynomials of each entry of $M_\sigma(x)$, that is, we write the (i, j)-th entry of $M_\sigma(x)$ as $y_{i,j,d}x^d + y_{i,j,d-1}x^{d-1} + \cdots + y_{i,j,0}$. More precisely, we search for the unknowns $y_{i,j,k}$, ranging over $\mathbb{Q}$. Focusing on the j-th column of $M_\sigma(x)$, the d-closedness condition $\mathrm{row}_C(r\sigma) = \mathrm{row}_C(r)M_\sigma(|r|+1)$ entails, for all $r \in \mathcal{R}$, the following equation:

$$f(r\sigma c_j) = \mathrm{row}_C(r)\sum_{k=0}^{d}(|r|+1)^k Y_k = \sum_{k=0}^{d}(|r|+1)^k \mathrm{row}_C(r) Y_k, \tag{4}$$

where the Y_k are the column vectors $\begin{bmatrix} y_{1,j,k} & \dots & y_{|\mathcal{R}|,j,k}\end{bmatrix}^\top$ of unknowns. By taking $H = H(\mathcal{R}, C)$ and Δ the $m \times m$ diagonal matrix $diag(|r_1|+1, \dots, |r_m|+1)$ where $\mathcal{R} = [r_1, \dots, r_m]$, we obtain the following system of equations in $Y_0, \dots, Y_d$:

$$\begin{bmatrix} f(r_1\sigma c_j) \\ \vdots \\ f(r_m\sigma c_j)\end{bmatrix} = \sum_{k=0}^{d}\Delta^k H Y_k = \begin{bmatrix}\Delta^0 H & \dots & \Delta^d H\end{bmatrix}\begin{bmatrix} Y_0 \\ \vdots \\ Y_d\end{bmatrix}. \tag{5}$$

We recover the *hypothesis automaton associated to the d-closed table $H(\mathcal{R}, C)$* from a solution to the above system of equations by setting the j-th column of $M_\sigma(x)$ to be $\sum_{k=0}^{d} Y_k x^k$. Henceforth we denote by $A_d(\mathcal{R}, C)$ the matrix

$$\begin{bmatrix}\Delta^0 H & \dots & \Delta^d H\end{bmatrix}.$$

In the following proposition, we state a sufficient condition for the above linear system of equations to have a solution, meaning that *the table $H(\mathcal{R}, C)$ is d-closed.*

Proposition 5.6. *Given two sequences of words $\mathcal{R}$ and C and $d \in \mathbb{N}$, the table $H(\mathcal{R}, C)$ is d-closed if $A_d(\mathcal{R}, C)$ has full row rank.*

Proof. Let $\mathcal{R} = [r_1, \dots, r_m]$ and $C = [c_1, \dots, c_n]$. Write A for $A_d(\mathcal{R}, C) \in \mathbb{Q}^{m\times(d+1)n}$, which, by hypothesis, has full row rank. Then for all vectors $V \in \mathbb{Q}^{m\times 1}$, the system $AX = V$ has a solution $X \in \mathbb{Q}^{(d+1)n\times 1}$. Indeed, the system $AX = V$ has a solution if and only if $\mathrm{rank}(A) = \mathrm{rank}(\begin{bmatrix} A & V\end{bmatrix})$. Since $A \in \mathbb{Q}^{m\times(d+1)n}$ and $\mathrm{rank}(A) = m$, we deduce that for all $V \in \mathbb{Q}^{m\times 1}$, the equality $\mathrm{rank}(\begin{bmatrix} A & V\end{bmatrix}) = \mathrm{rank}(A)$ holds and the system $AX = V$ has a solution.

Write n for the size of the sequence C. We construct the matrices $M_\sigma \in \mathbb{Q}[x]^{n\times n}$, for $\sigma \in \Sigma$, as follows. By the above argument, for $j \in \{1, \cdots, n\}$, the system of linear equations described in (5) has some solution, say $Y_0^*, \dots, Y_d^*$. We define the j-th column of $M_\sigma(x)$ to be $\sum_{k=0}^{d} Y_k^* x^k$, which in turn implies that the (i, j)-th entry of $M_\sigma(x)$ is the polynomial $y^*_{i,j,d}x^d + y^*_{i,j,d-1}x^{d-1} + \cdots + y^*_{i,j,0}$ of degree d.

Figure 8: Building an associated P-finite automaton to $H(\mathcal{R}, C)$

```
def build_automata($d$,$\mathcal{R}$, $C = [c_1, \dots, c_n]$)=
    // $H(\mathcal{R}, C)$ is assumed to be $d$-closed
    for $\sigma \in \Sigma$ do
        $C' := [\sigma c_1, \dots, \sigma c_n]$
        solve $A_d(\mathcal{R}, C)Y = H(\mathcal{R}, C')$    // Has a solution since $H(\mathcal{R}, C)$ is $d$-closed
        define $\mu(\sigma)_{i,j}(x) := \sum_{k=0}^{d} Y_{i+kn,j}\, x^k$ for all $i \in \{1, \dots, n\}$, $j \in \{1, \dots, n\}$
    return $(\boldsymbol{e}_1{}^\top H, \mu, \boldsymbol{e}_1)$
```

It remains to argue that the matrix $M_\sigma(x)$ so defined satisfies the closedness condition, that is, for all rows $r \in \mathcal{R}$ the condition $\mathrm{row}_C(r\sigma) = \mathrm{row}_C(r) M_\sigma(|r|+1)$ holds. But then, this is guaranteed by enforcing (4) for all columns $c \in C$. We conclude by noting that constraints (4) constitute the system of linear equations described in (5). □

Our algorithm for building the automaton associated to the d-closed table $H(\mathcal{R}, C)$ is given as function `build_automata` in Figure 8. This function is an implementation of the construction stated in (3), which is ensured by the d-closedness assumption on the input table. We note again that the maximal degree of polynomials in constructed automaton is at most d. In summary, we have:

Corollary 5.7. *The function* `build_automata(`d,$\mathcal{R}$, C`)`*, assuming that* $H(\mathcal{R}, C)$ *is* d*-closed, outputs an automaton* $\mathcal{H}$ *associated to the* d*-closed table* $H(\mathcal{R}, C)$*.*

Concretely, in the function `build_automata` computing $A_d(\mathcal{R}, C)$ and $H(\mathcal{R}, C)$ can be evaluated by asking membership queries from the teacher, through `membership_oracle`, at most $|\mathcal{R}| \times |C|$ times. Therefore, the execution of this function runs in time polynomial in $d + |\mathcal{R}| + |C| + |\Sigma|$.

5.2.1 *Correctness of P-finite automata.* Let $\mathcal{R} = [r_1, \dots, r_m]$ and $C = [c_1, \dots, c_n]$ be two sequences of words from Σ^*. Assume that $H(\mathcal{R}, C)$ is closed and let $\mathcal{H} = (\boldsymbol{\alpha}, \mu, \boldsymbol{\beta})$ be an associated P-finite automaton over Σ. We say that that $\mathcal{H}$ *is* correct *on the word* $w \in \Sigma^*$ if $\boldsymbol{\alpha}\mu(w, 1) = \mathrm{row}_C(w)$.

