#NFA Admits an FPRAS: Efficient Enumeration, Counting, and Uniform Generation for Logspace Classes

MARCELO ARENAS and LUIS ALBERTO CROQUEVIELLE, Pontificia Universidad Católica & IMFD, Chile
RAJESH JAYARAM, Carnegie Mellon University, United States
CRISTIAN RIVEROS, Pontificia Universidad Católica & IMFD, Chile

In this work, we study two simple yet general complexity classes, based on logspace Turing machines, that provide a unifying framework for efficient query evaluation in areas such as information extraction and graph databases, among others. We investigate the complexity of three fundamental algorithmic problems for these classes: enumeration, counting, and uniform generation of solutions, and show that they have several desirable properties in this respect.

Both complexity classes are defined in terms of non-deterministic logspace transducers (NL-transducers). For the first class, we consider the case of unambiguous NL-transducers, and we prove constant delay enumeration and both counting and uniform generation of solutions in polynomial time. For the second class, we consider unrestricted NL-transducers, and we obtain polynomial delay enumeration, approximate counting in polynomial time, and polynomial-time randomized algorithms for uniform generation. More specifically, we show that each problem in this second class admits a fully polynomial-time randomized approximation scheme (FPRAS) and a polynomial-time Las Vegas algorithm (with preprocessing) for uniform generation. Remarkably, the key idea to prove these results is to show that the fundamental problem #NFA admits an FPRAS, where #NFA is the problem of counting the number of strings of length n (given in unary) accepted by a non-deterministic finite automaton (NFA). While this problem is known to be #P-complete and, more precisely, SpanL-complete, it was open whether this problem admits an FPRAS. In this work, we solve this open problem and obtain as a welcome corollary that every function in SpanL admits an FPRAS.

CCS Concepts: • **Information systems** → **Query optimization**; *Information extraction*; Semi-structured data; • **Theory of computation** → **Turing machines**; **Complexity classes**; **Regular languages**; *Probabilistic computation;*

Additional Key Words and Phrases: Enumeration, counting, uniform generation

ACM Reference format:
Marcelo Arenas, Luis Alberto Croquevielle, Rajesh Jayaram, and Cristian Riveros. 2021. #NFA Admits an FPRAS: Efficient Enumeration, Counting, and Uniform Generation for Logspace Classes. *J. ACM* 68, 6, Article 48 (October 2021), 40 pages.
https://doi.org/10.1145/3477045

This work was funded by ANID - Millennium Science Initiative Program - Code ICN17_002, Fondecyt Grant 1191337, and ANID BECAS/MAGISTER NACIONAL 2017-22171763.

Authors' addresses: M. Arenas, L. A. Croquevielle, and C. Riveros, Department of Computer Science, Vicuna Mackenna 4860, Edificio San Agustin, 4to piso, Macul 7820436, Santiago, Chile; emails: marenas@ing.puc.cl, {lacroquevielle, cristian.riveros}@uc.cl; R. Jayaram, Computer Science Department, Gates Hillman Complex, Carnegie Mellon University, 4902 Forbes Ave., Pittsburgh, PA, 15213, USA; email: rkjayara@cs.cmu.edu.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.
© 2021 Association for Computing Machinery.

0004-5411/2021/10-ART48 $15.00
https://doi.org/10.1145/3477045

1 INTRODUCTION

Arguably, query answering is the most fundamental problem in databases. In this respect, developing efficient query answering algorithms, as well as understanding when this cannot be done, is of paramount importance for database theory and applications. In the most classical view of this problem, one is interested in computing all the answers, or solutions, to a query. However, as the quantity of data becomes enormously large, the number of answers to a query could also be enormous, so computing the complete set of solutions can be prohibitively expensive. To overcome this limitation, the idea of enumerating the answers to a query with a *small delay* has been recently studied in the database area [31]. More specifically, the idea is to divide the computation of the answers to a query into two phases. In a *preprocessing* phase, some data structures are constructed to accelerate the process of computing answers. Then, in an *enumeration* phase, the answers are enumerated with a small delay between them. In particular, in the case of constant delay enumeration algorithms, the preprocessing phase should take polynomial time, while the time between consecutive answers should be constant.

Constant delay enumeration algorithms allow users to retrieve a fixed number of answers very efficiently, which can give them a lot of information about the solutions to a query. In fact, the same holds if users need a linear or a polynomial number of answers. However, because of the data structures used in the preprocessing phase, these algorithms usually return answers that are very similar to each other [10, 15, 31]; for example, tuples with n elements where only the first few coordinates are changed in the first answers that are returned. In this respect, other approaches can be used to return some solutions efficiently but improving their variety. Most notably, the possibility of generating an answer uniformly, at random, is a desirable condition if it can be done efficiently. Notice that returning varied solutions has been identified as an important property not only in databases, but also for algorithms that retrieve information in a broader sense [1].

Efficient algorithms for either enumerating or uniformly generating the answers to a query are powerful tools to help in the process of understanding the answers to a query. But how can we know how long these algorithms should run and how complete the set of computed answers is? A third tool that is needed, then, is an efficient algorithm for computing, or estimating, the number of solutions to a query. Then, taken together, enumeration, counting, and uniform generation techniques form a powerful attacking trident when confronting the problem of answering a query.

In this article, we follow a principled approach to study the problems of enumerating, counting, and uniformly generating the answers to a query. More specifically, we begin by following the guidance of Reference [21], which urges the use of relations to formalize the notion of solution to a given input of a problem (for instance, to formalize the notion of answer to an input query over an input database). While there are many ways of formalizing this notion, most such formalizations only make sense for a specific kind of queries, e.g., a subset of the integers is well-suited as the solution set for counting problems, but not for sampling problems. We want a general framework, so by following Reference [21], we represent a problem as a relation $R \subseteq \{0,1\}^* \times \{0,1\}^*$, and we say that y is a solution for an input x if $(x,y) \in R$.[1] Note that the problem of enumerating the solutions to a given input x corresponds to the problem of enumerating the elements of the set $\{y \in \{0,1\}^* \mid (x,y) \in R\}$, while the counting and uniform generation problems correspond to the problems of computing the cardinality of $\{y \in \{0,1\}^* \mid (x,y) \in R\}$ and uniformly generating, at random, a string in this set, respectively.

