Logical Methods in Computer Science
Vol. 9(1:08)2013, pp. 1–22
www.lmcs-online.org

Submitted Aug. 12, 2012
Published Mar. 4, 2013

ON THE COMPLEXITY OF EQUIVALENCE AND MINIMISATION FOR $\mathbb{Q}$-WEIGHTED AUTOMATA *

STEFAN KIEFER [a], ANDRZEJ S. MURAWSKI [b], JOËL OUAKNINE [c], BJÖRN WACHTER [d], AND JAMES WORRELL [e]

[a,c,d,e] University of Oxford, UK
e-mail address: {stekie, joel, Bjoern.Wachter, jbw}@cs.ox.ac.uk

[b] Department of Computer Science, University of Warwick, UK
e-mail address: A.Murawski@warwick.ac.uk

Abstract. This paper is concerned with the computational complexity of equivalence and minimisation for automata with transition weights in the ring $\mathbb{Q}$ of rational numbers. We use polynomial identity testing and the Isolation Lemma to obtain complexity bounds, focussing on the class **NC** of problems within **P** solvable in polylogarithmic parallel time. For finite $\mathbb{Q}$-weighted automata, we give a randomised **NC** procedure that either outputs that two automata are equivalent or returns a word on which they differ. We also give an **NC** procedure for deciding whether a given automaton is minimal, as well as a randomised **NC** procedure that minimises an automaton. We consider probabilistic automata with rewards, similar to Markov Decision Processes. For these automata we consider two notions of equivalence: expectation equivalence and distribution equivalence. The former requires that two automata have the same expected reward on each input word, while the latter requires that each input word induce the same distribution on rewards in each automaton. For both notions we give algorithms for deciding equivalence by reduction to equivalence of $\mathbb{Q}$-weighted automata. Finally we show that the equivalence problem for $\mathbb{Q}$-weighted visibly pushdown automata is logspace equivalent to the polynomial identity testing problem.

1. Introduction

Probabilistic and weighted automata were introduced in the 1960s, with many fundamental results established in the papers of Schutzenberger [23] and Rabin [21]. Nowadays probabilistic automata are widely used in automated verification, natural-language processing, and machine learning. In this paper we consider weighted automata over the ring

2012 ACM CCS: [**Theory of computation**]: Design and analysis of algorithms—Approximation algorithms analysis—Numeric approximation algorithms & Semantics and reasoning—Program reasoning—Program verification.

Key words and phrases: weighted automata, equivalence checking, polynomial identity testing, minimisation.

* This is a full and improved version of the FoSSaCS'12 paper with the same title. An algorithm from the same authors' CAV'11 paper [15] was incorporated in Section 3.1 and new algorithms for minimisation were added in Section 4. Section 5.1 is also new.

LOGICAL METHODS IN COMPUTER SCIENCE
DOI:10.2168/LMCS-9(1:08)2013

$(\mathbb{Q}, +, \cdot, 0, 1)$, which generalise probabilistic automata. Note that we restrict to rational transition weights to permit effective representation of automata.

Two $\mathbb{Q}$-weighted automata are said to be equivalent if they assign the same weight to any given word. It has been shown by Schutzenberger [23] and later by Tzeng [28] that equivalence for $\mathbb{Q}$-weighted automata is decidable in polynomial time. By contrast, the natural analog of language inclusion, that one automaton accepts each word with weight at least as great as another automaton, is undecidable [9]. Let us emphasize that we consider the standard ring structure on $\mathbb{Q}$. For example, for weighted automata over the max-plus semiring on $\mathbb{Q}$, equivalence is undecidable [2, 18].

In this paper we show that the equivalence problem for $\mathbb{Q}$-weighted automata, and various extensions thereof, can be efficiently solved by techniques rooted in *polynomial identity testing*. We focus on establishing bounds involving complexity classes within the class **P** of polynomial-time solvable problems. In particular, we consider the class **NC** of problems solvable in polylogarithmic parallel time with polynomially many processors [13] (see Section 2 for background on complexity theory).

It has long been known that equivalence for $\mathbb{Q}$-weighted automata can be solved in polynomial time [23, 28]. There is moreover an **NC** algorithm for solving equivalence [29]. Our first contribution, in Section 3, is a randomised **NC** algorithm for deciding equivalence, based on polynomial identity testing. The advantage of using randomisation in this context is that our algorithm has much lower processor complexity than [29]. The latter performs quadratically more work than the classical sequential procedure. On the other hand, our randomised algorithm compared well with the classical sequential algorithm of [23, 28] on a collection of benchmarks [15].

We also show that our algorithm can be used not just to decide equivalence but also to generate counterexamples in case of inequivalence. However the counterexample generation is essentially sequential. We address this deficiency by giving a second randomised **NC** algorithm to decide equivalence of automata and output counterexamples in case of inequivalence. The algorithm is based on the Isolation Lemma, a classical technique in randomised algorithms that has previously been used, e.g., to derive randomised **NC** algorithms for matching in graphs [20]. Whether there is a deterministic **NC** algorithm that outputs counterexamples in case of inequivalence remains open.

A $\mathbb{Q}$-weighted automaton is *minimal* if no equivalent automaton has fewer states. Minimal automata are unique up to change of basis. In Section 4 we give an **NC** procedure to decide if a given automaton is minimal. For the associated function problem, that of minimising a given automaton, we give a randomised **NC** procedure. Thus the situation for minimisation is similar to that for equivalence: the decision problem is in **NC** whereas the function problem can only be shown to be in **RNC**.

In Section 5 we consider probabilistic automata with rewards on transitions, which can be seen as partially observable Markov decision processes. Rewards (and costs, which can be considered as negative rewards) are omnipresent in probabilistic modelling for capturing quantitative effects of probabilistic computations, such as consumption of time, allocation of memory, energy usage, etc. For these automata we consider a notion of *expectation equivalence*, requiring that two automata have the same expected reward on each input word, and a stronger notion of *distribution equivalence*, requiring that each word induce the same distribution on rewards in both automata. In both cases we give decision procedures for equivalence by reduction to the case of $\mathbb{Q}$-weighted automata, thus inheriting the complexity bounds established there.

We present a case study in which costs are used to model the computation time required by an RSA encryption algorithm, and show that the vulnerability of the algorithm to timing attacks depends on the equivalence of associated probabilistic reward automata. In [17] two possible defenses against such timing leaks were suggested. We also analyse their effectiveness.

In Section 6 we consider pushdown automata. Probabilistic pushdown automata are a natural model of recursive probabilistic procedures, stochastic grammars and branching processes [12, 19]. The equivalence problem for deterministic pushdown automata has been extensively studied [26, 27]. We study the equivalence problem for $\mathbb{Q}$-*weighted visibly pushdown automata (VPA)* [3]. In a visibly pushdown automaton the stack operation of a given transition—whether to pop or push—is determined by the input symbol being read.

We show that the equivalence problem for $\mathbb{Q}$-weighted VPA is logspace equivalent to *Arithmetic Circuit Identity Testing (**ACIT**)*, which is the problem of determining equivalence of polynomials presented via arithmetic circuits [1]. Several polynomial-time randomized algorithms are known for **ACIT**, but it is a major open problem whether it can be solved in polynomial time by a deterministic algorithm. A closely related result is that of Seidl [25], that equivalence of $\mathbb{Q}$-weighted tree automata is decidable in randomised polynomial time. However [25] does not establish a connection with **ACIT** in either direction.

2. Preliminaries

2.1. **Complexity Classes.** Recall that **NC** is the subclass of **P** comprising those problems considered efficiently parallelisable. **NC** can be defined via *parallel random-access machines (PRAMs)*, which consist of a set of processors communicating through a shared memory. A problem is in **NC** if it can be solved in time $(\log n)^{O(1)}$ (polylogarithmic time) on a PRAM with $n^{O(1)}$ (polynomially many) processors. A more abstract definition of **NC** is as the class of languages which have **L**-uniform Boolean circuits of polylogarithmic depth and polynomial size. More specifically, denote by $\mathbf{NC}^k$ the class of languages which have circuits of depth $O(\log^k n)$. The complexity class **RNC** consists of those languages with randomized **NC** algorithms. We have the following chain of inclusions, none of which is known to be strict:

$$\mathbf{NC}^1 \subseteq \mathbf{L} \subseteq \mathbf{NL} \subseteq \mathbf{NC}^2 \subseteq \mathbf{NC} \subseteq \mathbf{RNC} \cap \mathbf{P} \subseteq \mathbf{P}\,.$$

We also have $\mathbf{NC}^k \subseteq \mathbf{SPACE}(O(\log^k n))$, that is, problems in **NC** are solvable in polylogarithmic space.

Problems in **NC** include reachability in directed graphs, computing the rank and determinant of an integer matrix, solving linear systems of equations, and the Tree Isomorphism problem. Problems that are **P**-hard under logspace reductions include Circuit Value and Max Flow. Such problems are not in **NC** unless $\mathbf{P} = \mathbf{NC}$. Problems in $\mathbf{RNC} \cap \mathbf{P}$ include matching in graphs and max flow in 0/1-valued networks. In both cases these problems have resisted classification as either being in **NC** or **P**-hard. See [13] for more details about **NC** and **RNC**.

2.2. **Linear Algebra.** Given an $m \times n$ matrix $A = (a_{ij})$ and a $k \times l$ matrix $B = (b_{ij})$, the *Kronecker product* $A \otimes B$ is an $km \times nl$ matrix defined by

$$A \otimes B = \begin{bmatrix} a_{11}B & \cdots & a_{1n}B \\ \vdots & & \vdots \\ a_{m1}B & \cdots & a_{mn}B \end{bmatrix}$$

The following is a key property of the *Kronecker product*:

Proposition 2.1. *$(A \otimes B)(C \otimes D) = (AC \otimes BD)$ for matrices A, B, C, D of appropriate dimensions.*

Given two $m \times n$ matrices $A = (a_{ij})$ and $B = (b_{ij})$, the *Hadamard product* $C = A \odot B$ is the $m \times n$ matrix defined by $c_{ij} = a_{ij}b_{ij}$.

2.3. **Laurent Polynomials.** A *Laurent polynomial* in variables $t_1, \ldots, t_n$ with coefficients in $\mathbb{Q}$ is an expression of the form $p = \sum_{i \in I} a_i t_1^{i_1} \ldots t_n^{i_n}$, where $I \subseteq \mathbb{Z}^n$ is a finite set and $a_i \in \mathbb{Q}$. We say that p has degree bound d if $|i_1| + \ldots + |i_n| \leq d$. We write $\mathbb{Q}[t_1, t_1^{-1}, \ldots, t_n, t_n^{-1}]$ for the ring of such polynomials, with the usual addition and multiplication operations; we furthermore write $\mathbb{Q}(t_1, t_1^{-1}, \ldots, t_n, t_n^{-1})$ for the corresponding field of fractions, whose elements are quotients of Laurent polynomials.

The following proposition immediately follows from the cofactor formula for matrix inversion.