As previously mentioned, after building a hypothesis automaton $\mathcal{H}$ associated with a table, we will ask the teacher an equivalence query on $\mathcal{H}$, through `equivalence_oracle(`$\mathcal{H}$`)`, and receive a counterexample w in case $\mathcal{H}$ is not equivalent to the target automaton. The automaton $\mathcal{H}$ is correct on ε by construction and is necessarily incorrect on w, as indeed we initialize C with ε and ensure that the automaton is always correct on this word, and the fact that $f(w) \neq [\![\mathcal{H}]\!](\varepsilon \cdot w) = \boldsymbol{\alpha}\mu(w, 1)\boldsymbol{e}_1$. We compute the longest prefix $u\sigma$ of w such that $\mathcal{H}$ is correct on u but incorrect on $u\sigma$. Our learning algorithm extends its table by adding the row associated with u.

Computing such a prefix u can be straightforwardly done as depicted in Figure 9. The function `largest_correct_prefix(`$\mathcal{H}$,C, w`)` outputs u, σ as well as the word $c_j \in C$ that renders $\mathcal{H}$ incorrect on $u\sigma$. The execution of `largest_correct_prefix(`$\mathcal{H}$,C, w`)` runs in time polynomial in its parameters, that is, in time polynomial in $|w| + |C|$.

In Corollary 5.7, we assumed that the table $H(\mathcal{R}, C)$ is closed in order to build the hypothesis automaton from it. However, when augmenting the table with the row associated with u, the closedness condition might not hold anymore as the new row u might be linearly dependent with the previous rows in $\mathcal{R}$. We show in the next proposition that in such cases the closedness of the table can be restored by adding the column associated with σc_j to the table, where (u, σ, c_j) is the output of the function `largest_correct_prefix(`$\mathcal{H}$,C, w`)`.

Figure 9: Largest correct prefix

```
def largest_correct_prefix($\mathcal{H}$,$C = [c_1, \dots, c_n], w$)=
    // $\mathcal{H} = (\boldsymbol{\alpha}, \mu, \boldsymbol{\beta}(x))$ is a P-finite automaton of dimension $n$ over $\Sigma$
    $\boldsymbol{x} := \boldsymbol{\alpha}, u := \varepsilon, v := w$
    while $v = \sigma w'$ do
        $\boldsymbol{x} := \boldsymbol{x}\mu(\sigma, |u| + 1)$
        for $j = 1 \dots n$ do
            $y :=$ membership_oracle($u\sigma c_j$)
            if $\boldsymbol{x}_j \neq y$ then return $(u, \sigma, c_j)$
        $u := u\sigma, v := w'$
```

Proposition 5.8. *Let $d \in \mathbb{N}$, and $\mathcal{R}, C$ be sequences of words such that $A_d(\mathcal{R}, C)$ has full row rank. Let $\mathcal{H} = (\boldsymbol{\alpha}, \mu, \boldsymbol{\beta})$ be an automaton associated to the d-closed table $H(\mathcal{R}, C)$. Let $u \in \Sigma^*$ and $\sigma \in \Sigma$ be such that $\mathcal{H}$ is correct on u but not on $u\sigma$. Let c_j be the j-th word in C where $f(u\sigma c_j) \neq \boldsymbol{\alpha}\mu(u\sigma, 1)\boldsymbol{e}_j$. Define $\mathcal{R}' := \mathcal{R} \cdot [u]$. Then the matrix $\begin{bmatrix} A_d(\mathcal{R}', C) & \mathrm{col}_{\mathcal{R}'}(\sigma c_j) \end{bmatrix}$ is full row rank.*

Proof. Write M for the matrix $\begin{bmatrix} A_d(\mathcal{R}', C) & \mathrm{col}_{\mathcal{R}'}(\sigma c_j) \end{bmatrix}$. Since $A_d(\mathcal{R}, C)$ is a sub-matrix of M and M has $|\mathcal{R}| + 1$ rows, we deduce that $|\mathcal{R}| + 1 \geq \mathrm{rank}(M) \geq \mathrm{rank}(A_d(\mathcal{R}, C)) = |\mathcal{R}|$. For a contradiction, assume that M is not full row rank, implying that $\mathrm{rank}(M) = |\mathcal{R}|$. Then the last row of M is a linear combination of all other rows of M. In other words, writing $\mathcal{R}$ as the sequence $[r_1, \dots, r_m]$, there exists a row vector $\mathbf{x}$ such that for all words $c \in C$, for all $k \in \{0, \dots, d\}$,

$$\begin{cases} \mathbf{x} \begin{bmatrix} (|r_1| + 1)^k f(r_1 c) & \dots & (|r_m| + 1)^k f(r_m c) \end{bmatrix}^\top = (|u| + 1)^k f(uc) \\ \mathbf{x} \begin{bmatrix} f(r_1 \sigma c_j) & \dots & f(r_m \sigma c_j) \end{bmatrix}^\top = f(u\sigma c_j) \end{cases}.$$

By Proposition 5.6, since $A_d(\mathcal{R}, C)$ has full row rank, the table $H(\mathcal{R}, C)$ is d-closed, implying that for each row $i \in \{1, \dots, m\}$, the equality $\mathrm{row}_C(r_i\sigma) = \mathrm{row}_C(r_i)\mu(\sigma, |r_i| + 1)$ holds.

Recall that the $\mu(\sigma)$ are matrices of univariate polynomials. Define $\mu_\sigma^{(k)}$ to be the matrix whose (i, j)-th entry is the coefficient of x^k in the (i, j)-th entry of $\mu(\sigma)$. We obtain that $\mathrm{row}_C(r_i\sigma) = \mathrm{row}_C(r_i) \sum_{k=0}^{d} (|r_i| + 1)^k \mu_\sigma^{(k)}$. Therefore,

$$\begin{bmatrix} f(r_1 \sigma c_j) \\ \dots \\ f(r_m \sigma c_j) \end{bmatrix} = \sum_{k=0}^{d} \begin{bmatrix} (|r_1| + 1)^k f(r_1 c_1) & \dots & (|r_1| + 1)^k f(r_1 c_n) \\ \dots & \dots & \dots \\ (|r_m| + 1)^k f(r_m c_1) & \dots & (|r_m| + 1)^k f(r_m c_n) \end{bmatrix} \mu_\sigma^{(k)} \boldsymbol{e}_j.$$

Multiplying both sides of the equation by $\mathbf{x}$, we obtain:

$$f(u\sigma c_j) = \sum_{k=0}^{d} \begin{bmatrix} (|u| + 1)^k f(uc_1) & \dots & (|u| + 1)^k f(uc_n) \end{bmatrix} \mu_\sigma^{(k)} \boldsymbol{e}_j\,,$$

which in turn implies that

$$f(u\sigma c_j) = \mathrm{row}_C(u) \sum_{k=0}^{d} (|u| + 1)^k \mu_\sigma^{(k)} \boldsymbol{e}_j = \mathrm{row}_C(u)\mu(\sigma, |u| + 1)\boldsymbol{e}_j.$$

By hypothesis, the automaton $\mathcal{H}$ is correct on u, meaning that $\boldsymbol{\alpha}\mu(u, 1) = \mathrm{row}_C(u)$. Subsequently,

$$f(u\sigma c_j) = \boldsymbol{\alpha}\mu(u, 1)\mu(\sigma, |u| + 1)\boldsymbol{e}_j = \boldsymbol{\alpha}\mu(u\sigma, 1)\boldsymbol{e}_j.$$

This is in contradiction with the assumption $\boldsymbol{\alpha}\mu(u\sigma, 1)\boldsymbol{e}_j \neq f(u\sigma c_j)$, concluding the proof. □

Corollary 5.9. *Let $d \in \mathbb{N}$, and $\mathcal{R}, C$ be sequences of words such that $A_d(\mathcal{R}, C)$ is full row rank. Let $\mathcal{H} = (\boldsymbol{\alpha}, \mu, \boldsymbol{\beta})$ be an automaton associated to the d-closed table $H(\mathcal{R}, C)$. Let $u \in \Sigma^*$ and $\sigma \in \Sigma$ be such that $\mathcal{H}$ is correct on u but not on $u\sigma$. Let c_j be the j-th word in C where $f(u\sigma c_j) \neq \boldsymbol{\alpha}\mu(u\sigma, 1)\boldsymbol{e}_j$.*

Define $\mathcal{R}' := \mathcal{R} \cdot [u]$ and $C' := C \cdot [\sigma c_j]$. For all $d' > d$, the table $A_{d'}(\mathcal{R}', C')$ has full row rank.