Second, we study two simple yet general complexity classes for relations, based on **non-deterministic logspace transducers (NL-transducers)**, which provide a unifying framework

[1]For the sake of presentation, we assume relations to be defined over the binary alphabet $\{0, 1\}$. The results of this article also hold if we consider relations defined over an arbitrary finite alphabet Σ.

for studying enumeration, counting, and uniform generation. More specifically, an NL-transducer M is a non-deterministic Turing Machine with input and output alphabet $\{0,1\}$, a read-only input tape, a write-only output tape, and a work-tape of which, on input $x \in \{0,1\}^*$, only the first $O(\log(|x|))$ cells can be used. Moreover, a string $y \in \{0,1\}^*$ is said to be an output of M on input x if there exists a run of M on input x that halts in an accepting state with y as the string in the output tape. Finally, assuming that all outputs of M on input x are denoted by $M(x)$, a relation $R \subseteq \{0,1\}^* \times \{0,1\}^*$ is said to be accepted by M if for every input x, it holds that $M(x) = \{y \in \{0,1\}^* \mid (x,y) \in R\}$.

The first complexity class of relations studied in this article consists of the relations accepted by unambiguous NL-transducers. More precisely, an NL-transducer M is said to be unambiguous if for every input x and $y \in M(x)$, there exists exactly one run of M on input x that halts in an accepting state with y as the string in the output tape. For this class, we are able to achieve constant delay enumeration and both counting and uniform generation of solutions in polynomial time. For the second class, we consider (unrestricted) NL-transducers, and we obtain polynomial delay enumeration, approximate counting in polynomial time, and polynomial-time randomized algorithms for uniform generation. More specifically, we show that each problem in this second class admits a **fully polynomial-time randomized approximation scheme (FPRAS)** [21] and a polynomial-time Las Vegas algorithm (with preprocessing) for uniform generation. It is important to mention that the key idea to prove these results is to show that the fundamental problem #NFA admits an FPRAS, where #NFA is the problem of counting the number of strings of length n (given in unary) accepted by a **non-deterministic finite automaton (NFA)**. While this problem is known to be #P-complete and, more precisely, SpanL-complete [3], it was open whether it admits an FPRAS, and only **quasi-polynomial time randomized approximation schemes (QPRAS)** were known for it [18, 23]. In this work, we solve this open problem and obtain as a welcome corollary that every function in SpanL admits an FPRAS. Thus, to the best of our knowledge, we identify SpanL as the first complexity class with a simple and robust definition based on Turing Machines, which contains #P-complete problems and where each problem admits an FPRAS.

Proviso. This article is an extended version of the article *"Efficient Logspace Classes for Enumeration, Counting, and Uniform Generation,"* which was published in the 38th ACM SIGMOD-SIGACT-SIGAI Symposium on Principles of Database Systems (PODS 2019). For this extended version, we have made many changes. In particular, we have completely reworked the proof that #NFA admits a fully polynomial-time randomized approximation schema, which is the main result of this work. More specifically, we have carefully restructured this proof for the sake of readability, obtaining a simpler and more efficient algorithm. Besides, for the more general class of relations defined in terms of (unrestricted) NL-transducers, this result allows us to prove that each problem in this class admits a polynomial-time Las Vegas uniform generator. Such a randomized algorithm needs a preprocessing phase, which was not properly made explicit in the conference paper. We have solved this issue by introducing, and studying, the notion of preprocessing polynomial-time Las Vegas uniform generator.

Organization of the article. The main terminology used in the article is given in Section 2. In Section 3, we define the two classes studied in this article and state our main results. In Section 4, we show how these classes can be used to obtain positive results on query evaluation in information extraction, graph databases, and binary decision diagrams. The complete proofs of our results are presented in Sections 5 and 6 and Appendix A. In particular, we explain the algorithmic techniques used to obtain an FPRAS for the #NFA problem in Section 6, where we also provide a detailed proof of this result. Finally, some concluding remarks are given in Section 7.

2 PRELIMINARIES

Given natural numbers $n \leq m$, we use notation $[n, m]$ for the set $\{n, \ldots, m\}$. Besides, we use $\log(x)$ to refer to the logarithm of x to base e.

2.1 Relations and Problems

As usual, $\{0,1\}^*$ denotes the set of all strings over the binary alphabet $\{0,1\}$, $|x|$ denotes the length of a string $x \in \{0,1\}^*$, $x_1 \cdot x_2$ denotes the concatenation of two strings $x_1, x_2 \in \{0,1\}^*$, and $\{0,1\}^n$ denotes the set of all strings $x \in \{0,1\}^*$ such that $|x| = n$. A problem is represented as a relation $R \subseteq \{0,1\}^* \times \{0,1\}^*$. For every pair $(x,y) \in R$, we interpret x as being the encoding of an input to some problem, and y as being the encoding of a solution to that input. For each $x \in \{0,1\}^*$, we define the set $W_R(x) = \{y \in \{0,1\}^* \mid (x,y) \in R\}$ and call it the set of solutions for x. Also, if $y \in W_R(x)$, then we call y a solution to x.

This is a very general framework, so we work with p-relations [21]. Formally, a relation $R \subseteq \{0,1\}^* \times \{0,1\}^*$ is a p-relation if (1) there exists a polynomial q such that $(x,y) \in R$ implies that $|y| \leq q(|x|)$ and (2) there exists a deterministic Turing Machine that receives as input $(x,y) \in \{0,1\}^* \times \{0,1\}^*$, runs in polynomial time, and accepts if, and only if, $(x,y) \in R$. Without loss of generality, from now on we assume that for a p-relation R, there exists a polynomial q such that $|y| = q(|x|)$ for every $(x,y) \in R$. This is not a strong requirement, since all solutions can be made to have the same length through padding.

2.2 Enumeration, Counting, and Uniform Generation

There are several computational problems associated to a relation R. For example, given a relation R and an input $x \in \{0,1\}^*$, we could consider the existence problem of deciding whether there are any solutions for x. In this article, given a p-relation R, we are interested in the following problems:

Problem:	ENUM(R)
Input:	A word $x \in \{0,1\}^*$
Output:	Enumerate all $y \in W_R(x)$ without repetitions

Problem:	COUNT(R)
Input:	A word $x \in \{0,1\}^*$
Output:	The size $|W_R(x)|$

Problem:	GEN(R)
Input:	A word $x \in \{0,1\}^*$
Output:	Generate uniformly, at random, a word in $W_R(x)$

Given that $|y| = q(|x|)$ for every $(x,y) \in R$, we have that $W_R(x)$ is finite and these three problems are well defined. Notice that in the case of ENUM(R), we do not assume a specific order on words, so the elements of $W_R(x)$ can be enumerated in any order (but without repetitions). Moreover, in the case of COUNT(R), we assume that $|W_R(x)|$ is encoded in binary and, therefore, the size of the output is logarithmic in the size of $W_R(x)$. Finally, in the case of GEN(R), we generate a word $y \in W_R(x)$ with probability $\frac{1}{|W_R(x)|}$ if the set $W_R(x)$ is not empty; otherwise, we return a special symbol $\perp$ to indicate that $W_R(x) = \varnothing$.