Proposition 2.2. *Let M be an $m \times m$ matrix with entries in $\mathbb{Q}[t_1, t_1^{-1}, \ldots, t_n, t_n^{-1}]$ of degree bound d. If $\det(I - M) \neq 0$, then $I - M$ is invertible over $\mathbb{Q}(t_1, t_1^{-1}, \ldots, t_n, t_n^{-1})$, and each entry of $(I-M)^{-1}$ can be represented as the quotient of Laurent polynomials, each of degree bound at most md.*

In the situation of Proposition 2.2 we denote $(I - M)^{-1}$ by M^*.

3. Equivalence of $\mathbb{Q}$-Weighted Automata

Given a field $(\mathbb{F}, +, \cdot, 0, 1)$, an *$\mathbb{F}$-weighted automaton* $\mathcal{A} = (n, \Sigma, M, \boldsymbol{\alpha}, \boldsymbol{\eta})$ consists of a positive integer $n \in \mathbb{N}$ representing the number of states, a finite alphabet Σ, a map $M : \Sigma \to \mathbb{F}^{n \times n}$ assigning a transition matrix to each alphabet symbol, an initial (row) vector $\boldsymbol{\alpha} \in \mathbb{F}^n$, and a final (column) vector $\boldsymbol{\eta} \in \mathbb{F}^n$. We extend M to Σ^* as the matrix product $M(\sigma_1 \ldots \sigma_k) := M(\sigma_1) \cdot \ldots \cdot M(\sigma_k)$. The automaton $\mathcal{A}$ assigns to each word w a *weight* $\mathcal{A}(w) \in \mathbb{F}$, where $\mathcal{A}(w) := \boldsymbol{\alpha} M(w) \boldsymbol{\eta}$. An automaton $\mathcal{A}$ is said to be *zero* if $\mathcal{A}(w) = 0$ for all $w \in \Sigma^*$. Two automata $\mathcal{B}, \mathcal{C}$ over the same alphabet Σ are said to be *equivalent* if $\mathcal{B}(w) = \mathcal{C}(w)$ for all $w \in \Sigma^*$.

Given two automata $\mathcal{B}, \mathcal{C}$ that are to be checked for equivalence, one can compute an automaton $\mathcal{A}$ with $\mathcal{A}(w) = \mathcal{B}(w) - \mathcal{C}(w)$ for all $w \in \Sigma^*$. Then $\mathcal{A}$ is zero if and only if $\mathcal{B}$ and $\mathcal{C}$ are equivalent. Given $\mathcal{B} = (n^{(\mathcal{B})}, \Sigma, M^{(\mathcal{B})}, \boldsymbol{\alpha}^{(\mathcal{B})}, \boldsymbol{\eta}^{(\mathcal{B})})$ and $\mathcal{C} = (n^{(\mathcal{C})}, \Sigma, M^{(\mathcal{C})}, \boldsymbol{\alpha}^{(\mathcal{C})}, \boldsymbol{\eta}^{(\mathcal{C})})$, set $\mathcal{A} = (n, \Sigma, M, \boldsymbol{\alpha}, \boldsymbol{\eta})$ with $n := n^{(\mathcal{B})} + n^{(\mathcal{C})}$ and

$$M(\sigma) := \begin{pmatrix} M^{(\mathcal{B})}(\sigma) & 0 \\ 0 & M^{(\mathcal{C})}(\sigma) \end{pmatrix}, \qquad \boldsymbol{\alpha} := (\boldsymbol{\alpha}^{(\mathcal{B})}, -\boldsymbol{\alpha}^{(\mathcal{C})}), \qquad \boldsymbol{\eta} := \begin{pmatrix} \boldsymbol{\eta}^{(\mathcal{B})} \\ \boldsymbol{\eta}^{(\mathcal{C})} \end{pmatrix}.$$

This reduction allows us to focus on *zeroness*, i.e., the problem of determining whether a given $\mathbb{F}$-weighted automaton is zero. (Since transition weights can be negative, zeroness is

not the same as emptiness of the underlying unweighted automaton.) Note that a witness word $w \in \Sigma^*$ against zeroness of $\mathcal{A}$ is also a witness against the equivalence of $\mathcal{B}$ and $\mathcal{C}$.

In the remainder of this section we present two randomised $\mathbf{NC}^2$ algorithm algorithms for deciding equivalence of $\mathbb{Q}$-weighted automata. The following result from [28] immediately implies decidability of testing zeroness, and hence equivalence, of $\mathbb{Q}$-weighted automata.

Proposition 3.1. *Let $\mathbb{F}$ be any field and $\mathcal{A} = (n, \Sigma, M, \boldsymbol{\alpha}, \boldsymbol{\eta})$ an $\mathbb{F}$-weighted automaton. Then: (i) $\operatorname{span}\{\boldsymbol{\alpha}M(w) : w \in \Sigma^*\} = \operatorname{span}\{\boldsymbol{\alpha}M(w) : w \in \Sigma^{<n}\}$; (ii) if $\mathcal{A}$ is not equal to the zero automaton then there exists a word $w \in \Sigma^*$ of length at most $n-1$ such that $\mathcal{A}(w) \neq 0$.*

3.1. **Algorithm Based on the Schwartz-Zippel Lemma.** By Proposition 3.1 a $\mathbb{Q}$-weighted automaton with n states is zero if and only if its *n-bounded language* is zero, that is, it assigns weight zero to all words of length at most n. Inspired by the work of Blum, Carter and Wegman on free Boolean graphs [5], we represent the *n-bounded language* of an automaton by a polynomial in which each monomial represents a word and the coefficient of the monomial represents the weight of the word. We thereby reduce the zeroness problem to polynomial identity testing, for which there are a number of efficient randomised procedures.

Let $\mathcal{A} = (n, \Sigma, M, \boldsymbol{\alpha}, \boldsymbol{\eta})$ be a $\mathbb{Q}$-weighted automaton. We introduce a family of variables $\boldsymbol{x} = \{x_{\sigma,i} : \sigma \in \Sigma,\ 1 \leq i \leq n\}$ and associate the monomial $x_{w_1,1}x_{w_2,2}\dots x_{w_k,k}$ with a word $w = w_1w_2\dots w_k$ of length $k \leq n$. Then we define the polynomial $P(\boldsymbol{x})$ by

$$P(\boldsymbol{x}) := \sum_{k=0}^{n-1} \sum_{w \in \Sigma^k} \mathcal{A}(w) \cdot x_{w_1,1}x_{w_2,2}\dots x_{w_k,k}\,. \tag{3.1}$$

It is immediate from Proposition 3.1 that $P(\boldsymbol{x}) \equiv 0$ if and only if $\mathcal{A}$ is zero.

To test whether $P(\boldsymbol{x}) \equiv 0$ we select a value for each variable $x_{\sigma,i}$ independently and uniformly at random from a set of integers of size Kn, for some constant K. Clearly if $P(\boldsymbol{x}) \equiv 0$ then this yields the value 0. On the other hand, if $P(\boldsymbol{x}) \not\equiv 0$ then P will evaluate to a nonzero value with probability at least $(K-1)/K$ by the following result of De Millo and Lipton [11], Schwartz [24] and Zippel [30] and the fact that P has degree $n-1$.

Theorem 3.2 ([11, 24, 30])**.** *Let $\mathbb{F}$ be a field and $Q(x_1, \dots, x_n) \in \mathbb{F}[x_1, \dots, x_n]$ a multivariate polynomial of total degree d. Fix a finite set $\mathbb{S} \subseteq \mathbb{F}$, and let $r_1, \dots, r_n$ be chosen independently and uniformly at random from $\mathbb{S}$. Then*

$$\Pr[Q(r_1, \dots, r_n) = 0 \mid Q(x_1, \dots, x_n) \not\equiv 0] \leq \frac{d}{|\mathbb{S}|}\,.$$

While the number of monomials in P is proportional to $|\Sigma|^n$, i.e., exponential in n, writing

$$P(\boldsymbol{x}) = \boldsymbol{\alpha}\left(\sum_{i=0}^{n}\prod_{j=1}^{i}\sum_{\sigma\in\Sigma} x_{\sigma,j} \cdot M(\sigma)\right)\boldsymbol{\eta} \tag{3.2}$$

it is clear that P can be evaluated on a particular set of numerical arguments in time polynomial in n. The formula (3.2) can be evaluated in a forward direction, starting with the initial state vector $\boldsymbol{\alpha}$ and post-multiplying by the transition matrices, or in a backward

Algorithm ZERO

Input: Automaton $\mathcal{A} = (n, \Sigma, M, \boldsymbol{\alpha}, \boldsymbol{\eta})$

if $\boldsymbol{\alpha}\boldsymbol{\eta} \neq 0$
    return "$\boldsymbol{\alpha}\boldsymbol{\eta} = \mathcal{A}(\varepsilon) \neq 0$"
$\boldsymbol{v} := \boldsymbol{\eta}$
for i from 1 to n do
    choose a random vector $r \in \{1, 2, \ldots, Kn\}^{\Sigma}$
    $\boldsymbol{v} := \sum_{\sigma \in \Sigma} r(\sigma) M(\sigma) \boldsymbol{v}$
    if $\boldsymbol{\alpha}\boldsymbol{v} \neq 0$
        return "$\exists w$ with $|w| = i$ such that $\mathcal{A}(w) \neq 0$"
return "$\mathcal{A}$ is zero with probability at least $(K-1)/K$"

Figure 1: Algorithm for testing zeroness

Algorithm ZERO + CEX

Input: Automaton $\mathcal{A} = (n, \Sigma, M, \boldsymbol{\alpha}, \boldsymbol{\eta})$

if $\boldsymbol{\alpha}\boldsymbol{\eta} \neq 0$
    return "$\boldsymbol{\alpha}\boldsymbol{\eta} = \mathcal{A}(\varepsilon) \neq 0$"
$\boldsymbol{v}_0 := \boldsymbol{\eta}$
for i from 1 to n do
    choose a random vector $r \in \{1, 2, \ldots, Kn\}^{\Sigma}$
    $\boldsymbol{v}_i := \left(\sum_{\sigma \in \Sigma} r(\sigma) M(\sigma)\right) \boldsymbol{v}_{i-1}$
    if $\boldsymbol{\alpha}\boldsymbol{v}_i \neq 0$
        $w := \varepsilon$
        $\boldsymbol{u} := \boldsymbol{\alpha}$
        for j from i downto 1 do
            choose $\sigma \in \Sigma$ with $\boldsymbol{u} M(\sigma) \boldsymbol{v}_{j-1} \neq 0$
            $w := w\sigma$
            $\boldsymbol{u} := \boldsymbol{u} M(\sigma)$
        return "$\boldsymbol{u}\boldsymbol{\eta} = \mathcal{A}(w) \neq 0$"
return "$\mathcal{A}$ is zero with probability at least $(K-1)/K$"

Figure 2: Algorithm for testing zeroness, with counterexamples

direction, starting with the final state vector $\boldsymbol{\eta}$ and pre-multiplying by the transition matrices. In either case we get a polynomial-time Monte-Carlo algorithm for testing zeroness of $\mathbb{Q}$-weighted automata. The backward variant is shown in Figure 1.