Proof. The result follows from Proposition 5.8 and the fact that $\begin{bmatrix} A_d(\mathcal{R}', C) & \mathrm{col}_{\mathcal{R}'}(\sigma c_j) \end{bmatrix}$ is a submatrix of $A_{d'}(\mathcal{R}', C \cdot [\sigma c_j])$. □

We can combine the above-mentioned functions to define our *partial learner*, which is depicted in Figure 10. It takes four arguments: two sequences of words $\mathcal{R}$, C that determine the table, an integer d_{max} representing our *guess* of the maximal degree of polynomials occurring in the target automaton, and finally a *timeout* integer ℓ on the number of columns $|C|$. This limit ℓ acts as a safeguard in case our guess d_{max} is incorrect.

By construction, when partial_learner($d_{max}, \ell, \mathcal{R}, C$) returns Some($\mathcal{H}$), the automaton $\mathcal{H}$ computes the target function f, that is $[\![\mathcal{H}]\!] = f$. However, the function may return None if we fail to find an equivalent automaton within the time bound imposed by ℓ. In the next section, we will show that if we take ℓ large enough and if we have guessed correctly the value d_{max} for the maximal degree of polynomials occurring in the target automaton, then partial_learner($d_{max}, \ell, [\varepsilon], [\varepsilon]$) will always eventually learn the target.

5.2.2 *Bounding the number of columns.*

The function partial_learner incorporates a limit ℓ on the number of columns added while constructing the table. Having this limit ensures that our algorithm terminates even when the guess of the maximal degree of polynomials d_{max} in the target automaton is incorrect. However, we need also to guarantee that we never exceed the limit when the guess for d_{max} is correct. We can compute such a limit by relating the columns $c_1, \ldots, c_n$ that are added during the execution of partial_learner with the submodule of $\mathcal{B}_{\mathcal{A}}$ generated by

$$B_{\mathcal{A}}(c_1), \ldots, B_{\mathcal{A}}(c_n).$$

Intuitively, the following proposition shows that the sequence of modules $\mathcal{M}_1, \ldots, \mathcal{M}_n$ defined as $\mathcal{M}_i = \langle B_{\mathcal{A}}(c_j) : j \leq i \rangle_{\mathbb{Q}[x]}$, for all $i \in \{1 \ldots n\}$, is strictly increasing, that is, $\mathcal{M}_1 \subsetneq \ldots \subsetneq \mathcal{M}_n$.

We say that a sequence $[w_1, \ldots, w_n]$ of words is *totally suffix-closed* if for each w_i all its suffixes s occur before w_i in the sequence, meaning that $s = w_j$ for some $j \leq i$.

Proposition 5.10. *Let $\mathcal{R}, C$ be sequences of words from Σ^*, such that C is totally suffix-closed. Let d_{max} be the maximal degree of polynomials in the target P-finite automaton $\mathcal{A}$. Let $d \geq d_{max}(|C| + 1)|C|$, let $c \in C$ and let $\sigma \in \Sigma$. Assume that the matrix $A_d(\mathcal{R}, C)$ does not have full row rank, but the matrix $\begin{bmatrix} A_d(\mathcal{R}, C) & \mathrm{col}_{\mathcal{R}}(\sigma c) \end{bmatrix}$ has full row rank. Then*

$$B_{\mathcal{A}}(\sigma c) \notin \langle B_{\mathcal{A}}(c') : c' \in C \rangle_{\mathbb{Q}[x]}.$$

Proof. Write $[r_1, \ldots, r_m]$ for the sequence $\mathcal{R}$ and $[c_1, \ldots, c_n]$ for the sequence C. Towards a contradiction, assume that $B_{\mathcal{A}}(\sigma c) \in \langle B_{\mathcal{A}}(c_1), \ldots, B_{\mathcal{A}}(c_n) \rangle_{\mathbb{Q}[x]}$. Then there exist polynomials $p_1, \ldots, p_n \in \mathbb{Q}[x]$ such that $B_{\mathcal{A}}(\sigma c) = \sum_{i=1}^{n} p_i(x) B_{\mathcal{A}}(c_i)$ and hence the equation

$$\begin{bmatrix} B_{\mathcal{A}}(c_1) & \ldots & B_{\mathcal{A}}(c_n) \end{bmatrix} X = B_{\mathcal{A}}(\sigma c) \tag{6}$$

has solution $X = \begin{bmatrix} p_1(x) & \ldots & p_n(x) \end{bmatrix}^\top$.

A simple induction on the length of the words w gives that the degree of the polynomials in $B_{\mathcal{A}}(w)$ is at most $d_{max}(|w| + 1)$. Since C is totally suffix-closed, we deduce that $c_1 = \varepsilon$, as well as $\max(|c_1|, \ldots, |c_n|, |\sigma c|) \leq n$. These two above facts imply that the maximal degree of polynomials in $B_{\mathcal{A}}(c_1), \ldots, B_{\mathcal{A}}(c_n), B_{\mathcal{A}}(\sigma c)$ is at most $d_{max}(n + 1)$. By [Kannan 1985, Lemma 2.5], we can assume that the maximum degree of polynomials $p_i(x)$ in the solution X of Equation (6) is at most

Figure 10: The partial learner

```
def partial_learner($d_{max}, \ell, \mathcal{R}, C$)=
  if $\ell < |C|$ then return None
  else
    $d := d_{max}(|C|+1)|C|$
    $\mathcal{H}$ := build_automata($d, \mathcal{R}, C$)
    match equivalence_oracle($\mathcal{H}$) with
      | None -> return Some($\mathcal{H}$)                // Found equivalent automaton
      | Some($w$) ->                        // A counterexample $w$ has been found
        $(u, \sigma, c)$ := largest_correct_prefix($\mathcal{H}, w$)
        if rank($A_d(\mathcal{R} \cdot [u], C)$) = $|\mathcal{R}|+1$ then partial_learner($d_{max}, \ell, \mathcal{R} \cdot [u], C$)
        else partial_learner($d_{max}, \ell, \mathcal{R} \cdot [u], C \cdot [\sigma c]$)
```

n times the maximum degree of the polynomials in $B_{\mathcal{A}}(c_1), \ldots, B_{\mathcal{A}}(c_n), B_{\mathcal{A}}(\sigma c)$. Subsequently, the maximum degree of polynomials $p_i(x)$ in X is at most $d_{max}(n+1)n$.