2.3 Enumeration with Polynomial and Constant Delay

An enumeration algorithm for ENUM(R) is a procedure that receives an input $x \in \{0,1\}^*$ and, during the computation, it outputs each word in $W_R(x)$, one-by-one and without repetitions. The

time between two consecutive outputs is called the delay of the enumeration. In this article, we consider two restrictions on the delay: polynomial delay and constant delay. *Polynomial delay enumeration* is the standard notion of polynomial time efficiency in enumeration algorithms [22] and is defined as follows: An enumeration algorithm is of polynomial delay if there exists a polynomial p such that for every input $x \in \{0,1\}^*$, the time between the beginning of the algorithm and the initial output, between any two consecutive outputs, and between the last output and the end of the algorithm, is bounded by $p(|x|)$.

Constant delay enumeration is another notion of efficiency for enumeration algorithms that has attracted a lot attention in recent years [9, 12, 31]. This notion has stronger guarantees compared to polynomial delay: The enumeration is done in a second phase after the processing of the input and taking constant-time between two consecutive outputs in a very precise sense. Several notions of constant delay enumeration have been given, most of them in database theory where it is important to divide the analysis between query and data. In this article, we want a definition of constant delay that is agnostic of the distinction between query and data (i.e., combined complexity) and, for this reason, we use a more general notion of constant delay enumeration than the one in References [9, 12, 31].

Constant delay enumeration cannot be achieved in general with a standard Turing Machine, because merely moving the head through the tape will take up more than constant time. So, as it is standard in the literature [31], for the notion of constant delay enumeration, we consider enumeration algorithms on **Random Access Machines (RAM)** with addition and uniform cost measure [2]. Given a relation $R \subseteq \{0,1\}^* \times \{0,1\}^*$, an enumeration algorithm E for R has constant delay if E runs in two phases over the input x.

(1) The first phase (precomputation), which does not produce output.
(2) The second phase (enumeration), which occurs immediately after the precomputation phase, where all words in $W_R(x)$ are enumerated without repetitions and satisfying the following conditions, for a fixed constant c:
  (a) the time it takes to generate the first output y is bounded by $c \cdot |y|$;
  (b) the time between two consecutive outputs y and y' is bounded by $c \cdot |y'|$ and does not depend on y; and
  (c) the time between the final element y that is returned and the end of the enumeration phase is bounded by $c \cdot |y|$,

We say that E is a constant delay algorithm for R with precomputation phase f if E has constant delay and the precomputation phase takes time $O(f(|x|))$. Moreover, we say that $\text{ENUM}(R)$ can be solved with constant delay if there exists a constant delay algorithm for R with precomputation phase p for some polynomial p.

Our notion of constant delay algorithm differ from the definitions in Reference [31] in two aspects. First, in our definition the input is not divided into some components, so the preprocessing phase must take polynomial time in the size of the entire input. In the case of constant delay algorithms for query answering, the input is usually divided into the data and the query, and the preprocessing phase is only asked to take polynomial time in the size of the data (as the query is usually assumed to be fixed, which is referred to as data complexity [33]). Second, our definition of constant delay is what in References [9, 12] is called *linear delay in the size of the output*, namely, writing the next output is linear in its size and does not depend on the size of the input. This is a natural assumption, since each output must at least be written down to return it to the user. Notice that, given an input x and an output y, the notion of polynomial delay above means polynomial in $|x|$ and, instead, the notion of linear delay from References [9, 12] means linear in $|y|$, i.e., constant in the size of $|x|$. Thus, we have decided to call the two-phase enumeration from above "constant

delay," as it does not depend on the size of the input x, and the delay is just what is needed to write the output (which is the minimum requirement for such an enumeration algorithm).

2.4 Approximate Counting and Las Vegas Uniform Generation with Preprocessing

Given a relation $R \subseteq \{0,1\}^* \times \{0,1\}^*$, the problem $\text{COUNT}(R)$ can be solved efficiently if there exists a polynomial-time algorithm that, given $x \in \{0,1\}^*$, computes $|W_R(x)|$. In other words, if we think of $\text{COUNT}(R)$ as a function that maps x to the value $|W_R(x)|$, then $\text{COUNT}(R)$ can be computed efficiently if $\text{COUNT}(R) \in \text{FP}$, the class of functions that can be computed in polynomial time. As such a condition does not hold for many fundamental problems, we also consider the possibility of efficiently approximating the value of the function $\text{COUNT}(R)$. More precisely, $\text{COUNT}(R)$ is said to admit a FPRAS [21] if there exists a randomized algorithm $\mathcal{A} : \{0,1\}^* \times (0,1) \to \mathbb{N}$ and a polynomial $q(u,v)$ such that for every $x \in \{0,1\}^*$ and $\delta \in (0,1)$, it holds that:

$$\mathbf{Pr}(|\mathcal{A}(x,\delta) - |W_R(x)|| \leq \delta \cdot |W_R(x)|) \quad \geq \quad \frac{3}{4},$$

and the time needed to compute $\mathcal{A}(x,\delta)$ is at most $q(|x|, \frac{1}{\delta})$. Thus, $\mathcal{A}(x,\delta)$ approximates the value $|W_R(x)|$ with a relative error of δ, and it can be computed in polynomial time in the size of x and the value $\frac{1}{\delta}$.