The algorithm runs in time $O(n \cdot |M|)$, where $|M|$ is the number of nonzero entries in all $M(\sigma)$, provided that sparse-matrix representations are used. In a set of case studies this randomised algorithm outperformed deterministic algorithms [15].

We can obtain counterexamples from the randomised algorithm by exploiting the self-reducible structure of the equivalence problem. We generate counterexamples incrementally, starting with the empty string and using the randomised algorithm as an oracle to know at each stage what to choose as the next letter in our counterexample. For efficiency reasons it is important to avoid repeatedly running the randomised algorithm. In fact, as shown in

Figure 2, this can all be made to work with some post-processing following a single run of the randomised procedure.

To evaluate the polynomial $P(\boldsymbol{x})$ we substitute a set of randomly chosen rational values $\boldsymbol{r} = \{r_{\sigma,i} : \sigma \in \Sigma,\ 1 \leq i \leq n\}$ into Equation (3.2). Here we generalize this to a notion of *partial evaluation* $P_w(\boldsymbol{r})$ of polynomial P with respect to values $\boldsymbol{r}$ and a word $w \in \Sigma^m$, $m \leq n$. We define

$$P_w(\boldsymbol{r}) = \boldsymbol{\alpha} M(w) \left(\sum_{i=m}^{n} \prod_{j=m+1}^{i} \sum_{\sigma \in \Sigma} r_{\sigma,j}\, M(\sigma) \right) \boldsymbol{\eta}\,. \tag{3.3}$$

Notice that $P_\varepsilon(\boldsymbol{r}) = P(\boldsymbol{r})$, where ε is the empty word, and, at the other extreme, $P_w(\boldsymbol{r}) = \mathcal{A}(w)$ for any word w of length n.

Proposition 3.3. *Suppose that $w \in \Sigma^m$, where $m < n$. If $P_w(\boldsymbol{r}) \neq 0$ then either $\mathcal{A}(w) \neq 0$ or $P_{w\sigma}(\boldsymbol{r}) \neq 0$ for some $\sigma \in \Sigma$.*

Proof. We prove the contrapositive: if $\mathcal{A}(w) = 0$ and $P_{w\sigma}(\boldsymbol{r}) = 0$ for each $\sigma \in \Sigma$, then $P_w(\boldsymbol{r}) = 0$. This immediately follows from the equation

$$P_w(\boldsymbol{r}) = \mathcal{A}(w) + \sum_{\sigma \in \Sigma} r_{\sigma,m+1}\, P_{w\sigma}(\boldsymbol{r})\,.$$

This equation is established from the definition of $P_w(\boldsymbol{r})$ as follows:

$$\begin{aligned}
P_w(\boldsymbol{r}) &= \boldsymbol{\alpha} M(w) \left(\sum_{i=m}^{n} \prod_{j=m+1}^{i} \sum_{\sigma \in \Sigma} r_{\sigma,j}\, M(\sigma) \right) \boldsymbol{\eta} \\
&= \mathcal{A}(w) + \boldsymbol{\alpha} M(w) \left(\sum_{i=m+1}^{n} \prod_{j=m+1}^{i} \sum_{\sigma \in \Sigma} r_{\sigma,j}\, M(\sigma) \right) \boldsymbol{\eta} \\
&= \mathcal{A}(w) + \sum_{\sigma \in \Sigma} r_{\sigma,m+1}\, \boldsymbol{\alpha} M(w\sigma) \left(\sum_{i=m+1}^{n} \prod_{j=m+2}^{i} \sum_{\sigma \in \Sigma} r_{\sigma,j}\, M(\sigma) \right) \boldsymbol{\eta} \\
&= \mathcal{A}(w) + \sum_{\sigma \in \Sigma} r_{\sigma,m+1}\, P_{w\sigma}(\boldsymbol{r})\,.
\end{aligned}$$

□

From Proposition 3.3 it is clear that the algorithm in Figure 2 generates a counterexample trace given $\boldsymbol{r}$ such that $P(\boldsymbol{r}) \neq 0$.

The algorithm in Figure 1 can be parallelised, yielding an **RNC** algorithm, as iterated products of matrices can be computed in **NC**. On the other hand, the algorithm in Figure 2 yields a counterexample, but apparently cannot be parallelised efficiently because the counterexample is produced incrementally.

3.2. **Algorithm Based on the Isolating Lemma.** We now develop a randomised $\mathbf{NC}^2$ procedure that can produce a counterexample in case of inequivalence. To this end we employ the Isolating Lemma of Mulmuley, Vazirani and Vazirani [20]. We use this lemma in a very similar way to [20], who are concerned with computing maximum matchings in graphs in **RNC**.

Lemma 3.4. *Let $\mathcal{F}$ be a family of subsets of a set $\{x_1, \ldots, x_N\}$. Suppose that each element x_i is assigned a weight w_i chosen independently and uniformly at random from $\{1, \ldots, 2N\}$. Define the weight of $S \in \mathcal{F}$ to be $\sum_{x_i \in S} w_i$. Then the probability that there is a unique minimum weight set in $\mathcal{F}$ is at least $1/2$.*

We will apply the Isolating Lemma in conjunction with Proposition 3.1 to decide zeroness of a $\mathbb{Q}$-weighted automaton $\mathcal{A}$. Suppose $\mathcal{A}$ has n states and alphabet Σ. Given $\sigma \in \Sigma$ and $1 \leq i \leq n$, choose a weight $w_{i,\sigma}$ independently and uniformly at random from the set $\{1, \ldots, 2|\Sigma|n\}$. Define the weight of a word $u = \sigma_1 \ldots \sigma_k$, $k \leq n$, to be $\mathrm{wt}(u) := \sum_{i=1}^{k} w_{i,\sigma_i}$. (The reader should not confuse this with the weight $\mathcal{A}(u)$ assigned to u by the automaton $\mathcal{A}$.) Then we obtain a univariate polynomial P from automaton $\mathcal{A}$ as follows:

$$P(x) = \sum_{k=0}^{n} \sum_{u \in \Sigma^k} \mathcal{A}(u) x^{\mathrm{wt}(u)} .$$

If $\mathcal{A}$ is equivalent to the zero automaton then clearly $P \equiv 0$. On the other hand, if $\mathcal{A}$ is non-zero, then by Proposition 3.1 the set $\mathcal{F} = \{u \in \Sigma^{\leq n} : \mathcal{A}(u) \neq 0\}$ is non-empty. Thus there is a unique minimum-weight word $u \in \mathcal{F}$ with probability at least $1/2$ by the Isolating Lemma. In this case P contains the monomial $x^{\mathrm{wt}(u)}$ with coefficient $\mathcal{A}(u)$ as its smallest-degree monomial. Thus $P \not\equiv 0$ with probability at least $1/2$.

It remains to observe that from the formula

$$P(x) = \boldsymbol{\alpha} \left(\sum_{i=0}^{n} \prod_{j=1}^{i} \sum_{\sigma \in \Sigma} M(\sigma) x^{w_{j,\sigma}} \right) \boldsymbol{\eta}$$

and the fact that iterated products of matrices of univariate polynomials can be computed in $\mathbf{NC}^2$ [10] we obtain an **RNC** algorithm for determining zeroness of $\mathbb{Q}$-weighted automata.

It is straightforward to extend the above algorithm to obtain an **RNC** procedure that not only decides zeroness of $\mathcal{A}$ but also outputs a word u such that $\mathcal{A}(u) \neq 0$ in case $\mathcal{A}$ is non-zero. Assume that $\mathcal{A}$ is non-zero and that the random choice of weights has isolated a unique minimum-weight word $u = \sigma_1 \ldots \sigma_k$ such that $\mathcal{A}(u) \neq 0$. To determine whether $\sigma \in \Sigma$ is the i-th letter of u we can increase the weight $w_{i,\sigma}$ by 1 while leaving all other weights unchanged and recompute the polynomial $P(x)$. Then σ is the i-th letter in u if and only if the minimum-degree monomial in P changes. All of these tests can be done independently, yielding an **RNC** procedure.

Theorem 3.5. *Given two $\mathbb{Q}$-weighted automata $\mathcal{A}$ and $\mathcal{B}$, there is an **RNC** procedure that determines whether or not $\mathcal{A}$ and $\mathcal{B}$ are equivalent and that outputs a word w with $\mathcal{A}(w) \neq \mathcal{B}(w)$ in case $\mathcal{A}$ and $\mathcal{B}$ are inequivalent.*

From a practical perspective, the algorithm is less efficient than those from the previous subsection, as it requires computations on univariate polynomials rather than on mere numbers.

4. Minimisation of $\mathbb{Q}$-Weighted Automata

A $\mathbb{Q}$-weighted automaton is *minimal* if there is no equivalent automaton with strictly fewer states. It is known that minimal automata are unique up to a change of basis [7]. In this section we give an **NC** algorithm to decide whether a given $\mathbb{Q}$-weighted automaton $\mathcal{A}$ is

minimal. We also give an **RNC** algorithm that computes a minimal automaton equivalent to a given $\mathbb{Q}$-weighted automaton $\mathcal{A}$.

4.1. **Deciding Minimality.** Let $\mathcal{A} = (n, \Sigma, M, \boldsymbol{\alpha}, \boldsymbol{\eta})$ be an automaton. Define the (infinite) matrix F to have rows indexed by Σ^* and columns indexed by $\{1, \ldots, n\}$, with the row indexed by $w \in \Sigma^*$ being the vector $\boldsymbol{\alpha}M(w)$. The *forward space* F is defined to be the row space of F. Similarly define the matrix B to have rows indexed by $\{1, \ldots, n\}$ and columns indexed by Σ^*, with the column indexed by $w \in \Sigma^*$ being the vector $M(w)\boldsymbol{\eta}$. The *backward space* B is defined to be the column space of B. The product $H = FB$ is called the *Hankel matrix*; it has rows and columns indexed by Σ^* with $H_{x,y} = \boldsymbol{\alpha}M(x)M(y)\boldsymbol{\eta} = \mathcal{A}(xy)$. By linear algebra we have $\operatorname{rank}(H) \leq \min\{\operatorname{rank}(F), \operatorname{rank}(B)\} \leq n$. A fundamental result [7] is that the above inequalities are tight precisely when $\mathcal{A}$ is minimal:

Proposition 4.1 (Carlyle and Paz)**.** *An automaton $\mathcal{A}$ with n states is minimal if and only if the Hankel matrix H has rank n.*

Using this result we show

Theorem 4.2. *Deciding whether a $\mathbb{Q}$-weighted automaton is minimal is in **NC**.*

Proof. To check that a given automaton $\mathcal{A} = (n, \Sigma, M, \boldsymbol{\alpha}, \boldsymbol{\eta})$ is minimal it suffices to verify that the associated Hankel matrix H has rank n. Since $H = FB$, this holds if and only if the matrices F and B both have rank n. We show how to check that F has rank n; the procedure for B is entirely analogous.