Let $(\boldsymbol{\alpha}, \mu, \boldsymbol{\beta})$ be the target automaton $\mathcal{A}$, and let $f := [\![A]\!]$. Since $B_{\mathcal{A}}(\sigma c) = \sum_{i=1}^{n} p_i(x) B_{\mathcal{A}}(c_i)$ holds, for all words $r \in \mathcal{R}$ we have that

$$\boldsymbol{\alpha}\mu(r,1)B_{\mathcal{A}}(\sigma c) = \sum_{i=1}^{n} p_i(|r|+1)\, \boldsymbol{\alpha}\mu(r,1)B_{\mathcal{A}}(c_i)\,,$$

which, in turn, by the definition of the backward function gives

$$f(r\sigma c) = \sum_{i=1}^{n} p_i(|r|+1) f(rc_i)\,. \tag{7}$$

Since $d \geq d_{max}(n+1)n$, we write the polynomials p_i in X as $p_i(x) := p_i^{(0)} + p_i^{(1)}x + \ldots + p_i^{(d)}x^d$ where $p_i^{(k)}$ the coefficient of monomial x^k in $p_i(x)$. Substituting this representation into (7), we get

$$f(r\sigma c) = \sum_{i=1}^{n} \sum_{k=0}^{d} p_i^{(k)}\,(|r|+1)^k f(rc_i). \tag{8}$$

We group the coefficient $p_i^{(k)}$ of the monomial x^k in all polynomials $p_i(x)$ in a single vector $P^{(k)}$; formally, define d vectors $P^{(0)}, \ldots, P^{(d)}$ such that $P^{(k)} = \begin{bmatrix} p_1^{(k)} & \ldots & p_n^{(k)} \end{bmatrix}^\top$ for $k \in \{1, \ldots, d\}$. From (8) we obtain that:

$$\mathrm{col}_{\mathcal{R}}(\sigma c) = \begin{bmatrix} f(r_1\sigma c) \\ \vdots \\ f(r_m\sigma c) \end{bmatrix} = \sum_{k=0}^{d} \Delta^k H P^{(k)} = A_d(\mathcal{R}, C) \begin{bmatrix} P^0 \\ \vdots \\ P^d \end{bmatrix},$$

where $H = H(\mathcal{R}, C)$ and Δ is the $m \times m$ diagonal matrix $diag(|r_1|+1, \ldots, |r_m|+1)$. We deduce that the rank of $A_d(\mathcal{R}, C)$ is equal to the rank of $\begin{bmatrix} A_d(\mathcal{R}, C) & \mathrm{col}_{\mathcal{R}}(\sigma c) \end{bmatrix}$. This is in contradiction with the assumption that $A_d(\mathcal{R}, C)$ is not full row rank, but $\begin{bmatrix} A_d(\mathcal{R}, C) & \mathrm{col}_{\mathcal{R}}(\sigma c) \end{bmatrix}$ is, concluding the proof. □

Using Corollary 5.3, we can compute an upper bound on the maximum size of increasing submodule of the backward module $\mathcal{B}_{\mathcal{A}}$.

Proposition 5.11. *Let $\mathcal{A}$ be the target automaton of dimension n and with d_{max} the maximal degree of its polynomials. Define $L(y_1, y_2) := \left((y_1+1)y_2^2\right)^{y_2}$.*

Let $n' \geq n$ and $d \geq d_{max}$. Then `partial_learner`$(d, L(d,n'), [\varepsilon], [\varepsilon]) \neq$ None.

Proof. Consider a totally suffix-closed sequence $C = [c_1, \ldots, c_m]$ of words. A simple induction on the length of the words w gives that the degree of the polynomials in $B_{\mathcal{A}}(w)$ is at most $d_{max}(|w|+1)$. Since C is totally suffix-closed, we deduce that $c_1 = \varepsilon$, as well as $\max(|c_1|, \ldots, |c_m|) < m$. These two above facts imply that the maximum degree of polynomials in $B_{\mathcal{A}}(c_1)$, ..., $B_{\mathcal{A}}(c_m)$ is at most $d_{max}m$.

Recall that, by Corollary 5.3, every strictly increasing sequence $\mathcal{M}_0 \subsetneq \ldots \subsetneq \mathcal{M}_k$ of submodules of $\mathbb{Q}[x]^n$ with the same rank r has length $k \leq d \cdot r$, where d is the maximal degree of polynomials of the vectors generating $\mathcal{M}_0$.

For all $i \in \{1, \ldots, m\}$, define $\mathcal{M}_i := \left\langle B_{\mathcal{A}}(c_j) : j \leq i \right\rangle_{\mathbb{Q}[x]}$. The ranks of the submodules $\mathcal{M}_i$ are at most n. We aim at upper bounding m by $L(d_{\max}, n)$; due to Corollary 5.3, the worst upper bound is reached when some module in the strictly increasing sequence of modules reaches full rank. Below we assume that that our increasing sequence reaches full rank, meaning that $\operatorname{rank}(\mathcal{M}_m) = n$. Define $i_1, \ldots, i_n$ as the indices corresponding to when the rank of the submodules $\mathcal{M}_i$ has strictly increased. Formally, we have that $i_1 = 1$ and $i_1 < \ldots < i_n$. Furthermore, for all $j \in \{2, \ldots, n\}$,

$$\operatorname{rank}(\mathcal{M}_{i_{j-1}}) = \operatorname{rank}(\mathcal{M}_{i_j - 1}) < \operatorname{rank}(\mathcal{M}_{i_j}).$$

By the above-mentioned bound on the degree of polynomials in the generators of $\mathcal{M}_i$ together with Corollary 5.3, we infer the following properties:

- $m - i_n \leq \deg(\mathcal{M}_{i_n})n \leq i_n d_{max} n$;
- and for all $j \in \{2, \cdots, n\}$, we have

$$(i_j - 1) - i_{j-1} \leq \deg(\mathcal{M}_{i_{j-1}})(j-1) \leq i_{j-1} d_{max}(j-1). \tag{9}$$

By telescoping (9) from i_j to i_1, we get $i_j \leq j + d_{max} \sum_{k=1}^{j-1} i_k k$, where the right-hand side is at most $j + d_{max} i_{j-1} \sum_{k=1}^{j-1} k \leq (d_{max}+1) i_{j-1} j^2$. By a simple induction, for all $j \in \{2, \ldots, n\}$, we obtain that $i_j \leq (d_{max}+1)^{j-1} j^{2(j-1)}$, and so $m \leq (d_{max}+1)^n n^{2n} = L(d_{max}, n)$ holds .

Now we are ready to analyse the maximum number of columns added to the sequence C during the successive recursive calls to the procedure `partial_learner` $\ell = L(d, n')$. Let $\mathcal{R}_1$, $\mathcal{R}_2$, ... and C_1, C_2, ... be the successive values passed to `partial_learner`, where $\mathcal{R}_1, C_1$ are initialized to $[\varepsilon]$.