The problem $\text{GEN}(R)$ can be solved efficiently if there exists a polynomial-time randomized algorithm that, given $x \in \{0,1\}^*$, generates an element of $W_R(x)$ with uniform probability distribution (if $W_R(x) = \emptyset$, then it returns $\perp$). However, as in the case of $\text{COUNT}(R)$, the existence of such a generator is not guaranteed for many fundamental problems, so we also consider a relaxed notion of generation that has a probability of failing in returning a solution. More precisely, $\text{GEN}(R)$ is said to admit a **preprocessing polynomial-time Las Vegas uniform generator (PPLVUG)** if there exists a pair of randomized algorithms $\mathcal{P} : \{0,1\}^* \times (0,1) \to (\{0,1\}^* \cup \{\perp\})$, $\mathcal{G} : \{0,1\}^* \to (\{0,1\}^* \cup \{\textbf{fail}\})$ and a pair of polynomials $q(u,v)$, $r(u)$ such that for every $x \in \{0,1\}^*$ and $\delta \in (0,1)$:

(1) The preprocessing algorithm $\mathcal{P}$ receives as inputs x and δ and runs in time bounded by $q(|x|, \log(1/\delta))$. If $W_R(x) \neq \emptyset$, then $\mathcal{P}(x,\delta)$ returns a string $\mathcal{D}$ such that $\mathcal{D}$ is *good-for-generation* with probability $1-\delta$. If $W_R(x) = \emptyset$, then $\mathcal{P}(x,\delta)$ returns $\perp$.
(2) The generator algorithm $\mathcal{G}$ receives as input $\mathcal{D}$ and runs in time bounded by $r(|\mathcal{D}|)$. Moreover, if $\mathcal{D}$ is good-for-generation, then:
   (a) $\mathcal{G}(\mathcal{D})$ returns **fail** with a probability at most $\frac{1}{2}$, and
   (b) conditioned on not returning **fail**, $\mathcal{G}(\mathcal{D})$ returns a truly uniform sample $y \in W_R(x)$, i.e., with a probability $1/|W_R(x)|$ for each $y \in W_R(x)$.
   Otherwise, if $\mathcal{D}$ is not good-for-generation, then $\mathcal{G}(\mathcal{D})$ outputs a string without any guarantee.

In line with the notion of constant delay enumeration algorithm, we allow the previous concept of uniform generator to have a preprocessing phase. If there is no solution for the input x (that is, $W_R(x) = \emptyset$), then the preprocessing algorithm $\mathcal{P}$ returns the symbol $\perp$. Otherwise, the invocation $\mathcal{P}(x,\delta)$ returns a string $\mathcal{D}$ in $\{0,1\}^*$, namely, a data structure or "advice" for the generation procedure $\mathcal{G}$. The output of the invocation $\mathcal{P}(x,\delta)$ is used by the generator algorithm $\mathcal{G}$ to produce a solution of x with uniform distribution (that is, with probability $1/|W_R(x)|$). If the output of $\mathcal{P}(x,\delta)$ is not good-for-generation (which occurs with probability δ), then we have no guarantees on the output of the generator algorithm $\mathcal{G}$. Otherwise, we know that $\mathcal{G}(\mathcal{D})$ returns an element of $W_R(x)$ with uniform distribution, or it returns **fail**. Furthermore, we can repeat $\mathcal{G}(\mathcal{D})$ as many times as needed, generating each time a truly uniform sample y from $W_R(x)$ whenever $y \neq \textbf{fail}$.

Notice that by condition (2a), we know that this probability of failing is smaller than $\frac{1}{2}$, so by invoking $\mathcal{G}(\mathcal{D})$ several times, we can make this probability arbitrarily small (for example, the probability that $\mathcal{G}(\mathcal{D})$ returns **fail** in 1,000 consecutive independent invocations is at most $(\frac{1}{2})^{1000}$). Moreover, we have that $\mathcal{P}(x, \delta)$ can be computed in time $q(|x|, \log(1/\delta))$, so we can consider an exponentially small value of δ such as

$$\frac{1}{2^{|x|+1000}},$$

and still obtain that $\mathcal{P}(x, \delta)$ can be computed in time polynomial in $|x|$. Notice that with such a value of δ, the probability of producing a good-for-generation string $\mathcal{D}$ is at least

$$1 - \frac{1}{2^{1000}},$$

which is an extremely high probability. Finally, it is important to notice that the size of $\mathcal{D}$ is at most $q(|x|, \log(1/\delta))$, so $\mathcal{G}(\mathcal{D})$ can be computed in time polynomial in $|x|$ and $\log(1/\delta)$. Therefore, $\mathcal{G}(\mathcal{D})$ can be computed in time polynomial in $|x|$ even if we consider an exponentially small value for δ such as $1/2^{|x|+1000}$.

It is important to notice that the notion of preprocessing polynomial-time Las Vegas uniform generator imposes stronger requirements than the notion of fully polynomial-time almost uniform generator introduced in Reference [21]. In particular, the latter not only has a probability of failing, but also considers the possibility of generating a solution with a probability distribution that is *almost* uniform, that is, an algorithm that generates an string $y \in W_R(x)$ with a probability in an interval $[1/|W_R(x)| - \varepsilon, 1/|W_R(x)| + \varepsilon]$ for a given error $\varepsilon \in (0, 1)$.

3 NLOGSPACE TRANSDUCERS: DEFINITIONS AND OUR MAIN RESULTS

The goal of this section is to provide simple yet general definitions of classes of relations with good properties in terms of enumeration, counting, and uniform generation. More precisely, we are first aiming at providing a class $\mathcal{C}$ of relations that has a simple definition in terms of Turing Machines and such that for every relation $R \in \mathcal{C}$, it holds that ENUM(R) can be solved with constant delay, and both COUNT(R) and GEN(R) can be solved in polynomial time. Moreover, as it is well known that such good conditions cannot always be achieved, we are then aiming at extending the definition of $\mathcal{C}$ to obtain a simple class, also defined in terms of Turing Machines and with good approximation properties. It is important to mention that we are not looking for an exact characterization in terms of Turing Machines of the class of relations that admit constant delay enumeration algorithms, as this may result in an overly complicated model. Instead, we are looking for simple yet general classes of relations with good properties in terms of enumeration, counting, and uniform generation, and which can serve as a starting point for the systematic study of these three fundamental properties.

A key notion that is used in our definitions of classes of relations is that of transducer. An NL-transducer M is a non-deterministic Turing Machine with input and output alphabet $\{0, 1\}$, a read-only input tape, a write-only output tape where the head is always moved to the right once a symbol is written in it (so the output cannot be read by M), and a work-tape of which, on input x, only the first $f(|x|)$ cells can be used, where $f(n) \in O(\log(n))$. A string $y \in \{0, 1\}^*$ is said to be an output of M on input x if there exists a run of M on input x that halts in an accepting state with y as the string in the output tape. The set of all outputs of M on input x is denoted by $M(x)$ (notice that $M(x)$ can be empty). Finally, the relation accepted by M, denoted by $\mathcal{R}(M)$, is defined as $\{(x, y) \in \{0, 1\}^* \times \{0, 1\}^* \mid y \in M(x)\}$.

Definition 3.1. A relation R is in RelationNL if, and only if, there exists an NL-transducer M such that $\mathcal{R}(M) = R$.

Cycles are forbidden in NL-transducers to ensure polynomial-size solutions for each input [3]. However, we do not need to impose this restriction here, as we only work with p-relations in this article (see Section 2).