Let $\widetilde{F}$ be the sub-matrix of F obtained by retaining only those rows indexed by words in $\Sigma^{<n}$. By Proposition 3.1(i) we have $\operatorname{rank}(F) = \operatorname{rank}(\widetilde{F})$. Thus

$$\begin{aligned} \operatorname{rank}(F) = n &\Leftrightarrow \operatorname{rank}(\widetilde{F}) = n \\ &\Leftrightarrow \ker(\widetilde{F}) = \{0\} \\ &\Leftrightarrow \ker(\widetilde{F}^T\widetilde{F}) = \{0\} \\ &\Leftrightarrow \det(\widetilde{F}^T\widetilde{F}) \neq 0\,. \end{aligned}$$

The middle equivalence holds because for any vector $x \in \mathbb{Q}^n$, $\widetilde{F}^T\widetilde{F}x = 0$ implies $0 = x^T\widetilde{F}^T\widetilde{F}x = (\widetilde{F}x)^T\widetilde{F}x$, which in turn implies that $\widetilde{F}x = 0$.

Since determinants can be computed in **NC** it only remains to show that we can compute each entry of the $n \times n$ matrix $\widetilde{F}^T\widetilde{F}$ in **NC**. Let $\boldsymbol{e}_i \in \mathbb{Q}^n$ be the column vector with 1 in the i-th position and 0 in all other positions. Given $1 \leq i, j \leq n$ we have

$$\begin{aligned} (\widetilde{F}^T\widetilde{F})_{ij} &= \sum_{w \in \Sigma^{<n}} (\boldsymbol{\alpha}M(w)\boldsymbol{e}_i)(\boldsymbol{\alpha}M(w)\boldsymbol{e}_j) \\ &= \sum_{w \in \Sigma^{<n}} (\boldsymbol{\alpha} \otimes \boldsymbol{\alpha})(M(w) \otimes M(w))(\boldsymbol{e}_i \otimes \boldsymbol{e}_j) \\ &= (\boldsymbol{\alpha} \otimes \boldsymbol{\alpha}) \left(\sum_{k=0}^{n-1} \sum_{w \in \Sigma^k} (M(w) \otimes M(w)) \right) (\boldsymbol{e}_i \otimes \boldsymbol{e}_j) \\ &= (\boldsymbol{\alpha} \otimes \boldsymbol{\alpha}) \left(\sum_{k=0}^{n-1} \left(\sum_{\sigma \in \Sigma} (M(\sigma) \otimes M(\sigma)) \right)^k \right) (\boldsymbol{e}_i \otimes \boldsymbol{e}_j)\,. \end{aligned}$$

Algorithm Forward-Basis

Input: Automaton $\mathcal{A} = (n, \Sigma, M, \boldsymbol{\alpha}, \boldsymbol{\eta})$ and error parameter K

for i from 1 to n do
    choose a random vector $\boldsymbol{r}^{(i)} \in \{1, 2, \ldots, Kn\}^{\Sigma \times n}$
    $\boldsymbol{v}_i := \rho(\boldsymbol{r}^{(i)})$
let k be maximum such that $\{\boldsymbol{v}_1, \ldots, \boldsymbol{v}_k\}$ is linearly independent
return "$\{\boldsymbol{v}_1, \ldots, \boldsymbol{v}_k\}$ is a basis of F"

Figure 3: Algorithm for generating a basis of the forward space

But this last expression can be computed in **NC** since sums and matrix powers can be computed in **NC** [10]. □

4.2. **Minimising an Automaton.** Next we give an **RNC** algorithm to minimise a given automaton. The key idea is that we can compute a basis of the forward space F by generating random vectors in the space. We show that a randomly generated set of such vectors of cardinality equal to the dimension of F is likely to be a basis of F. We can likewise compute a basis of the backward space B. We give the construction of the forward space; the proof for the backward space is similar.

The construction involves an application of polynomial identity testing in similar manner to Section 3.1. Consider again a family of variables $\boldsymbol{x} = \{x_{\sigma,i} : \sigma \in \Sigma,\ 1 \le i \le n\}$ and associate the monomial $x_{w_1,1}x_{w_2,2}\ldots x_{w_k,k}$ with a word $w = w_1 w_2 \ldots w_k$. Then we define the row vector $\rho(\boldsymbol{x}) \in \mathbb{Q}[\boldsymbol{x}]^n$ by

$$\rho(\boldsymbol{x}) := \sum_{k=0}^{n} \sum_{w \in \Sigma^k} \boldsymbol{\alpha} M(w) \cdot x_{w_1,1} x_{w_2,2} \ldots x_{w_k,k}\,. \tag{4.1}$$

Note that evaluating $\rho(\boldsymbol{x})$ at a vector of rationals $\boldsymbol{r} = (r_{\sigma,i} : \sigma \in \Sigma,\ 1 \le i \le n)$ yields a vector $\rho(\boldsymbol{r})$ in the forward space F.

Proposition 4.3. *Let U be a proper subspace of F and let K be a positive integer. Then for $\boldsymbol{r}$ chosen uniformly at random from $\{1, \ldots, Kn\}^{\Sigma \times n}$ we have $\Pr(\rho(\boldsymbol{r}) \in \mathsf{U}) \le 1/K$.*

Proof. Pick a non-zero vector $\boldsymbol{v} \in \mathsf{F}$ that is orthogonal to U. Notice that the polynomial $\rho(\boldsymbol{x})\boldsymbol{v}^T$ is non-zero since the coefficient of the monomial corresponding to a word $w \in \Sigma^{<n}$ is $\boldsymbol{\alpha} M(w) \boldsymbol{v}^T$, and this is clearly non-zero for at least one w. Now $\rho(\boldsymbol{r}) \in \mathsf{U}$ only if $\rho(\boldsymbol{r})\boldsymbol{v}^T = 0$. Since $\rho(\boldsymbol{x})\boldsymbol{v}^T$ has degree at most n, it follows from Theorem 3.2 that $\Pr(\rho(\boldsymbol{r}) \in \mathsf{U})$ is at most $1/K$. □

The procedure to generate a basis for the forward space F is shown in Figure 3. The algorithm **Forward-Basis** necessarily returns a linearly independent set of vectors in the forward space. It only fails to output a basis if $\boldsymbol{v}_{m+1} \in \operatorname{span}\{\boldsymbol{v}_1, \ldots, \boldsymbol{v}_m\}$ for some $m < \dim(\mathsf{F})$. By Proposition 4.3 this happens with probability at most $1/K$ for any given m, so the total probability that **Forward-Basis** does not give a correct output is at most n/K. Thus, e.g., choosing $K = 3n$ we have an error probability of at most $1/3$.

It remains to observe that **Forward-Basis** can be made to run in $O(\log^2 n)$ parallel time. We perform the assignments $\boldsymbol{v}_i := \rho(\boldsymbol{r}^{(i)})$ for $i = 1, \ldots, n$ in parallel. As observed in Section 3.1, the computation of $\rho(\boldsymbol{r}^{(i)})$ involves an iterated matrix product, which can

be done in $O(\log^2 n)$ parallel time. We also check linear independence of $\{\boldsymbol{v}_1, \ldots, \boldsymbol{v}_k\}$ for $k = 1, \ldots, n$ in parallel. Each check involves computing the rank of an $k \times n$ matrix, which can again be done in $O(\log^2 n)$ parallel time [14].

Given bases of F and B, minimisation proceeds via a classical construction of Schützenberger [23]. We briefly recall this construction and show that it can be implemented in **NC** by making one call to algorithm **Forward-Basis** and one call to the corresponding backward version of this algorithm.

Let $\overrightarrow{n} \in \mathbb{N}$ and $\overrightarrow{F} \in \mathbb{Q}^{\overrightarrow{n} \times n}$ be such that the rows of $\overrightarrow{F}$ form a basis of the forward space F, with the first row of $\overrightarrow{F}$ being $\boldsymbol{\alpha}$. Similarly, let $\overleftarrow{n} \in \mathbb{N}$ and $\overleftarrow{B} \in \mathbb{Q}^{n \times \overleftarrow{n}}$ be such that the columns of $\overleftarrow{B}$ form a basis of the backward space B, with the first column of $\overleftarrow{B}$ being $\boldsymbol{\eta}$. Since $\mathsf{F}M(\sigma) \subseteq \mathsf{F}$ and $M(\sigma)\mathsf{B} \subseteq \mathsf{B}$ for all $\sigma \in \Sigma$, there exist maps $\overrightarrow{M} : \Sigma \to \mathbb{Q}^{\overrightarrow{n} \times \overrightarrow{n}}$ and $\overleftarrow{M} : \Sigma \to \mathbb{Q}^{\overleftarrow{n} \times \overleftarrow{n}}$ such that

$$\overrightarrow{F}M(\sigma) = \overrightarrow{M}(\sigma)\overrightarrow{F} \quad \text{and} \quad M(\sigma)\overleftarrow{B} = \overleftarrow{B}\overleftarrow{M}(\sigma) \quad \text{for all } \sigma \in \Sigma. \tag{4.2}$$

Call $\overrightarrow{\mathcal{A}} := (\overrightarrow{n}, \Sigma, \overrightarrow{M}, \boldsymbol{e}_1, \overrightarrow{F}\boldsymbol{\eta})$ a *forward reduction* of $\mathcal{A}$ with base $\overrightarrow{F}$ and similarly $\overleftarrow{\mathcal{A}} := (\overleftarrow{n}, \Sigma, \overleftarrow{M}, \boldsymbol{\alpha}\overleftarrow{B}, \boldsymbol{e}_1^T)$ a *backward reduction* of $\mathcal{A}$ with base $\overleftarrow{B}$.

Proposition 4.4 ([23])**.** *Let $\mathcal{A}$ be an automaton. Then $\overleftrightarrow{\mathcal{A}}$ is minimal and equivalent to $\mathcal{A}$.*

Theorem 4.5. *There is an **RNC** algorithm that transforms a given automaton into an equivalent minimal automaton.*

Proof. Let $\mathcal{A} = (n, \Sigma, M, \boldsymbol{\alpha}, \boldsymbol{\eta})$ be an automaton. We have already shown that we can compute in randomised **NC** a matrix $\overrightarrow{F}$ whose rows form a basis of the forward space of $\mathcal{A}$. Given $\overrightarrow{F}$ we can compute the forward reduction $\overrightarrow{\mathcal{A}}$ in **NC** since each transition matrix $\overrightarrow{M}(\sigma)$ is uniquely defined as the solution to the linear system of equations (4.2). Using the same reasoning we can compute $\overleftrightarrow{\mathcal{A}}$ from $\overrightarrow{\mathcal{A}}$ in randomised **NC**. This is the minimal automaton that we seek. □

5. Probabilistic Reward Automata

In this section we consider *probabilistic reward automata*, which extend Rabin's probabilistic automata [21] with rewards on transitions. The resulting notion can be seen as a type of partially observable Markov Decision Process [4]. A similar model has been investigated from the point of view of language theory in [8]. Rewards are allowed to be negative, in which case they can be seen as costs. In Example 5.5 we use costs to record the passage of time in an encryption protocol.