Using Proposition 5.6, the construction of the hypothesis automaton in Corollary 5.7, and by Corollary 5.9, we deduce that for every C_k, in the successive values C_1, C_2, ... as defined above, for $d' := d(|C_k|+1)|C_k|$, the matrix $A_{d'}(\mathcal{R}_k, C_k)$ is full row rank and also C_k is totally suffix-closed. Moreover, writing $C_k = [c_1, \ldots, c_k]$, as above, we define the sequence of modules $\mathcal{M}_1$, ..., $\mathcal{M}_k$ where $\mathcal{M}_i = \left\langle B_{\mathcal{A}}(c_j) : j \leq i \right\rangle_{\mathbb{Q}[x]}$, with $1 \leq i \leq k$. By Propositions 5.8 and 5.10 we know that $\mathcal{M}_1 \subsetneq \ldots \subsetneq \mathcal{M}_k$, that is, the sequence consists of strictly increasing modules. Since $d \geq d_{max}$ and $n' \geq n$, the inequality $L(n', d) \geq L(n, d_{\max})$ implies that $L(n', d) \geq |C_k|$ for every C_k in the sequence of C_1, C_2, ... This concludes that `exact_learner`$(d, L(d, n'), [\varepsilon], [\varepsilon]) \neq$ None. □

5.2.3 Bounding the number of rows. As previously mentioned, every row added by the learning algorithm is a prefix of a counterexample given by the teacher. In this section, we exhibit a bound on the total number of rows added during the learning procedure, which is polynomial in the maximum size of the counterexamples and the target automaton. For this, we define a *bounded forward vector space* that only considers words of bounded size.

Let $\mathcal{A} = (\boldsymbol{\alpha}, \mu, \boldsymbol{\beta})$ be a P-finite automaton of dimension n over Σ. Let $s \in \mathbb{N}$. The *s-bounded forward function* associated with $\mathcal{A}$ is the function $F^s_{\mathcal{A}} : \{r \in \Sigma^* : |r| \leq s\} \longrightarrow \mathbb{Q}^{1\times(s+1)n}$ given by:

$$F^s_{\mathcal{A}}(u) = \begin{bmatrix}\mathbf{0}_{1\times|u|n} & \boldsymbol{\alpha}\mu(u,1) & \mathbf{0}_{1\times(s-|u|)n}\end{bmatrix}.$$

The *s-bounded forward space*, denoted $\mathcal{F}^s_{\mathcal{A}}$, is the vector space $\left\langle F^s_{\mathcal{A}}(u) : |u| \leq s, u \in \Sigma^*\right\rangle_{\mathbb{Q}}$.

Observe that $\mathcal{F}^s_{\mathcal{A}}$ consists of row vectors from $\mathbb{Q}^{1\times(s+1)n}$. Thus, the dimension of the vector space $\mathcal{F}^s_{\mathcal{A}}$ is at most $(s+1)n$. Intuitively, here s represents the maximal size of the counterexamples given by the teacher. In the following proposition, we show that if $r_1, \ldots, r_m$ are the prefixes of the counterexamples added as rows during the learning procedure, then the dimension of $\left\langle F^s_{\mathcal{A}}(r_1), \ldots, F^s_{\mathcal{A}}(r_m)\right\rangle_{\mathbb{Q}}$ is m.

Proposition 5.12. *Let $\mathcal{R}$ and C be sequences of words. Let $d, s \in \mathbb{N}$ be such that $A_d(\mathcal{R}, C)$ has full row rank, and $|r| < s$ for all words $r \in \mathcal{R}$. The dimension of the vector space $\left\langle F^s_{\mathcal{A}}(r) : r \in \mathcal{R}\right\rangle_{\mathbb{Q}}$ is $|\mathcal{R}|$.*

Proof. Let $\mathcal{R} = [r_1, \ldots, r_m]$. It suffices to argue that the set $\{F^s_{\mathcal{A}}(r_i) : 1 \leq i \leq m\}$ is linearly independent. Towards a contradiction, we assume without loss of generality that $F^s_{\mathcal{A}}(r_m)$ is dependant on the other $F^s_{\mathcal{A}}(r_i)$, meaning that that there exist $q_1, \ldots, q_{m-1} \in \mathbb{Q}$ such that $F^s_{\mathcal{A}}(r_m) = \sum_{i=1}^{m-1} q_i F^s_{\mathcal{A}}(r_i)$. By the definition of $F^s_{\mathcal{A}}$, as the vectors $F^s_{\mathcal{A}}(r_i)$ have only n non-zero entries, we can further without loss of generality assume that $q_i \neq 0$, with $1 \leq i \leq m-1$, implies $|r_i| = |r_m|$. Hence, for all $k \in \mathbb{N}$,

$$(|r_m|+1)^k F^s_{\mathcal{A}}(r_m) = \sum_{i=1}^{m-1} q_i(|r_m|+1)^k F^s_{\mathcal{A}}(r_i) = \sum_{i=1}^{m-1} q_i(|r_i|+1)^k F^s_{\mathcal{A}}(r_i). \tag{10}$$

Finally, for all $c \in \Sigma^*$, denote by $B(c) \in \mathbb{Q}^{n(s+1)\times 1}$ the column vector defined as follows:

$$B(c) = \begin{bmatrix} B_{\mathcal{A}}(c)(1) \\ \vdots \\ B_{\mathcal{A}}(c)(s+1)\end{bmatrix}.$$

By the definitions of forward and backward functions, for all words $r \in \mathcal{R}$ and $c \in \Sigma^*$, we have

$$F^s_{\mathcal{A}}(r)B(c) = \boldsymbol{\alpha}\mu(r,1)B_{\mathcal{A}}(c)(|r|+1) = f(rc),$$

where the first equality holds due to the match of placement of non-zero entries in $F^s_{\mathcal{A}}(r)$ with the placement of $B_{\mathcal{A}}(c)(|r|+1)$ in $B(c)$. As an immediate result of the above equation and (10), we obtain that, for all words $c \in C$ and for all $k \leq d$,

$$\begin{aligned}(|r_m|+1)^k f(r_m c) =& (|r_m|+1)^k F^s_{\mathcal{A}}(r_m)B(c) \\ =& \sum_{i=1}^{m-1} q_i(|r_i|+1)^k F^s_{\mathcal{A}}(r_i)B(c) \\ =& \sum_{i=1}^{m-1} q_i(|r_i|+1)^k f(r_i c).\end{aligned}$$

Therefore, if we denote by $A_1, \ldots, A_m$ the rows of $A_d(\mathcal{R}, C)$ then we have shown that $A_m = \sum_{i=1}^{m-1} q_i A_i$ which is in contradiction with $\mathrm{rank}(A_d(\mathcal{R}, C)) = |\mathcal{R}|$. □

Proposition 5.13. *Let $d, \ell \in \mathbb{N}$. Let s be the maximal length of counterexamples given by the teacher during the execution of* `partial_learner`$(d, \ell, [\varepsilon], [\varepsilon])$. *Then the number of recursive calls to* `partial_learner` *is at most $(s+1)n$, where n is the number of states in the target automata.*

Figure 11: The exact learner.

```
def exact_learner()=
    $sum := 1$
    while $true$ do
        for $n = 1$ to $sum$ do
            $d = sum - n$
            $\ell = 2(d+1)^n n^{2n}$
            match partial_learner($d, \ell, [\varepsilon], [\varepsilon]$) with
                | None -> ()
                | Some($\mathcal{H}$) -> return $\mathcal{H}$
        $sum := sum + 1$
```

Proof. Using Corollary 5.9 and Proposition 5.6, for every call to `partial_learner` with arguments $d, \ell, \mathcal{R}, C$, we have $\text{rank}(A_{d'}(\mathcal{R}, C)) = |\mathcal{R}|$ with $d' = d(|C|+1)|C|$.