The class RelationNL should be general enough to contain some natural and well-studied problems. A first such a problem is the satisfiability of a propositional formula in DNF. As a relation, this problem can be represented as follows:

$$\text{SAT-DNF} = \{(\varphi, \sigma) \mid \varphi \text{ is a propositional formula in DNF}, \sigma \text{ is a truth assignment, and } \sigma(\varphi) = 1\}.$$

Thus, we have that ENUM(SAT-DNF) corresponds to the problem of enumerating the truth assignments satisfying a propositional formula φ in DNF, while COUNT(SAT-DNF) and GEN(SAT-DNF) correspond to the problems of counting and uniformly generating such truth assignments, respectively. It is not difficult to see that SAT-DNF $\in$ RelationNL. In fact, assume that we are given a propositional formula φ of the form $D_1 \vee \cdots \vee D_m$, where each D_i is a conjunction of literals, that is, a conjunction of propositional variables and negation of propositional variables. Moreover, assume that each propositional variable in φ is of the form x_k, where k is a binary number, and that $x_1, \ldots, x_n$ are the variables occurring in φ. Notice that with such a representation, we have that φ is a string over the alphabet $\{x, _, 0, 1, \wedge, \vee, \neg\}$.[2] We define as follows an NL-transducer M such that $M(\varphi)$ is the set of truth assignments satisfying φ. On input φ, the NL-transducer M non-deterministically chooses a disjunct D_i, which is represented by two indexes indicating the starting and ending symbols of D_i in the string φ. Then it checks whether D_i is satisfiable, that is, whether D_i does not contain complementary literals. Notice that this can be done in logarithmic space by checking for every $j \in \{1, \ldots, n\}$, whether x_j and $\neg x_j$ are both literals in D_i. If D_i is not satisfiable, then M halts in a non-accepting state. Otherwise, M returns a satisfying truth assignment of D_i as follows: A truth assignment for φ is represented by a string of length n over the alphabet $\{0, 1\}$, where the jth symbol of this string is the truth value assigned to variable x_j. Then for every $j \in \{1, \ldots, n\}$, if x_j is a conjunct in D_i, then M writes the symbol 1 in the output tape, and if $\neg x_j$ is a conjunct in D_i, then M writes the symbol 0 in the output tape. Finally, if neither x_j nor $\neg x_j$ is a conjunct in D_i, then M non-deterministically chooses a symbol $b \in \{0, 1\}$, and it writes b in the output tape.

Given that COUNT(SAT-DNF) is a #P-complete problem [27], we cannot expect COUNT(R) to be solvable in polynomial time for every $R \in$ RelationNL. However, COUNT(SAT-DNF) admits an FPRAS [24], so we can still hope for COUNT(R) to admit an FPRAS for every $R \in$ RelationNL. It turns out that proving such a result involves providing an FPRAS for another natural and fundamental problem: #NFA. More specifically, #NFA is the problem of counting the number of words of length k accepted by a non-deterministic finite automaton without epsilon transitions (NFA), where k is given in unary (that is, k is given as a string 0^k). It is known that #NFA is #P-complete [3], but it is open whether it admits an FPRAS; in fact, the best randomized approximation scheme known for #NFA runs in time $n^{O(\log(n))}$ [23]. In our notation, this problem is represented by the following relation:

$$\text{MEM-NFA} = \{((A, 0^k), w) \mid A \text{ is an NFA with alphabet } \{0,1\}, w \in \{0,1\}^k \text{ and } w \text{ is accepted by } A\},$$

that is, we have that #NFA = COUNT(MEM-NFA). It is easy to see that MEM-NFA $\in$ RelationNL. Hence, we give a positive answer to the open question of whether #NFA admits an FPRAS by proving the following general result about RelationNL.

[2] For the sake of presentation, we consider a non-binary alphabet in this case, although it is easy to see how SAT-DNF can be represented by using the binary alphabet.

Theorem 3.2. *If* $R \in$ RelationNL, *then* $\text{ENUM}(R)$ *can be solved with polynomial delay,* $\text{COUNT}(R)$ *admits an FPRAS, and* $\text{GEN}(R)$ *admits a PPLVUG.*

It is worth mentioning a fundamental consequence of this result in computational complexity. The class of functions SpanL was introduced in Reference [3] to provide a characterization of some functions that are hard to compute. More specifically, a function $f : \{0,1\}^* \to \mathbb{N}$ is in SpanL if there exists an NL-transducer M with input alphabet $\{0,1\}$ such that $f(x) = |M(x)|$ for every $x \in \{0,1\}^*$. The complexity class SpanL is contained in #P, and it is a hard class in the sense that if SpanL $\subseteq$ FP, then P = NP [3], where FP is the class of functions that can be computed in polynomial time. In fact, SpanL has been instrumental in proving that some functions are difficult to compute [3, 7, 19, 25]. It is not difficult to see that #NFA belongs to SpanL [3].

Given functions $f, g : \{0,1\}^* \to \mathbb{N}$, f is said to be parsimoniously reducible to g in polynomial-time if there exists a polynomial-time computable function $h : \{0,1\}^* \to \{0,1\}^*$ such that, for every $x \in \{0,1\}^*$, it holds that $f(x) = g(h(x))$. It is known that #NFA is SpanL-complete under polynomial-time parsimonious reductions, which in particular implies that if #NFA can be computed in polynomial time, then P = NP [3]. Moreover, given that #NFA admits an FPRAS and parsimonious reductions preserve the existence of FPRAS, we obtain the following corollary from Theorem 3.2:

Corollary 3.3. *Every function in* SpanL *admits an FPRAS.*

Although some classes $\mathcal{C}$ containing #P-complete functions and for which every $f \in \mathcal{C}$ admits an FPRAS have been identified before [8, 29], to the best of our knowledge this is the first such a class with a simple and robust definition based on Turing Machines.

A tight relationship between the existence of an FPRAS and the existence of a schema for almost uniform generation was proved in Reference [21] for the class of relations that are *self-reducible*. Thus, one might wonder whether the existence of a PPLVUG for $\text{GEN}(R)$ in Theorem 3.2 is just a corollary of our FPRAS for $\text{COUNT}(R)$ along with the result in Reference [21]. However, as the notion of PPLVUG asks for a uniform generator without any distributional error ε, it is not clear how to infer its existence from the results in Reference [21]. Thus, we prove in Section 6 that $\text{COUNT}(R)$ admits an FPRAS and $\text{GEN}(R)$ admits a PPLVUG, for a relation $R \in$ RelationNL, without utilizing the aforementioned result from Reference [21].