A *Probabilistic Reward Automaton* is a tuple $\mathcal{A} = (n, s, \Sigma, M, R, \boldsymbol{\alpha}, \boldsymbol{\eta})$, where $n \in \mathbb{N}$ is the number of states; $s \in \mathbb{N}$ is the number of types of reward; Σ is a finite alphabet, $M(\sigma)$ is an $n \times n$ rational sub-stochastic matrix for each $\sigma \in \Sigma$; $R(\sigma)$ is an $n \times n$ matrix with entries in $\{-1, 0, 1\}^s$ for each $\sigma \in \Sigma$; $\boldsymbol{\alpha}$ is an n-dimensional rational stochastic row vector; $\boldsymbol{\eta}$ is a rational n-dimensional column vector with all entries lying in the interval $[0, 1]$. We think of $M(\sigma)$ as the transition matrix, $R(\sigma)$ as the reward matrix, $\boldsymbol{\alpha}$ as the initial-state vector, and $\boldsymbol{\eta}$ as the final-state vector.

The total reward of a run is the sum of the rewards along all its transitions. The expected reward of a word is the sum of the rewards of all runs over that word, weighted by

their respective probabilities. Formally, given a word $w = w_1, \ldots, w_k$ and a path of states $p = p_0, \ldots, p_k$, the probability and total reward of the path are respectively defined by

$$\Pr(p) \;=\; \boldsymbol{\alpha}_{p_0} \left(\prod_{i=1}^{k} M(\sigma)_{p_{i-1},p_i} \right) \boldsymbol{\eta}_{p_k} \quad \text{and} \quad \text{Reward}(p) \;=\; \sum_{i=1}^{k} R(w_i)_{p_{i-1},p_i} \,.$$

The *value* of the word w is the expected reward over all runs:

$$\mathcal{A}(w) \;=\; \sum_{p \in \{1,\ldots,n\}^{k+1}} \Pr(p) \cdot \text{Reward}(p) \,. \tag{5.1}$$

5.1. **Expectation Equivalence.** Two probabilistic reward automata $\mathcal{A}$ and $\mathcal{B}$ over the same alphabet Σ are defined to be *equivalent in expectation* if $\mathcal{A}(w) = \mathcal{B}(w)$ for all words $w \in \Sigma^*$. In this section we give a simple reduction of the equivalence problem for probabilistic reward automata to the equivalence problem for $\mathbb{Q}$-weighted automata. The idea is to combine transition probabilities and rewards in a single matrix. Without loss of generality we consider automata with a single type of reward; the general problem can be reduced to this by considering each component separately.

Let $\mathcal{A} = (n, \Sigma, M, R, \boldsymbol{\alpha}, \boldsymbol{\eta})$ be a probabilistic reward automaton. We define a $\mathbb{Q}$-weighted automaton $\mathcal{B} = (2n, \Sigma, M', \boldsymbol{\alpha}', \boldsymbol{\eta}')$ such that $\mathcal{A}(w) = \mathcal{B}(w)$ for each word $w \in \Sigma^*$. First we introduce the following matrices:

$$A = \begin{bmatrix} 1 & 0 \end{bmatrix} \qquad E = \begin{bmatrix} 0 \\ 1 \end{bmatrix} \qquad C = \begin{bmatrix} 0 & 1 \\ 0 & 0 \end{bmatrix}$$

We also write I_n for the $n \times n$ identity matrix. Now we define

$$\begin{aligned} \boldsymbol{\alpha}' &:= \boldsymbol{\alpha} \otimes A \\ \boldsymbol{\eta}' &:= \boldsymbol{\eta} \otimes E \\ M'(\sigma) &:= (M(\sigma) \otimes I_2) + ((M(\sigma) \odot R(\sigma)) \otimes C) \end{aligned}$$

where $\otimes$ denotes Kronecker product and $\odot$ denotes Hadamard product (cf. Section 2.2).

Proposition 5.1. *$\mathcal{A}(w) = \mathcal{B}(w)$ for all words $w \in \Sigma^*$.*

Proof. We show by induction that for all words $w \in \Sigma^*$ we have

$$M'(w) = (M(w) \otimes I_2) + \left(\sum_{\substack{w', w'' \\ w = w' a w''}} (M(w')(M(a) \odot R(\sigma))M(w'')) \otimes C \right). \tag{5.2}$$

The base case, $w = \varepsilon$, is clear. For the induction step we have

$$\begin{aligned} M'(w\sigma) &= M'(w)M'(\sigma) \\ &= (M(w) \otimes I_2)(M(\sigma) \otimes I_2) + (M(w) \otimes I_2)((M(\sigma) \odot R(\sigma)) \otimes C) \\ &\quad + \left(\sum_{\substack{w', w'' \\ w = w' a w''}} (M(w')(M(a) \odot R(\sigma))M(w'')) \otimes C \right) (M(\sigma) \otimes I_2) \\ &\quad + \left(\sum_{\substack{w', w'' \\ w = w' a w''}} (M(w')(M(a) \odot R(\sigma))M(w'')) \otimes C \right) ((M(\sigma) \odot R(\sigma)) \otimes C) \end{aligned}$$

But using Proposition 2.1 and the identity $C^2 = 0$, the above expression simplifies to

$$(M(w\sigma) \otimes I_2) + \left(\sum_{\substack{w',w'' \\ w\sigma = w'aw''}} (M(w')(M(a) \odot R(\sigma))M(w'')) \otimes C \right).$$

This completes the induction step.

Using Proposition 2.1 and the fact that $AE = 0$ and $ACE = I_1$ it follows from (5.2) that

$$\begin{aligned} \mathcal{B}(w) = \boldsymbol{\alpha}' M'(w) \boldsymbol{\eta}' &= \sum_{\substack{w',w'' \\ w = w'aw''}} \boldsymbol{\alpha}(M(w')(M(a) \odot R(a))M(w''))\boldsymbol{\eta} \\ &= \sum_{i=1}^{k} \sum_{p \in \{1,\dots,n\}^{k+1}} \alpha_{p_0} \left(\prod_{j=1}^{k} M(w_j)_{p_{j-1},p_j} R(w_i)_{p_{i-1},p_i} \right) \eta_{p_k} . \end{aligned}$$

But the equivalence of the above expression and (5.1) follows from distributivity of multiplication over addition. □

Corollary 5.2. *Expectation equivalence of probabilistic reward automata can be decided in* ***NC****. Moreover there is an* ***RNC*** *procedure that determines whether or not two automata are equivalent and outputs a word on which they differ in case they are inequivalent.*

Proof. The first part follows by combining Proposition 5.1 with the **NC** algorithm for $\mathbb{Q}$-weighted automaton equivalence in [29]. The second part follows by combining Proposition 5.1 with Theorem 3.5. □

5.2. **Distribution Equivalence.** Two probabilistic reward automata are called *distribution equivalent* if they induce identical distributions on rewards for each input word $w \in \Sigma^*$. We formalise this notion by translating probabilistic reward automata into $\mathbb{Q}$-weighted automata over the field $\mathbb{F} = \mathbb{Q}(t_1, t_1^{-1}, \dots, t_s, t_s^{-1})$ of rational Laurent functions, as defined in Section 2. We consider ε-transitions in this section because they are convenient for applications (cf. Example 5.4) and because we cannot rely on existing ε-elimination results in the presence of rewards.

Let $\mathcal{A} = (n, s, \Sigma, M, R, \boldsymbol{\alpha}, \boldsymbol{\eta})$ be a probabilistic reward automaton, where $\varepsilon \in \Sigma$. To make ε-elimination more straightforward, we assume that the transition matrix $M(\varepsilon)$ has no recurrent states, i.e., that its spectral radius is strictly less than one. We now define an $\mathbb{F}$-weighted automaton $\mathcal{A}' = (n, \Sigma, M', \boldsymbol{\alpha}, \boldsymbol{\eta})$ as follows. For $1 \leq i, j \leq n$, let $M'(\sigma)_{i,j} = at_1^{k_1} \dots, t_s^{k_s}$, where $M(\sigma)_{i,j} = a$ and $R(\sigma)_{i,j} = (k_1, \dots, k_s)$. We extend M' to a map $M' : \Sigma^* \to \mathbb{F}^{n \times n}$ by defining

$$M'(w) := M'(\varepsilon)^* M'(w_1) M'(\varepsilon)^* \cdots M'(w_m) M'(\varepsilon)^* \tag{5.3}$$

for a word $w = w_1 \dots w_m$. Our convention on ε-transitions implies that $\det(I - M'(\varepsilon)) \neq 0$ and therefore, by Proposition 2.2, that $M'(\varepsilon)^*$ is well-defined and has entries whose numerators and denominators are Laurent polynomials with degree bound sn. It follows that the entries of $M'(w)$ have degree bound $(sn + 1)m$.

Two probabilistic reward automata $\mathcal{B}, \mathcal{C}$ over the same alphabet Σ and with the same number of reward types are said to be *equivalent* if the corresponding $\mathbb{F}$-weighted automata $\mathcal{B}'$ and $\mathcal{C}'$ are equivalent, i.e., $\mathcal{B}'(w) = \mathcal{C}'(w)$ for all words $w \in \Sigma^*$. Now Proposition 3.1 implies that equivalence for $\mathbb{F}$-weighted automata is decidable, but the algorithms of Schützenberger [23] and Tzeng [28] do not yield polynomial-time procedures in our case because the complexity of solving systems of linear equations over the field $\mathbb{Q}(t_1, t_1^{-1}, \ldots, t_s, t_s^{-1})$ is not polynomial in s (indeed the solution need not have length exponential in s). However, it not difficult to give a randomised polynomial-time algorithm to decide equivalence of probabilistic reward automata.

Let $\mathcal{A}'$ be the $\mathbb{F}$-weighted automaton corresponding to a probabilistic reward automaton $\mathcal{A}$ with n states. For each word $w \in \Sigma^*$ of length at most n we have a rational function $\mathcal{A}'(w)$ whose numerator and denominator are polynomials of degree at most $d := (sn+1)n$, as observed above. Now consider the set $R := \{1, 2, \ldots, 2d\}^s$. Suppose that we pick $\boldsymbol{r} \in R$ uniformly at random. Denote by $\mathcal{A}'(w)(\boldsymbol{r})$ the result of substituting $\boldsymbol{r}$ for the formal variables $t_1, \ldots, t_s$ in the rational function $\mathcal{A}'(w)$. Clearly if $\mathcal{A}'$ is a zero automaton then $\mathcal{A}'(w)(\boldsymbol{r}) = 0$ for all $\boldsymbol{r} \in R$. On the other hand, if $\mathcal{A}'$ is non-zero then by Proposition 3.1 there exists a word $w \in \Sigma^*$ of length at most n such that $\mathcal{A}'(w) \not\equiv 0$. Since the degree of the rational expression $A'(w)$ is at most d it follows from the Schwartz-Zippel theorem [11, 24, 30] that the probability that $\mathcal{A}(w)(\boldsymbol{r}) = 0$ is at most $1/2$.