By Proposition 5.12, the dimension of the vector space $\left\langle F^s_{\mathcal{A}}(u) : u \in \mathcal{R} \right\rangle_{\mathbb{Q}}$ is $|\mathcal{R}|$. But then, since $\left\langle F^s_{\mathcal{A}}(u) : u \in \mathcal{R} \right\rangle_{\mathbb{Q}} \subseteq \mathbb{Q}^{1 \times (s+1)n}$, we have $|\mathcal{R}| \leq (s+1)n$. We note that every recursive call to `partial_learner` increases the size of $\mathcal{R}$ by one starting from $[\varepsilon]$. Therefore, the number of recursive calls to `partial_learner` is at most $(s+1)n$. □

5.2.4 The exact learner. The exact learner function is displayed in Figure 11. The core of the learning process comes from the procedure `partial_learner`. However, it still remains to correctly guess the values of d_{max}, the maximal degree of polynomials in the target automaton $\mathcal{A}$, and of n, its number of states. Guessing n is important in order to compute the limit value ℓ for the number of columns added during the execution of `partial_learner`. As we need to guess two positive integers, we use the standard diagonal progression of the Cantor pairing function.

The following theorem shows that P-finite automata can be exactly learned in time polynomial in the size of the target automaton and the maximal length of counterexamples given by the teacher.

Theorem 5.14. *Let $\mathcal{A}$ be the target P-finite automaton. The procedure* `exact_learner` *terminates and returns a P-finite automaton $\mathcal{H}$ such that $[\![\mathcal{A}]\!] = [\![\mathcal{H}]\!]$. Moreover,* `exact_learner` *runs in time polynomial in the length of the encoding of $\mathcal{A}$ and the maximal length of counterexamples given by the teacher during the execution of* `exact_learner`.

Proof. Let $n_{\mathcal{A}}$ be the number of states in the target automaton $\mathcal{A}$ and d_{max} be the maximal degree of polynomials in $\mathcal{A}$. Let s be the maximal size of all counterexamples given by `equivalence_oracle` during the execution of `exact_learner`.

By Proposition 5.13, when executing `partial_learner`($d, \ell, [\varepsilon], [\varepsilon]$), the number of recursive call to `partial_learner` is at most $(s+1)n_{\mathcal{A}}$, irrespective of the choice of d, ℓ. Furthermore, by the construction of `partial_learner`, we also know that for every call to `partial_learner` with arguments $d, \ell, \mathcal{R}, C$, the size of $\mathcal{R}$ is at most $(s+1)n_{\mathcal{A}}$ and $|C| \leq |\mathcal{R}|$.

Define $d := d_{max}(|C|+1)|C|$. Observe that $d \leq d_{max}((s+1)n_{\mathcal{A}}+1)^2$. Recall that the procedure `build_automata`($d, \mathcal{R}, C$) runs in time polynomial in $d + |\mathcal{R}| + |C|$, and also the procedure `largest_correct_prefix`($\mathcal{H}, w$) runs in time polynomial in $|w| + |C|$. We note that the rank of $A_d(\mathcal{R} \cdot [u], C)$ can be computed in time polynomial in $(d+1)|C|(|\mathcal{R}|+1)$ as well.

By the above, the computation of one recursive call of `partial_learner` is polynomial in $s + d_{max} + n_{\mathcal{A}}$, which in turns implies that each execution of `partial_learner(`$d, \ell, [\varepsilon], [\varepsilon]$`)` runs in time polynomial in s and in the length of the encoding of $\mathcal{A}$, irrespective of the choice of d, ℓ.

Intuitively, the variables n, d in `exact_learner` are the "guessed" values for the number of states and the degrees of polynomials in the target automaton, and the variable *sum* is to implement the standard diagonal progression for n and d. By Proposition 5.11, we know that when the variable *sum* in Line 10 in `exact_learner` reaches the value $d_{max} + n_{\mathcal{A}}$, in the inner **for**-loop, the output of the call to `partial_learner` with d set to d_{max} and ℓ set to $(d+1)^{n}_{\mathcal{A}} n_{\mathcal{A}}^{2n_{\mathcal{A}}}$ is necessarily different from None, terminating the computation (the procedure might terminate before this point). Therefore, we will call `partial_learner(`$d, \ell, [\varepsilon], [\varepsilon]$`)` for different values of d, ℓ at most $(d_{max} + n_{\mathcal{A}})^2$ times, which concludes the proof. □

ACKNOWLEDGMENTS

Mahsa Shirmohammadi is supported by International Emerging Actions grant (IEA'22), by ANR grant VeSyAM (ANR-22- CE48-0005) and by the grant CyphAI (ANR-CREST-JST).

REFERENCES

Dana Angluin. 1987. Learning Regular Sets from Queries and Counterexamples. *Inf. Comput.* 75, 2 (1987), 87–106. https://doi.org/10.1016/0890-5401(87)90052-6

Dana Angluin and Michael Kharitonov. 1995. When Won't Membership Queries Help? *J. Comput. Syst. Sci.* 50, 2 (1995), 336–355.

Amos Beimel, Francesco Bergadano, Nader Bshouty, Eyal Kushilevitz, and Stefano Varricchio. 1999. Learning Functions Represented as Multiplicity Automata. *J. ACM* 47 (10 1999). https://doi.org/10.1007/978-0-387-30162-4_194

Michael Benedikt, Timothy Duff, Aditya Sharad, and James Worrell. 2017. Polynomial automata: Zeroness and applications. In *32nd Annual ACM/IEEE Symposium on Logic in Computer Science, LICS 2017*. IEEE Computer Society, 1–12.

Jean Berstel and Christophe Reutenauer. 1988. *Rational series and their languages*. Vol. 12. Springer-Verlag.

Jean Berstel and Christophe Reutenauer. 2010. . Encyclopedia of Mathematics and its Applications, Vol. 137. Cambridge University Press.

Benedikt Bollig, Peter Habermehl, Carsten Kern, and Martin Leucker. 2009. Angluin-Style Learning of NFA. In *IJCAI 2009, Proceedings of the 21st International Joint Conference on Artificial Intelligence*. 1004–1009.

Benedikt Bollig, Joost-Pieter Katoen, Carsten Kern, Martin Leucker, Daniel Neider, and David R. Piegdon. 2010. libalf: The Automata Learning Framework. In *CAV 2010 (LNCS, Vol. 6174)*. Springer, Edinburgh, UK, 360–364.

J.W.S. Cassels and A. Fröhlich. 2010. *Algebraic Number Theory: Proceedings of an Instructional Conference Organized by the London Mathematical Society (a NATO Advanced Study Institute) with the Support of the International Mathematical Union*. London Mathematical Society. https://books.google.fr/books?id=DQP_RAAACAAJ

Michel Fliess. 1974. Matrices de hankel. *J. Math. Pures Appl* 53, 9 (1974), 197–222.

Falk Howar, Bengt Jonsson, and Frits W. Vaandrager. 2019. Combining Black-Box and White-Box Techniques for Learning Register Automata. In *Computing and Software Science - State of the Art and Perspectives*. LNCS, Vol. 10000. Springer, Cham, 563–588.

Andreas Humenberger, Maximilian Jaroschek, and Laura Kovács. 2017a. Automated generation of non-linear loop invariants utilizing hypergeometric sequences. In *Proceedings of the 2017 ACM on International Symposium on Symbolic and Algebraic Computation*. 221–228.