A natural question at this point is whether a simple syntactic restriction on the definition of RelationNL gives rise to a class of relations with better properties in terms of enumeration, counting, and uniform generation. Fortunately, the answer to this question comes by imposing a natural and well-studied restriction on Turing Machines, which allows the definition of a class that contains many natural problems. More precisely, we consider the notion of UL-transducer, where the letter "U" stands for "unambiguous." Formally, M is a UL-transducer if M is an NL-transducer such that for every input x and $y \in M(x)$, there exists exactly one run of M on input x that halts in an accepting state with y as the string in the output tape. Notice that this notion of transducer is based on well-known classes of decision problems (e.g., UP [32] and UL [28]) adapted to our case, namely, adapted to problems defined as relations.

Definition 3.4. A relation R is in RelationUL if, and only if, there exists a UL-transducer M such that $\mathcal{R}(M) = R$.

For the class RelationUL, we obtain the following result:

Theorem 3.5. *If* $R \in$ RelationUL, *then* $\text{ENUM}(R)$ *can be solved with constant delay, there exists a polynomial-time algorithm for* $\text{COUNT}(R)$, *and there exists a polynomial-time randomized algorithm for* $\text{GEN}(R)$.

In particular, it should be noticed that given $R \in$ RelationUL and an input x, the solutions for x can be enumerated, counted, and uniformly generated efficiently. In the following section, we provide examples of relations in this class.

Classes of problems definable by machine models and that can be enumerated with constant delay have been proposed before. In Reference [4], it is shown that if a problem is definable by a d-DNNF circuit, then the solutions of an instance can be listed with linear preprocessing and constant delay enumeration. Still, to the best of our knowledge, RelationUL is the first such a class with a simple and robust definition based on Turing Machines.

On the relationship of RelationNL *and* RelationUL *with known complexity classes.* It is well-known that a function f is in #P if and only if there exists a p-relation R such that $f = \text{COUNT}(R)$ (recall the definition of p-relation from Section 2). In the same way, there exists a tight relationship between SpanL and RelationNL, as it is easy to see that a function f is in SpanL if and only if there exists a relation $R \in$ RelationNL such that $f = \text{COUNT}(R)$. Hence, the reader may wonder why it is necessary to introduce RelationNL and RelationUL, considering further that such classes are defined in terms of well-known Turing Machine models. The key issue to consider here is that function complexity classes, such as #P and SpanL, are not appropriate to state results about the enumeration and uniform generation problems. For instance, it would not be correct to state that every function in SpanL admits a PPLVUG, as the definition of SpanL does not provide a unique notion of solution for an input, which is the object to be generated in this case. In this respect, we introduce RelationNL and RelationUL to have a unified framework to study the counting, enumeration, and uniform generation problems. The definition of such classes should not be considered as a contribution of this article. In fact, they should only be seen as our way of following the guidance of Reference [21], which, as mentioned before, urges the use of relations to formalize the notion of solution for an input of a problem.

4 APPLICATIONS OF THE MAIN RESULTS

Before providing the proofs of Theorems 3.2 and 3.5, we give some implications of these results. In particular, we show how NL- and UL-transducers can be used to obtain positive results on query evaluation in areas such as information extraction, graph databases, and binary decision diagrams.

4.1 Information Extraction

In Reference [14], the framework of document spanners was proposed as a formalization of ruled-based information extraction. In this framework, the main data objects are documents and spans. Formally, given a finite alphabet Σ, a document is a string $d = a_1 \ldots a_n$, and a span is pair $s = [i, j\rangle$ with $1 \leq i \leq j \leq n + 1$. A span represents a continuous region of the document d, whose content is the substring of d from positions i to $j - 1$. Given a finite set of variables $\mathbf{X}$, a mapping μ is a function from $\mathbf{X}$ to the spans of d.

Variable set automata (VA) are one of the main formalisms to specify sets of mappings over a document. Here, we use the notion of extended VA (eVA) from Reference [15] to state our main results. We only recall the main definitions, and we refer the reader to References [14, 15] for more intuition and further details. An eVA is a tuple $\mathcal{A} = (Q, q_0, F, \delta)$ such that Q is a finite set of states, q_0 is the initial state, and F is the final set of states. Further, δ is the transition relation consisting of letter transitions (q, a, q'), or variable-set transitions (q, S, q'), where $S \subseteq \{x\vdash, \dashv x \mid x \in \mathbf{X}\}$ and $S \neq \emptyset$. The symbols $x\vdash$ and $\dashv x$ are called markers, and they are used to denote that variable x is opened or closed by $\mathcal{A}$, respectively. A run ρ over a document $d = a_1 \cdots a_n$ is a sequence of the form: $q_0 \xrightarrow{X_1} p_0 \xrightarrow{a_1} q_1 \xrightarrow{X_2} p_1 \xrightarrow{a_2} \cdots \xrightarrow{a_n} q_n \xrightarrow{X_{n+1}} p_n$ where each X_i is a (possible empty) set of markers, $(p_i, a_{i+1}, q_{i+1}) \in \delta$, and $(q_i, X_{i+1}, p_i) \in \delta$ whenever $X_{i+1} \neq \emptyset$, and $q_i = p_i$ otherwise (that is, when

$X_{i+1} = \emptyset$). We say that a run ρ is valid if for every $x \in \mathbf{X}$ there exists exactly one pair $[i, j\rangle$ such that $x\vdash \in X_i$ and $\dashv x \in X_j$. A valid run ρ naturally defines a mapping μ^ρ that maps x to the only span $[i, j\rangle$ such that $x\vdash \in X_i$ and $\dashv x \in X_j$. We say that ρ is accepting if $p_n \in F$. Finally, the semantics $[\![\mathcal{A}]\!](d)$ of $\mathcal{A}$ over d is defined as the set of all mappings μ^ρ where ρ is a valid and accepting run of $\mathcal{A}$ over d.

In References [16, 26], it was shown that the decision problem related to query evaluation, namely, given an eVA $\mathcal{A}$ and a document d deciding whether $[\![\mathcal{A}]\!](d) \neq \emptyset$, is NP-hard. For this reason, in Reference [15] a subclass of eVA is considered to recover polynomial-time evaluation. An eVA $\mathcal{A}$ is called functional if every accepting run is valid. Intuitively, a functional eVA does not need to check validity of the run given that it is already known that every run that reaches a final state will be valid.