Thus our randomised procedure is to pick $\boldsymbol{r} \in R$ uniformly at random and to check whether $\mathcal{A}(w)(\boldsymbol{r}) = 0$ for some $w \in \Sigma^*$. To perform this final check we show that there is a $\mathbb{Q}$-weighted automaton $\mathcal{B}$ such that $\mathcal{A}'(w)(\boldsymbol{r}) = \mathcal{B}(w)$ for all $w \in \Sigma^*$. Then check $\mathcal{B}$ for zeroness using, e.g., Tzeng's algorithm [28]. The automaton $\mathcal{B}$ has the form $\mathcal{B} = (n^{(\mathcal{B})}, \Sigma, M^{(\mathcal{B})}, \boldsymbol{\alpha}^{(\mathcal{B})}, \boldsymbol{\eta}^{(\mathcal{B})})$, where $n^{(\mathcal{B})} = n$, $\boldsymbol{\alpha}^{(\mathcal{B})} = \boldsymbol{\alpha}$, $\boldsymbol{\eta}^{(\mathcal{B})} = \boldsymbol{\eta}$ and $M^{(\mathcal{B})}(\sigma) = M(\sigma)(\boldsymbol{r})$ for all $\sigma \in \Sigma$.

Theorem 5.3. *There is an **RNC** procedure that determines whether or not two probabilistic reward automata are distribution equivalent, and which outputs a word on which they differ in case they are inequivalent.*

Example 5.4. We consider probabilistic programs that randomly increase and decrease a single counter (initialised with 0) so that upon termination the counter has a random value $X \in \mathbb{Z}$. The programs should be such that X is a random variable with $X = Y - Z$ where Y and Z are independent random variables with a geometric distribution with parameters $p = 1/2$ and $p = 1/3$, respectively. (By that we mean that $\Pr(Y = k) = (1-p)^k p$ for $k \in \{0, 1, \ldots\}$, and similarly for Z.) Figure 4 shows code in the syntax of the APEX tool [16].

The program on the left consecutively runs two while loops: it first increments the counter according to a geometric distribution with parameter $1/2$ and then decrements the counter according to a geometric distribution with parameter $1/3$, so that the final counter value is distributed as desired. The program on the right is more efficient in that it runs only one of two while loops, depending on a single coin flip at the beginning. It may not be obvious though that the final counter value follows the same distribution as in the left program. We used the APEX tool to translate the programs to the probabilistic reward automata $\mathcal{B}$ and $\mathcal{C}$ shown in Figure 5. Here each counter increment corresponds to a reward of 1 and each counter decrement to a reward of -1. Since the input alphabets are empty, it suffices to consider the input word ε when comparing $\mathcal{B}$ and $\mathcal{C}$ for equivalence. If we construct the difference automaton $\mathcal{A} = (5, 1, \emptyset, M, \boldsymbol{\alpha}, \boldsymbol{\eta})$ and invert the matrix of

```
inc:com, dec:com |-
  var%2 flip;
  flip := 0;
  while (flip = 0) do {
    flip := coin[0:1/2,1:1/2];
    if (flip = 0) then {
     inc;
    };
  };
  flip := 0;
  while (flip = 0) do {
    flip := coin[0:2/3,1:1/3];
    if (flip = 0) then {
     dec;
    };
  }
:com
```

```
inc:com, dec:com |-
  var%2 flip;
  flip := coin[0:1/2,1:1/2];
  if (flip = 0) then {
    while (flip = 0) do {
      flip := coin[0:1/2,1:1/2];
      if (flip = 0) then {
       inc;
      };
    };
  } else {
    flip := 0;
    while (flip = 0) do {
      dec;
      flip := coin[0:2/3,1:1/3];
    };
  }
:com
```

Figure 4: Two APEX programs for producing a counter that is distributed as the difference between two geometrically distributed random variables.

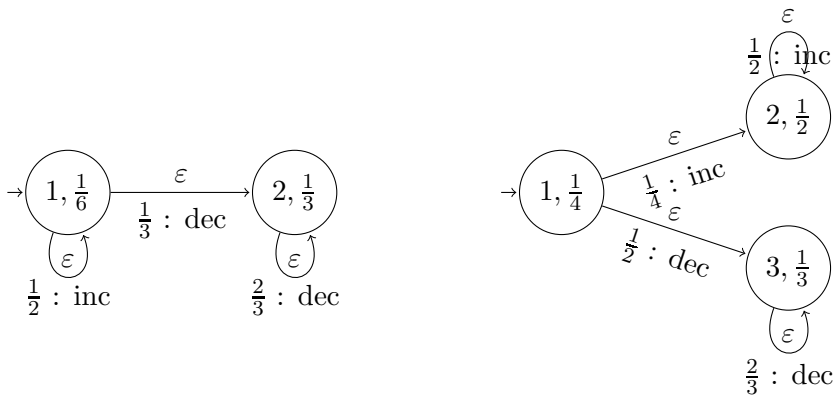

Figure 5: Automata produced from the code in Figure 4. The states are labelled with their number and their "acceptance probability" ($\boldsymbol{\eta}$-weight). In both automata, state 1 is the only initial state ($\boldsymbol{\alpha}_1 = 1$ and $\boldsymbol{\alpha}_i = 0$ for $i \neq 1$). The transitions are labelled with the input symbol ε, with a probability (weight) and a cost.

polynomials $I - M(\varepsilon)$, we obtain

$$\mathcal{A}(\varepsilon)(x) = \left(\frac{2}{x-2}, \frac{2}{(3x-2)(x-2)}, 1, \frac{-x}{2(x-2)}, \frac{3}{2(3x-2)}\right)\boldsymbol{\eta} \equiv 0\,,$$

which proves equivalence of $\mathcal{B}$ and $\mathcal{C}$. Notice that the actual algorithm would not compute $\mathcal{A}(\varepsilon)(x)$ as a polynomial, but it would compute $\mathcal{A}(\varepsilon)(r)$ only for a few concrete values $r \in \mathbb{Q}$.

Example 5.5. RSA [22] is a widely-used cryptographic algorithm. Popular implementations of the RSA algorithm have been shown to be vulnerable to timing attacks that reveal private keys [17, 6]. The preferred countermeasures are blinding techniques that randomise certain aspects of the computation, which are described in, e.g., [17]. We model the timing behaviour of the RSA algorithm using probabilistic cost automata, where costs encode time. These automata are produced by APEX, and are then used to check for timing leaks with and without blinding.

At the heart of RSA decryption is a modular exponentiation, which computes the value $m^d \mod N$ where $m \in \{0, \ldots, N-1\}$ is the encrypted message, $d \in \mathbb{N}$ is the private decryption exponent and $N \in \mathbb{N}$ is a modulus. An attacker wants to find out d. We model RSA decryption in APEX by implementing modular exponentiation by iterative squaring (see Figure 6). We consider the situation where the attacker is able to control the message m, and tries to derive d by observing the runtime distribution over different messages m. Following [17] we assume that the running time of multiplication depends on the operand values (because a source-level multiplication typically corresponds to a cascade of processor-level multiplications). By choosing the 'right' input message m, an attacker can observe which private keys are most likely.

We consider two blinding techniques mentioned in Kocher [17]. The first one is base blinding, i.e., the message is multiplied by r^d before exponentiation where d is a random number, which gives a result that can be fixed by dividing by r but makes it impossible for the attacker to control the basis of the exponentiation. The second one is exponent blinding, which adds a multiple of the group order $\varphi(N)$ of $\mathbb{Z}/N\mathbb{Z}$ to the exponent, which doesn't change the result of the exponentiation[1] but changes the timing behaviour.

Figure 7 shows the automaton for $N = 10$, and private key $0, 1, 0, 1$ with message blinding enabled. The APEX program is given in Figure 6.

We investigate the effectiveness of blinding. Two private keys are indistinguishable if the resulting automata are equivalent. The more keys are indistinguishable the safer the algorithm. We analyse which private keys are identified by plain RSA, RSA with a blinded message and RSA with blinded exponent.

For example, in plain RSA, the following keys $0, 1, 0, 1$ and $1, 0, 0, 1$ are indistinguishable, keys $0, 1, 1, 0$ and $0, 0, 1, 1$ are indistinguishable with base blinding, lastly $1, 0, 0, 1$ and $1, 0, 1, 1$ are equivalent only with exponent blinding. Overall 9 different keys are distinguishable with plain RSA, 7 classes with base blinding and 4 classes with exponent blinding.

6. Pushdown Automata and Arithmetic Circuits

In a visibly pushdown automaton [3] the stack operations are determined by the input word. Consequently VPA have a more tractable language theory than ordinary pushdown automata. The main result of this section shows that the equivalence problem for $\mathbb{Q}$-weighted VPA is logspace equivalent to the problem **ACIT** of determining whether a polynomial represented by an arithmetic circuit is identically zero.

[1] Euler's totient function φ satisfies $a^{\varphi(N)} \equiv 1 \mod N$ for all $a \in \mathbb{Z}$.

```
const N := 10;    // modulus
const Bits := 4 ; // number of bits of the key

m :int%N, inc:com |-
var%2 exponent[Bits] = [0,1,0,1];
com power(x:int%N) {
   var%N s := 1;
   var%N R;
   for(var%(Bits + 1) k; k < Bits; ++k) do {
      R:=s;
      if(exponent[k]) then {
         R := R*x;
         if(5<=R) then { inc; inc } else { inc }
      }
      s := R*R;
   }
}
var%N message := m*rand[N]; // blinding
power(message) : com
```

Figure 6: APEX code for RSA.

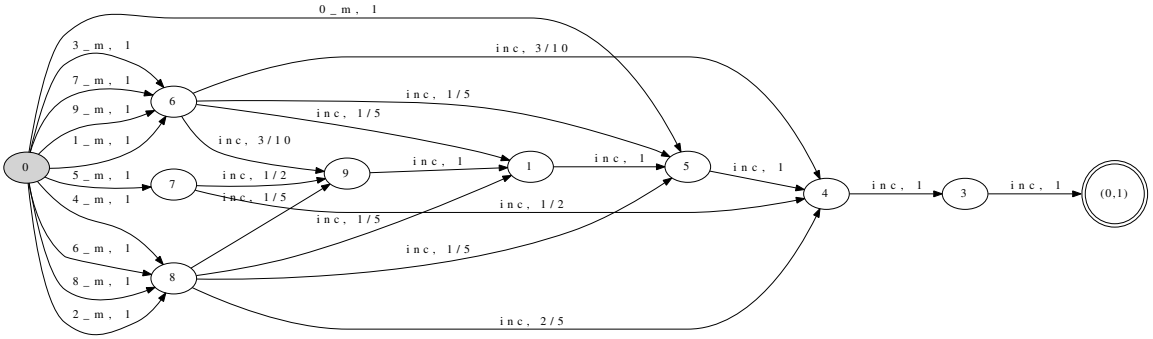

Figure 7: Modeling RSA decryption with APEX.