Andreas Humenberger, Maximilian Jaroschek, and Laura Kovács. 2017b. Invariant generation for multi-path loops with polynomial assignments. In *International Conference on Verification, Model Checking, and Abstract Interpretation*. Springer, 226–246.

Thomas W. Hungerford. 1974. *Algebra*. Springer, New York, NY, 225.

Malte Isberner, Falk Howar, and Bernhard Steffen. 2015. The Open-Source LearnLib - A Framework for Active Automata Learning. In *CAV 2015 (LNCS, Vol. 9206)*. Springer, San Francisco, CA, USA, 487–495.

R. Kannan. 1985. Solving systems of linear equations over polynomials. *Theoretical Computer Science* 39 (1985), 69–88. https://doi.org/10.1016/0304-3975(85)90131-8 Third Conference on Foundations of Software Technology and Theoretical Computer Science.

Manuel Kauers and Peter Paule. 2011. *The Concrete Tetrahedron* (1st ed.). Springer Wien.

Stefan Kiefer. 2020. Notes on Equivalence and Minimization of Weighted Automata. arXiv:2009.01217 [cs.FL]

Laura Kovács. 2008. Reasoning Algebraically About P-Solvable Loops. In *Tools and Algorithms for the Construction and Analysis of Systems, 14th International Conference, TACAS, Proceedings (Lecture Notes in Computer Science, Vol. 4963)*. Springer, 249–264.

Jakub Michaliszyn and Jan Otop. 2022. Learning Deterministic Visibly Pushdown Automata Under Accessible Stack. In *47th International Symposium on Mathematical Foundations of Computer Science, MFCS (LIPIcs, Vol. 241)*. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 74:1–74:16.

Joshua Moerman, Matteo Sammartino, Alexandra Silva, Bartek Klin, and Michal Szynwelski. 2017. Learning nominal automata. In *Proceedings of the 44th ACM SIGPLAN Symposium on Principles of Programming Languages, POPL*. ACM, 613–625.

Morris Newman. 1997. The Smith normal form. *Linear algebra and its applications* 254, 1-3 (1997), 367–381.

Christophe Reutenauer. 2012. On a Matrix Representation for Polynomially Recursive Sequences. *Electron. J. Comb.* 19, 3 (2012), 36.

Marcel Paul Schützenberger. 1961. On the Definition of a Family of Automata. *Inf. Control.* 4, 2-3 (1961), 245–270.

Henry John Stephen Smith. 1861. XV. On systems of linear indeterminate equations and congruences. *Philosophical Transactions of the Royal Society of London* 151 (Dec. 1861), 293–326. https://doi.org/10.1098/rstl.1861.0016

Wen-Guey Tzeng. 1992. A Polynomial-Time Algorithm for the Equivalence of Probabilistic Automata. *SIAM J. Comput.* 21, 2 (1992), 216–227. https://doi.org/10.1137/0221017

Gerco van Heerdt, Clemens Kupke, Jurriaan Rot, and Alexandra Silva. 2020. Learning Weighted Automata over Principal Ideal Domains. In *Foundations of Software Science and Computation Structures - 23rd International Conference, FOSSACS 2020, Proceedings (Lecture Notes in Computer Science, Vol. 12077).* Springer, 602–621.

A PROOF OF PROPOSITION 3.3

Recall that in Section 3 we considered a PID R, and R being a PID implies that it is also a unique factorization domain. Thus, for all $r \in R$, we can compute the number of (not necessarily distinct) prime factors of r, that we denote $\pi(r)$, with the convention that $\pi(1_R) = 0$.

Given two R-modules M and N, a *homomorphism* between M and N is an R-linear map $\phi : M \longrightarrow N$. That is, for all $r_1, r_2 \in R$ and $m_1, m_2 \in M$, $\phi(r_1 m_1 + r_2 m_2) = r_1\phi(m_1) + r_2\phi(m_2)$. When ϕ is additionally bijective, we say that ϕ is a *isomorphism* and that M and N are *isomorphic*, denoted $M \simeq N$.

We also define the *direct-sum* R-module as $M \oplus N = \{(m, n) : m \in M, n \in N\}$. We say that M is *torsion* if for all $m \in M$, there exists $r \in R \setminus \{0\}$ such that $rm = 0$. When N is a submodule of M, the *quotient* R-module M/N is the set of elements of the form $m + N = \{m + n : n \in N\}$ for some $m \in M$, endowed with addition and multiplication operations as follows: for all $r \in R$, and $m, m' \in M$, we have $(m + N) + (m' + N) = (m + m') + N$ and $r(m + N) = (rm) + N$.

For example, $3\mathbb{Z}$ is a $\mathbb{Z}$-submodule of $\mathbb{Z}$ and the quotient $\mathbb{Z}/3\mathbb{Z}$ is a torsion $\mathbb{Z}$-module that contains the elements $0 + 3\mathbb{Z}$, $1 + 3\mathbb{Z}$ and $2 + 3\mathbb{Z}$, isomorphic to $\mathbb{Z}_3$. → of rank r

As mentioned in Section 3, for any submodules M of R^n, there exist an R-basis $f_1, \ldots, f_n$ of R^n and elements $d_1, \ldots, d_r \in R$ such that $d_1 f_1, \ldots, d_r f_r$ is an R-basis of M. Moreover, when M is generated by some vectors $v_1, \ldots, v_m$, each d_i are defined as $\frac{D_i(A)}{D_{i-1}(A)}$ where $A \in R^{n \times m}$ is the matrix whose columns are the vectors $v_1, \ldots, v_m$ with $D_0(A) = 1$ by convention. This correlates directly with the following theorem.

Theorem A.1 ([Hungerford 1974, Theorem 6.12]). *Let M be a finitely generated R-module. There exists $d_1, \ldots, d_r \in R$ such that M is isomorphic to $R^k \oplus \bigoplus_{j=1}^{r} R/\langle d_j \rangle_R$ for some k and* invariant factors *$d_1 \mid d_2 \mid \ldots \mid d_r$ that are unique (up to multiplication by units). Moreover, M is torsion iff $k = 0$, and in such a case we define the R-*dimension *of M, denoted* $\dim_R(M)$, *as $\pi(d_1 d_2 \ldots d_r)$.*

The proof of Proposition 3.3 relies on the following well known result:

Theorem A.2 ([Cassels and Fröhlich 2010, Proposition 1]). *Let M, M', M'' be R-submodules of R^n such that $M'' \subset M' \subset M$, all having the same rank. Then*

$$\dim_R(M/M'') = \dim_R(M/M') + \dim_R(M'/M'').$$

The R-dimensions of M/M'', M/M', and M'/M'' are well defined as one can observe that two finitely generated modules have the same rank if and only if their quotient module is torsion. We are now ready to prove a general bound on the length of a strictly increasing sequence of submodules of the same rank.

Proposition 3.3. *Let $n, k \in \mathbb{N}$. Let R be a PID and $M = \langle v_1, \ldots, v_m \rangle$ be a R-submodule of R^n. Let A be the $n \times m$ matrix whose i-th column is v_i. Let $M \subsetneq M_1 \subsetneq M_2 \subsetneq \ldots \subsetneq M_k$ be a strictly increasing chain of R-submodules of R^n, all having the same rank $r \leq n$. Then k is bounded by the number of (not necessarily distinct) prime factors of $D_r(A)$.*

Proof. Let $f_1, \ldots, f_n$ be a R-basis of R^n and let $d_1, \ldots, d_r \in R$ such that $d_1 f_1, \ldots, d_r f_r$ is a R-basis of M and $d_1 \mid d_2 \mid \ldots \mid d_r$. Recall that for all $i \in \{1, \ldots, r\}$, $d_i = \frac{D_i(A)}{D_{i-1}(A)}$ and $D_0(A) = 1_R$ by convention.