For the query evaluation problem of functional eVA (i.e., to compute $[\![\mathcal{A}]\!](d)$), one can naturally associate the following relation:

$$\text{EVAL-eVA} = \{((\mathcal{A}, d), \mu) \mid \mathcal{A} \text{ is a functional eVA, } d \text{ is a document, and } \mu \in [\![\mathcal{A}]\!](d)\}.$$

It is not difficult to show that EVAL-eVA is in RelationNL. Hence, by Theorem 3.2, we get the following results:

Corollary 4.1. ENUM(EVAL-eVA) *can be enumerated with polynomial delay,* COUNT (EVAL-eVA) *admits an FPRAS, and* GEN(EVAL-eVA) *admits a PPLVUG.*

In Reference [15], it was shown that every functional RGX or functional VA (not necessarily extended) can be converted in polynomial time into an functional eVA. Therefore, Corollary 4.1 also holds for these more general classes. Notice that in Reference [17], a polynomial delay enumeration algorithm for $[\![\mathcal{A}]\!](d)$ was provided. Thus, only the results about COUNT(EVAL-eVA) and GEN(EVAL-eVA) can be considered as new.

Regarding efficient enumeration and exact counting, a constant delay algorithm with polynomial preprocessing was given in Reference [15] for the class of deterministic functional eVA. Here, we can easily extend these results for a more general class, which we called unambiguous functional eVA. Formally, we say that an eVA is unambiguous if for every two valid and accepting runs ρ_1 and ρ_2, it holds that $\mu^{\rho_1} \neq \mu^{\rho_2}$. In other words, each output of an unambiguous eVA is witnessed by exactly one run. As in the case of EVAL-eVA, we can define the relation EVAL-UeVA by restricting the input to unambiguous functional eVA. By using UL-transducers and Theorem 3.5, we can then extend the results in Reference [15] for the unambiguous case.

Corollary 4.2. ENUM(EVAL-UeVA) *can be solved with constant delay, there exists a polynomial-time algorithm for* COUNT(EVAL-UeVA)*, and there exists a polynomial-time randomized algorithm for* GEN(EVAL-UeVA).

Notice that this result gives a constant delay algorithm with polynomial preprocessing for the class of unambiguous functional eVA. Instead, the algorithm in Reference [15] has linear preprocessing over documents, restricted to the case of deterministic eVA. This leaves open whether there exists a constant delay algorithm with linear preprocessing over documents for the unambiguous case.

4.2 Query Evaluation in Graph Databases

Enumerating, counting, and generating paths are relevant tasks for query evaluation in graph databases [6]. Given a finite set Σ of labels, a graph database G is a pair (V, E) where V is a finite set of vertices and $E \subseteq V \times \Sigma \times V$ is a finite set of labeled edges. Here, vertices represent pieces of data and edges specify relations between them [6]. One of the core query languages for posing queries on graph databases are **regular path queries (RPQ)**. An RPQ is a triple (x, R, y) where

x, y are variables and R is a regular expression over Σ. As usual, we denote by $\mathcal{L}(R)$ all the strings over Σ that conform to R. Given an RPQ $Q = (x, R, y)$, a graph database $G = (V, E)$, and vertices $u, v \in V$, one would like to retrieve, count, or uniformly generate all paths[3] in G going from u to v that satisfy Q. Formally, a path from u to v in G is a sequence of vertices and labels of the form $\pi = v_0, p_1, v_1, p_2, \ldots, p_n, v_n$, such that $(v_i, p_{i+1}, v_{i+1}) \in E$, $u = v_0$, and $v = v_n$. A path π is said to satisfy $Q = (x, R, y)$ if the string $p_1 p_2 \ldots p_n \in \mathcal{L}(R)$. The length of π is defined as $|\pi| = n$. Clearly, between u and v there can be an infinite number of paths that satisfy Q. For this reason, one usually wants to retrieve all paths between u and v of at most a certain length n, namely, one usually considers the set $[\![Q]\!]_n(G, u, v)$ of all paths π from u to v in G such that π satisfies Q and $|\pi| = n$. This naturally defines the following relation representing the problem of evaluating an RPQ over a graph database:

$$\text{EVAL-RPQ} = \{((Q, 0^n, G, u, v), \pi) \mid \pi \in [\![Q]\!]_n(G, u, v)\}.$$

Using this relation, fundamental problems for RPQs, such as enumerating, counting, or uniform generating paths, can be naturally represented. It is not difficult to show that EVAL-RPQ is in RelationNL, from which the following corollary can be obtained by using Theorem 3.2:

Corollary 4.3. ENUM(EVAL-RPQ) *can be enumerated with polynomial delay,* COUNT (EVAL-RPQ) *admits an FPRAS, and* GEN(EVAL-RPQ) *admits a PPLVUG.*

It is important to mention that giving a polynomial delay enumeration algorithm for EVAL-RPQ is straightforward, but the existence of an FPRAS and a PPLVUG for EVAL-RPQ was not known before when queries are part of the input (that is, in combined complexity [33]).

4.3 Binary Decision Diagrams

Binary decision diagrams are an abstract representation of Boolean functions that are widely used in computer science and have found many applications in areas like formal verification [11]. A **binary decision diagram (BDD)** is a directed acyclic graph $D = (V, E)$ where each vertex v is labeled with a variable $\text{var}(v)$ and has at most two edges going to children $\text{lo}(v)$ and $\text{hi}(v)$. Intuitively, $\text{lo}(v)$ and $\text{hi}(v)$ represent the next vertices when $\text{var}(v)$ takes values 0 and 1, respectively. D contains only two terminal, or sink vertices, labeled by 0 or 1, and one initial vertex called v_0. We assume that every path from v_0 to a terminal vertex does not repeat variables. Then given an assignment σ from the variables in D to $\{0, 1\}$, we have that σ naturally defines a path from v_0 to a terminal vertex 0 or 1. In this way, D defines a Boolean function that gives a value in $\{0, 1\}$ to each assignment σ; in particular, $D(\sigma) \in \{0, 1\}$ corresponds to the sink vertex reached by starting from v_0 and following the values in σ. For **Ordered BDDs (OBDDs)**, we also have a linear order $<$ over the variables in D such that, for every $v_1, v_2 \in V$ with v_2 a child of v_1, it holds that $\text{var}(v_1) < \text{var}(v_2)$. Notice that not necessarily all variables appear in a path from the initial vertex v_0 to a terminal vertex 0 or 1. Nevertheless, the promise in an OBDD is that variables will appear following the order $<$.