A *visibly pushdown alphabet* $\Sigma = \Sigma_c \cup \Sigma_r \cup \Sigma_{int}$ consists of a finite set of *calls* Σ_c, a finite set of *returns* Σ_r, and a finite set of *internal actions* Σ_{int}. A visibly pushdown automaton over alphabet Σ is restricted so that it pushes onto the stack when it reads a call, pops the stack when it reads a return, and leaves the stack untouched when reading internal actions. Due to this restriction visibly pushdown automata only accept words in which calls and returns are appropriately matched. Define the set of *well-matched words* to be $\bigcup_{i\in\mathbb{N}} L_i$, where $L_0 = \Sigma_{int} + \{\varepsilon\}$ and $L_{i+1} = \Sigma_c L_i \Sigma_r + L_i L_i$.

A $\mathbb{Q}$-*weighted visibly pushdown automaton* on alphabet Σ is a tuple $\mathcal{A} = (n, \boldsymbol{\alpha}, \boldsymbol{\eta}, \Gamma, M)$, where n is the number of *states*, $\boldsymbol{\alpha}$ is an n-dimensional *initial* (row) vector, $\boldsymbol{\eta}$ is an n-dimensional *final* (column) vector, Γ is a finite *stack alphabet*, and $M = (M_c, M_r, M_{int})$ is a

tuple of *matrix-valued transition functions* with types $M_c : \Sigma_c \times \Gamma \to \mathbb{Q}^{n\times n}$, $M_r : \Sigma_r \times \Gamma \to \mathbb{Q}^{n\times n}$ and $M_{int} : \Sigma_{int} \to \mathbb{Q}^{n\times n}$. If $a \in \Sigma_c$ and $\gamma \in \Gamma$ then $M_c(a,\gamma)_{i,j}$ gives the weight of an a-labelled transition from state i to state j that pushes γ on the stack. If $a \in \Sigma_r$ and $\gamma \in \Gamma$ then $M_r(a,\gamma)_{i,j}$ gives the weight of an a-labelled transition from state i to j that pops γ from the stack.

For each well-matched word $u \in \Sigma^*$ we define an $n \times n$ rational matrix $M^{(\mathcal{A})}(u)$ whose (i,j)-th entry denotes the total weight of all paths from state i to state j along input u. The definition of $M^{(\mathcal{A})}(u)$ follows the inductive definition of well-matched words. The base cases are $M^{(\mathcal{A})}(\varepsilon) = I$ and $M^{(\mathcal{A})}(a)_{i,j} = M_{int}(a)_{i,j}$. The inductive cases are

$$M^{(\mathcal{A})}(uv) = M^{(\mathcal{A})}(u) \cdot M^{(\mathcal{A})}(v) \tag{6.1}$$

$$M^{(\mathcal{A})}(aub) = \sum_{\gamma\in\Gamma} M_c(a,\gamma) \cdot M^{(\mathcal{A})}(u) \cdot M_r(b,\gamma)\,, \tag{6.2}$$

for $a \in \Sigma_c$, $b \in \Sigma_r$.

The weight assigned by $\mathcal{A}$ to a well-matched word w is defined as $\mathcal{A}(w) := \boldsymbol{\alpha} M^{(\mathcal{A})}(u) \boldsymbol{\eta}$. We say that two $\mathbb{Q}$-weighted VPA $\mathcal{A}$ and $\mathcal{B}$ are *equivalent* if for each well-matched word w we have $\mathcal{A}(w) = \mathcal{B}(w)$.

An *arithmetic circuit* is a finite directed acyclic multigraph whose vertices, called *gates*, have indegree 0 or 2. Vertices of indegree 0 are called *input gates* and are labelled with a constant 0 or 1, or a variable from the set $\{x_i : i \in \mathbb{N}\}$. Vertices of indegree 2 are called *internal gates* and are labelled with one of the arithmetic operations $+$, $*$ or $-$. We assume that there is a unique gate with outdegree 0 called the *output*. Note that C is a multigraph, so there can be two edges between a pair of gates, i.e., both inputs to a given gate can lead from the same source. We call a circuit *variable-free* if all inputs gates are labelled 0 or 1.

The *Arithmetic Circuit Identity Testing (**ACIT**)* problem asks whether the output of a given circuit is equal to the zero polynomial. **ACIT** is known to be in **coRP** but it remains open whether there is a polynomial or even sub-exponential algorithm for this problem [1]. Utilising the fact that a variable-free arithmetic circuit of size $O(n)$ can compute 2^{2^n}, Allender *et al.* [1] give a logspace reduction of the general **ACIT** problem to the special case of variable-free circuits. Henceforth we assume without loss of generality that all circuits are variable-free. Furthermore we recall that **ACIT** can be reformulated as the problem of deciding whether two variable-free circuits using only the arithmetic operations $+$ and $*$ compute the same number [1].

We have the following proposition:

Proposition 6.1. ACIT *is logspace reducible to the equivalence problem for $\mathbb{Q}$-weighted visibly pushdown automata.*

Proof. Let C and C' be two circuits over basis $\{+,*\}$. Without loss of generality we assume that in each circuit the inputs of a depth-i gate both have depth $i+1$, $+$-nodes have even depth, $*$-nodes have odd depth, and input nodes all have the same depth d. Notice that in either circuit any path from an input gate to an output gate has length d.

We define two automata $\mathcal{A}$ and $\mathcal{A}'$ that are equivalent if and only if C and C' have the same output. Both automata are defined over the alphabet $\{c, r, \iota\}$, with c a call, r a return and ι an internal event. We explain how $\mathcal{A}$ arises from C; the definition of $\mathcal{A}'$ is entirely analogous.

Suppose that C has set of gates $\{g_0, g_1, \ldots, g_n\}$, with g_0 the output gate. For each gate g_i of C we include a state s_i of $\mathcal{A}$ and a stack symbol γ_i. The initial state of $\mathcal{A}$ is s_0, and all states are accepting. The transitions of $\mathcal{A}$ are defined as follows:

- For each $+$-gate $g_i := g_j + g_k$ in C we include an internal transition from s_i that goes to s_j with probability $1/2$ and to s_k with probability $1/2$.
- For each $*$-gate $g_i := g_j * g_k$ we include a probability-1 call transition from s_i to s_j that pushes γ_k onto the stack.
- An input gate g_i with label 0 contributes no transitions.
- For each input gate g_i with label 1 and each stack symbol γ_j, we include a return transition from s_i that pops γ_j off the stack and ends in state s_j with probability 1.

Recall that acceptance is by empty stack and final state. By construction $\mathcal{A}$ only accepts a single word, as we now explain. Define a sequence of words $w_n \in \{c, r, \iota\}^*$ by $w_0 = \iota$, $w_{n+1} = \iota w_n$ for n even, and $w_{n+1} = c w_n r w_n$ for n odd. Furthermore, write $M_0 = 1$, $M_{n+1} = 2M_n$ for n even, and $M_{n+1} = M_n^2$ for n odd. Then $\mathcal{A}$ accepts w_d with probability N/M_d, where d is the depth of the circuit C and N is output of C. All other words are accepted with probability 0. We conclude that C and C' have the same value if and only if $\mathcal{A}$ and $\mathcal{A}'$ are equivalent. □

In the remainder of this section we give a converse reduction: from equivalence of $\mathbb{Q}$-weighted VPA to **ACIT**. The following result gives a decision procedure for the equivalence of two $\mathbb{Q}$-weighted VPA $\mathcal{A}$ and $\mathcal{B}$.

Proposition 6.2. *$\mathcal{A}$ is equivalent to $\mathcal{B}$ if and only if $\mathcal{A}(w) = \mathcal{B}(w)$ for all words $w \in L_{n^2}$, where n is the sum of the number of states of $\mathcal{A}$ and the number of states of $\mathcal{B}$.*

Proof. Recall that for each balanced word $u \in \Sigma^*$ we have rational matrices $M^{(\mathcal{A})}(u)$ and $M^{(\mathcal{B})}(u)$ giving the respective state-to-state transition weights of $\mathcal{A}$ and $\mathcal{B}$ on reading u. These two families of matrices can be combined into a single family

$$\mathcal{M} = \left\{ \begin{pmatrix} M^{(\mathcal{A})}(u) & \mathbf{0} \\ \mathbf{0} & M^{(\mathcal{B})}(u) \end{pmatrix} : u \text{ well-matched} \right\}$$

of $n \times n$ matrices. Let us also write $\mathcal{M}_i$ for the subset of $\mathcal{M}$ generated by those well-matched words $u \in L_i$.

Let $\boldsymbol{\alpha}^{(\mathcal{A})}, \boldsymbol{\eta}^{(\mathcal{A})}$ and $\boldsymbol{\alpha}^{(\mathcal{B})}, \boldsymbol{\eta}^{(\mathcal{B})}$ be the respective initial and final-state vectors of $\mathcal{A}$ and $\mathcal{B}$. Then $\mathcal{A}$ is equivalent to $\mathcal{B}$ if and only if

$$\begin{pmatrix} \boldsymbol{\alpha}^{(\mathcal{A})} & \boldsymbol{\alpha}^{(\mathcal{B})} \end{pmatrix} M \begin{pmatrix} \boldsymbol{\eta}^{(\mathcal{A})} \\ -\boldsymbol{\eta}^{(\mathcal{B})} \end{pmatrix} = 0 \tag{6.3}$$

for all $M \in \mathcal{M}$. It follows that $\mathcal{A}$ is equivalent to $\mathcal{B}$ if and only if (6.3) holds for all M in $\mathrm{span}(\mathcal{M})$, where the span is taken in the rational vector space of $n \times n$ rational matrices. But $\mathrm{span}(\mathcal{M}_i)$ is an ascending sequence of vector spaces:

$$\mathrm{Span}(\mathcal{M}_0) \subseteq \mathrm{Span}(\mathcal{M}_1) \subseteq \mathrm{Span}(\mathcal{M}_2) \subseteq \ldots$$

It follows from a dimension argument that this sequence stops in at most n^2 steps and we conclude that $\mathrm{span}(\mathcal{M}) = \mathrm{span}(\mathcal{M}_{n^2})$. □

Proposition 6.3. *Given a $\mathbb{Q}$-weighted visibly pushdown automaton $\mathcal{A}$ and $n \in \mathbb{N}$ one can compute in logarithmic space a circuit that represents $\sum_{w \in L_{n^2}} \mathcal{A}(w)$.*

Proof. From the definition of the language L_i and the family of matrices $M^{(\mathcal{A})}$ we have:

$$\begin{aligned}\sum_{w \in L_{i+1}} M^{(\mathcal{A})}(w) &= \sum_{a \in \Sigma_c} \sum_{b \in \Sigma_r} \sum_{\gamma \in \Gamma} M^{(\mathcal{A})}(a,\gamma) \left(\sum_{u \in L_i} M^{(\mathcal{A})}(u) \right) M^{(\mathcal{A})}(b,\gamma) \\ &\quad + \left(\sum_{u \in L_i} M^{(\mathcal{A})}(u) \right) \left(\sum_{u \in L_i} M^{(\mathcal{A})}(u) \right) .\end{aligned}$$

The above equation implies that we can compute in logarithmic space a circuit that represents $\sum_{w \in L_n} M^{(\mathcal{A})}(w)$. The result of the proposition immediately follows by premultiplying by the initial state vector and postmultiplying by the final state vector. □

A key property of $\mathbb{Q}$-weighted VPA is their closure under product.