Let $N = \langle f_1, \dots, f_r \rangle_R$. We first show that $M_k \subseteq N$, so that the whole increasing sequence is in N. Assume, for a contradiction, that there is some $m \in M_k$ but $m \notin N$. Then, when expressing m in the basis $f_1, \dots, f_n$, at least one coefficient of some f_i, $i > r$, must be non-zero. This further implies that the vectors m, $d_1 f_1$, $d_2 f_2$,..., $d_r f_r$ are linearly independent (as some linear dependence would result in a linear dependence between $f_i, f_1, f_2, \dots, f_r$). But, as $\{d_1 f_1, \dots, d_r f_r\} \subseteq M \subseteq M_k$, the module M_k contains $r + 1$ linearly independent elements. Their span is contained in M_k and has rank $r + 1$, which contradicts the fact that M_k has rank r.

We are now ready to inductively apply Theorem A.2 to the chain $M \subsetneq M_1 \subsetneq M_2 \subsetneq \dots \subsetneq M_k \subseteq N$. Note that all these submodules have rank r, as M does (by definition), and so does N (by construction). We have:

$$\dim_R(N/M) = \dim_R(N/M_k) + \dim_R(M_k/M_{k-1}) + \dots + \dim_R(M_2/M_1) + \dim_R(M_1/M).$$

The first term on the right-hand side is clearly non-negative, so we can bound k as

$$k \le \frac{\dim_R(N/M)}{\min\limits_{1 \le i \le k} \dim_R(M_i/M_{i-1})}.$$

For all $i \in \{1, \dots, k\}$, we can show that $\dim_R(M_i/M_{i-1}) \ge 1$. Otherwise, by contradiction, there would exists $i \in \{1, \dots, k\}$ such that $\dim_R(M_i/M_{i-1}) = 0$, which entails M_i/M_{i-1} is isomorphic to $\bigoplus_{k=1}^{r'} R/\langle 1_R \rangle_R$ for some r'. As $R/\langle 1_R \rangle_R$ is isomorphic to $\{0\}$ then so is M_i/M_{i-1}. This implies that $M_i = M_{i-1}$, contradicting our hypothesis that $M_{i-1} \subsetneq M_i$.

Finally, observe that $\dim_R(N/M) = \pi(d_1 d_2 \dots d_r)$ since $d_1 f_1, \dots, d_r f_r$ is a basis of M and $N = \langle f_1, \dots, f_r \rangle_R$. As $d_i = \frac{D_i(A)}{D_{i-1}(A)}$ for all $i \in \{1, \dots, r\}$, we conclude that $\pi(d_1 \dots d_r) = \pi(\frac{D_r(A)}{D_1(A)}) = \pi(D_r(A))$ and so $k \le \pi(D_r(A))$. □

B PROOFS OF SECTION 5.1

Proposition 5.1. *The equivalence problem of P-finite automata is polynomial-time reducible to the zeroness problem.*

Proof. Let $\mathcal{A}_i = (\boldsymbol{\alpha}_i, \mu_i, \boldsymbol{\beta}_i(x))$ for $i \in \{1, 2\}$ be two P-finite automata over Σ, respectively, of dimension n_1 and n_2. We construct a P-finite automata $\mathcal{A}_- = (\boldsymbol{\alpha}, \mu, \boldsymbol{\beta}(x))$ of dimension $n_1 + n_2$ over Σ such that

$$\boldsymbol{\alpha} = \begin{bmatrix} \boldsymbol{\alpha}_1 & -\boldsymbol{\alpha}_2 \end{bmatrix} \qquad \text{and} \qquad \boldsymbol{\beta}(x) = \begin{bmatrix} \boldsymbol{\beta}_1(x) \\ \boldsymbol{\beta}_2(x) \end{bmatrix},$$

and for all $\sigma \in \Sigma$,

$$\mu(\sigma, x) = \begin{bmatrix} \mu_1(\sigma, x) & 0_{n_1 \times n_2} \\ 0_{n_2 \times n_1} & \mu_2(\sigma, x) \end{bmatrix}.$$

The construction of $\mathcal{A}$ can be done in time polynomial in the size of encoding of $\mathcal{A}_1$ and $\mathcal{A}_2$.

We first claim that $f_{\mathcal{A}_-}(w) = f_{\mathcal{A}_1}(w) - f_{\mathcal{A}_2}(w)$ for all $w \in \Sigma^*$. By definition,

$$f_{\mathcal{A}_-}(\varepsilon) = \boldsymbol{\alpha}\boldsymbol{\beta}(1) = \boldsymbol{\alpha}_1\boldsymbol{\beta}_1(1) - \boldsymbol{\alpha}_2\boldsymbol{\beta}_2(1) = f_{\mathcal{A}_1}(\varepsilon) - f_{\mathcal{A}_2}(\varepsilon),$$

as required. Consider $w = \sigma_1 \dots \sigma_k$ with the $\sigma_i \in \Sigma$. We have that

$$\begin{aligned} f_{\mathcal{A}_-}(w) &= \begin{bmatrix} \boldsymbol{\alpha}_1 & -\boldsymbol{\alpha}_2 \end{bmatrix} \begin{bmatrix} \mu_1(\sigma_1, 1) & 0_{n_1 \times n_2} \\ 0_{n_2 \times n_1} & \mu_2(\sigma_1, 1) \end{bmatrix} \dots \begin{bmatrix} \mu_1(\sigma_k, k) & 0_{n_1 \times n_2} \\ 0_{n_2 \times n_1} & \mu_2(\sigma_k, k) \end{bmatrix} \begin{bmatrix} \boldsymbol{\beta}_1(k+1) \\ \boldsymbol{\beta}_2(k+1) \end{bmatrix} \\ &= \begin{bmatrix} \boldsymbol{\alpha}_1 & -\boldsymbol{\alpha}_2 \end{bmatrix} \begin{bmatrix} \mu_1(\sigma_1, 1) \dots \mu_1(\sigma_k, k) & 0_{n_1 \times n_2} \\ 0_{n_2 \times n_1} & \mu_2(\sigma_1, 1) \dots \mu_2(\sigma_k, k) \end{bmatrix} \begin{bmatrix} \boldsymbol{\beta}_1(k+1) \\ \boldsymbol{\beta}_2(k+1) \end{bmatrix} \\ &= \boldsymbol{\alpha}_1 \mu_1(\sigma_1, 1) \dots \mu_1(\sigma_k, k) \boldsymbol{\beta}_1(k+1) - \boldsymbol{\alpha}_2 \mu_2(\sigma_1, 1) \dots \mu_2(\sigma_k, k) \boldsymbol{\beta}_2(k+1) \\ &= f_{\mathcal{A}_1}(w) - f_{\mathcal{A}_2}(w). \end{aligned}$$

□

We can now conclude the proof by observing that $[\![\mathcal{A}]\!]$ is identically zero if and only if $[\![\mathcal{A}_1]\!] = [\![\mathcal{A}_2]\!]$, as required.