Proposition 6.4. *Given $\mathbb{Q}$-weighted VPA $\mathcal{A}$ and $\mathcal{B}$ on the same alphabet Σ one can define a* synchronous-product automaton*, denoted $\mathcal{A} \otimes \mathcal{B}$, such that $(\mathcal{A} \otimes \mathcal{B})(w) = \mathcal{A}(w)\mathcal{B}(w)$ for all $w \in \Sigma^*$.*

Proof. The proof exploits the fact that the stack height is determined by the input word, so the respective stacks of $\mathcal{A}$ and $\mathcal{B}$ operating in parallel can be simulated in a single stack.

Let $\mathcal{A} = (n^{(\mathcal{A})}, \Sigma, \Gamma^{(\mathcal{A})}, M^{(\mathcal{A})}, \boldsymbol{\alpha}^{(\mathcal{A})}, \boldsymbol{\eta}^{(\mathcal{A})})$ and $\mathcal{B} = (n^{(\mathcal{B})}, \Sigma, \Gamma^{(\mathcal{A})}, M^{(\mathcal{B})}, \boldsymbol{\alpha}^{(\mathcal{B})}, \boldsymbol{\eta}^{(\mathcal{B})})$. We define a product automaton $\mathcal{C}$. Note that since the stack height is determined by the input word we can simulate the respective stacks of $\mathcal{A}$ and $\mathcal{B}$ using a single stack in $\mathcal{C}$ whose alphabet is the product of the respective stack alphabets of $\mathcal{A}$ and $\mathcal{B}$.

The number of states of $\mathcal{C}$ is $n^{(\mathcal{A})} \cdot n^{(\mathcal{B})}$. The initial vector $\boldsymbol{\alpha}^{(\mathcal{C})}$ in the vector $\boldsymbol{\alpha}^{(\mathcal{A})} \otimes \boldsymbol{\alpha}^{(\mathcal{B})}$ and the final vector $\boldsymbol{\eta}^{(\mathcal{C})}$ is $\boldsymbol{\eta}^{(\mathcal{A})} \otimes \boldsymbol{\eta}^{(\mathcal{B})}$. The stack alphabet of $\mathcal{C}$ is $\Gamma^{(\mathcal{A})} \times \Gamma^{(\mathcal{B})}$. Given $a \in \Sigma_c \cup \Sigma_r$ the transition matrix $M^{(\mathcal{C})}(a,(\gamma,\gamma'))$ is $M^{(\mathcal{A})}(a,\gamma) \otimes M^{(\mathcal{B})}(a,\gamma')$. Likewise, given $a \in \Sigma_{int}$ the transition matrix $M^{(\mathcal{C})}(a)$ is $M^{(\mathcal{A})}(a) \otimes M^{(\mathcal{C})}(a)$.

It is now straightforward to show that $M^{(\mathcal{C})}(w) = M^{(\mathcal{A})}(w) \otimes M^{(\mathcal{B})}(w)$ for all balanced words $w \in \Sigma^*$. The proof proceeds by induction on balanced words, following (6.1) and (6.2), and using Proposition 2.1 on Kronecker products. □

Proposition 6.5. *The equivalence problem for $\mathbb{Q}$-weighted visibly pushdown automata is logspace reducible to **ACIT**.*

Proof. Let $\mathcal{A}$ and $\mathcal{B}$ be $\mathbb{Q}$-weighted visibly pushdown automata with a total of n states between them. Then

$$\begin{aligned}\sum_{w \in L_n} (\mathcal{A}(w) - \mathcal{B}(w))^2 &= \sum_{w \in L_n} \mathcal{A}(w)^2 + \mathcal{B}(w)^2 - 2\mathcal{A}(w)\mathcal{B}(w) \\ &= \sum_{w \in L_n} (\mathcal{A} \otimes \mathcal{A})(w) + (\mathcal{B} \otimes \mathcal{B})(w) - 2(\mathcal{A} \otimes \mathcal{B})(w)\end{aligned}$$

Thus $\mathcal{A}$ is equivalent to $\mathcal{B}$ iff $\sum_{w \in L_n} (\mathcal{A} \otimes \mathcal{A})(w) + (\mathcal{B} \otimes \mathcal{B})(w) = 2 \sum_{w \in L_n} (\mathcal{A} \otimes \mathcal{B})(w)$. But Propositions 6.3 and 6.4 allow us to translate the above equation into an instance of **ACIT**. □

The trick of considering sums-of-squares of acceptance weights in the above proof is inspired by [29, Lemma 1].

7. Conclusion

It is known that deciding equivalence of $\mathbb{Q}$-weighted finite automata is in **NC** [29]. We have shown that deciding minimality is also in **NC**. Regarding the corresponding function problems, we have given an **RNC** algorithm to decide equivalence and output a counterexample word in case the input automata differ, and an **RNC** algorithm to minimise an automaton. We do not know whether either of these problems is in **NC**. It would be interesting to explore whether there is a relationship between these two problems, and to relate them to other problems in **RNC** that are not known to be in **NC**, such as bipartite matching.

For $\mathbb{Q}$-weighted VPA the situation is more complete. We have shown that deciding equivalence is equivalent to polynomial identity testing, the complexity of which is an important open problem.

References

[1] E.E. Allender, P. Bürgisser, J. Kjeldgaard-Pedersen, and P. Bro Miltersen. On the complexity of numerical analysis. *SIAM J. Comput.*, 38(5):1987–2006, 2009.

[2] S. Almagor, U. Boker, and O. Kupferman. What's decidable about weighted automata? In *ATVA*, volume 6996 of *LNCS*, pages 482–491. Springer, 2011.

[3] R. Alur and P. Madhusudan. Visibly pushdown languages. In *Proc. 36th Annual ACM Symposium on Theory of Computing STOC*, pages 202–211. ACM, 2004.

[4] R. Bellman. A Markovian Decision Process. *Journal of Mathematics and Mechanics*, 6, 1957.

[5] M. Blum, A. Chandra, and M. Wegman. Equivalence of free boolean graphs can be decided probabilistically in polynomial time. *Inf. Process. Lett.*, 10(2):80–82, 1980.

[6] D. Brumley and D. Boneh. Remote timing attacks are practical. *Computer Networks*, 48(5):701–716, 2005.

[7] J. W. Carlyle and A. Paz. Realizations by stochastic finite automata. *J. Comput. Syst. Sci.*, 5(1):26–40, 1971.

[8] K. Chatterjee, L. Doyen, and T. A. Henzinger. Probabilistic weighted automata. In *CONCUR*, volume 5710 of *LNCS*, pages 244–258. Springer, 2009.

[9] A. Condon and R. Lipton. On the complexity of space bounded interactive proofs (extended abstract). In *Proceedings of FOCS*, pages 462–467, 1989.

[10] S. A. Cook. A taxonomy of problems with fast parallel algorithms. *Information and Control*, 64(1-3):2–22, 1985.

[11] R. DeMillo and R. Lipton. A probabilistic remark on algebraic program testing. *Inf. Process. Lett.*, 7(4):193–195, 1978.

[12] K. Etessami and M. Yannakakis. Recursive Markov chains, stochastic grammars, and monotone systems of nonlinear equations. *J. ACM*, 56(1):1:1–1:66, 2009.

[13] R. Greenlaw, H.J. Hoover, and W.L. Ruzzo. *Limits to parallel computation: P-completeness theory.* Oxford University Press, 1995.

[14] O. H. Ibarra, S. Moran, and L. E. Rosier. A note on the parallel complexity of computing the rank of order n matrices. *Inf. Process. Lett.*, 11(4/5):162, 1980.

[15] S. Kiefer, A.S. Murawski, J. Ouaknine, B. Wachter, and J. Worrell. Language equivalence for probabilistic automata. In *CAV*, volume 6806 of *LNCS*, pages 526–540, 2011.

[16] Stefan Kiefer, Andrzej S. Murawski, Joël Ouaknine, Björn Wachter, and James Worrell. Apex: An analyzer for open probabilistic programs. In Madhusudan Parathasarathy and Sanjit A. Seshia, editors, *Proceedings of the 24th International Conference on Computer Aided Verification (CAV)*, volume 7358 of *LNCS*, pages 693–698, Berkeley, California, USA, 2012. Springer.

[17] P.C. Kocher. Timing attacks on implementations of Diffie-Hellman, RSA, DSS, and other systems. In *CRYPTO*, volume 1109 of *LNCS*, pages 104–113. Springer, 1996.
[18] D. Krob. The equality problem for rational series with multiplicities in the tropical semiring is undecidable. *Int. Journal of Alg. and Comp.*, 4(3):232–249, 1994.
[19] A. Kučera, J. Esparza, and R. Mayr. Model checking probabilistic pushdown automata. *Logical Methods in Computer Science*, 2(1):1–31, 2006.
[20] K. Mulmuley, U. V. Vazirani, and V. V. Vazirani. Matching is as easy as matrix inversion. In *STOC*, pages 345–354, 1987.
[21] M. O. Rabin. Probabilistic automata. *Inf. and Control*, 6 (3):230–245, 1963.
[22] R. L. Rivest, A. Shamir, and L. Adleman. A method for obtaining digital signatures and public-key cryptosystems. *Communications of the ACM*, 21:120–126, 1978.
[23] M.-P. Schützenberger. On the definition of a family of automata. *Inf. and Control*, 4:245–270, 1961.
[24] J. Schwartz. Fast probabilistic algorithms for verification of polynomial identities. *J. ACM*, 27(4):701–717, 1980.
[25] H. Seidl. Deciding equivalence of finite tree automata. *SIAM J. Comput.*, 19(3):424–437, 1990.
[26] G. Sénizergues. The equivalence problem for deterministic pushdown automata is decidable. In *ICALP*, volume 1256 of *LNCS*. Springer, 1997.
[27] C. Stirling. Deciding DPDA equivalence is primitive recursive. In *ICALP*, volume 2380 of *Lecture Notes in Computer Science*, pages 821–832. Springer, 2002.
[28] W. Tzeng. A polynomial-time algorithm for the equivalence of probabilistic automata. *SIAM Journal on Computing*, 21(2):216–227, 1992.
[29] W. Tzeng. On path equivalence of nondeterministic finite automata. *Inf. Process. Lett.*, 58(1):43–46, 1996.
[30] R. Zippel. Probabilistic algorithms for sparse polynomials. In *EUROSAM*, volume 72 of *Lecture Notes in Computer Science*, pages 216–226. Springer, 1979.