An OBDD D defines the set of assignments σ such that $D(\sigma) = 1$. Then D can be considered as a succinct representation of the set $\{\sigma \mid D(\sigma) = 1\}$, and one would like to enumerate, count, and uniformly generate assignments given D. This motivates the relation:

$$\text{EVAL-OBDD} = \{(D, \sigma) \mid D(\sigma) = 1\}.$$

Given (D, σ) in EVAL-OBDD, there is exactly one path in D that witnesses $D(\sigma) = 1$. Therefore, one can easily show that EVAL-OBDD is in RelationUL. By Theorem 3.5, we obtain that:

[3] Notice that the standard semantics for RPQs is to retrieve pair of vertices. Here, we consider a less standard semantics based on paths that is also relevant for graph databases [6, 7, 25].

Corollary 4.4. ENUM(EVAL-OBDD) *can be enumerated with constant delay, there exists a polynomial-time algorithm for* COUNT(EVAL-OBDD)*, and there exists a polynomial-time randomized algorithm for* GEN(EVAL-OBDD).

The above results are well known. Nevertheless, they show how easy and direct it is to use UL-transducers to realize the good algorithmic properties that a data structure like OBDD has.

Some non-deterministic variants of BDDs have been studied in the literature [5]. In particular, an nOBDD extends an OBDD with vertices u without variables (i.e., $\text{var}(u) = \bot$) and without labels on its children. Thus, an nOBDD is non-deterministic in the sense that given an assignment σ, there can be several paths that bring σ from the initial vertex v_0 to a terminal vertex with labeled 0 or 1. Without lost of generality, nOBDDs are assumed to be consistent in the sense that, for each σ, all paths of σ in D can reach 0 or 1, but not both.

As in the case of OBDDs, we can define a relation EVAL-nOBDD that pairs an nOBDD D with an assignment σ that evaluate D to 1 (i.e., $D(\sigma) = 1$). Contrary to OBDDs, an nOBDD loses the single witness property, and now an assignment σ can have several paths from the initial vertex to the 1 terminal vertex. Thus, it is not clear whether EVAL-nOBDD is in RelationUL. Still one can easily show that EVAL-nOBDD in RelationNL:

Corollary 4.5. ENUM(EVAL-nOBDD) *can be solved with polynomial delay,* COUNT (EVAL-nOBDD) *admits an FPRAS, and* GEN(EVAL-nOBDD) *admits a PPLVUG.*

It is important to stress that the existence of an FPRAS and a PPLVUG for EVAL-nOBDD was not known before, and one can easily show this by using NL-transducers and then applying Theorem 3.2.

5 COMPLETENESS, SELF-REDUCIBILITY, AND THEIR IMPLICATIONS FOR THE CLASS RelationUL

The goal of this section is to establish the good algorithmic properties of RelationUL, that is, to prove Theorem 3.5. To this end, we start by introducing a simple notion of reduction for the classes RelationNL and RelationUL, which will allow for much simpler proofs.

A natural question to ask is which notions of "completeness" and "reduction" are appropriate for our framework. Notions of reductions for relations have been proposed before, in particular in the context of search problems [13]. However, we do not intend to discuss them here; instead, we use an idea of completeness that is very restricted, but that turns out to be useful in this context.

Let $\mathcal{C}$ be a complexity class of relations and $R, S \in \mathcal{C}$, and recall that $W_R(x)$ is defined as the set of solutions for input x, that is, $W_R(x) = \{y \mid (x, y) \in R\}$. We say R is reducible to S if there exists a function $f : \{0,1\}^* \to \{0,1\}^*$, computable in polynomial time, such that for every $x \in \{0,1\}^*$: $W_R(x) = W_S(f(x))$. Also, if T is reducible to S for every $T \in \mathcal{C}$, then we say S is complete for $\mathcal{C}$. Notice that this definition is very restricted, since the notion of reduction requires the set of solutions to be exactly the same for both relations (it is not sufficient that they have the same size, for example). The benefit of this kind of reduction is that it preserves all the properties of efficient enumeration, counting, and uniform generation that we introduced in Sections 2 and 3, as stated in Proposition 5.1.

Proposition 5.1. *If a relation R can be reduced to a relation S, then:*

- *If* ENUM(S) *can be solved with constant (respectively, polynomial) delay, then* ENUM(R) *can be solved with constant (respectively, polynomial) delay.*
- *If there exists a polynomial-time algorithm (respectively, an FPRAS) for* COUNT(S)*, then there exists a polynomial-time algorithm (respectively, an FPRAS) for* COUNT(R).

- *If there exists a polynomial-time randomized algorithm (respectively, a PPLVUG) for* GEN(S), *then there exists a polynomial-time randomized algorithm (respectively, a PPLVUG) for* GEN(R).

Proof. We go into some detail, but the idea of the proof is very simple. Because our notion of reduction is so strong, all efficient algorithms for S apply immediately for R, provided we add a preprocessing phase where we compute a function reducing from R to S. Since that takes only polynomial time, it preserves the overall complexity of all the types of algorithms we have discussed.

Now, with more detail and formality. Since R can be reduced to S, there exist a polynomial $p(u)$ and a function f such that $W_S(f(x)) = W_R(x)$ for every input string x, and $f(x)$ can be computed in time $p(|x|)$. First, suppose ENUM(S) can be solved with constant (respectively, polynomial) delay, so there is an algorithm $\mathcal{E}$ that enumerates $W_S(f(x))$ with constant (respectively, polynomial) delay and with precomputation phase of time $q(|f(x)|)$ for some polynomial q. Now, consider the following procedure for ENUM(R) on input x: First, we compute $f(x)$ in time $p(|x|)$. Then, we run $\mathcal{E}(f(x))$, which enumerates all solutions in $W_S(f(x))$, that is, it enumerates all solutions in $W_R(x)$. So, the precomputation time of the procedure takes time $p(|x|) + q(|f(x)|) \leq p(|x|) + q(p(|x|))$, which is polynomial in $|x|$. The enumeration phase is the same as for $\mathcal{E}(f(x))$, so it has constant (respectively, polynomial) delay. We conclude that ENUM(R) can be solved with constant (respectively, polynomial) delay.

Now, suppose there exists a polynomial-time algorithm $\mathcal{A}$ for COUNT(S), let q be the polynomial that characterizes its complexity, and consider the following procedure for COUNT(R) on input x. First, we construct $f(x)$ in time $p(|x|)$. Next, we run $\mathcal{A}(f(x))$, which computes $|W_S(f(x))|$, that is, it computes $|W_R(x)|$. So, the procedure calculates $|W_R(x)|$ and takes time $p(|x|) + q(|f(x)|) \leq p(|x|) + q(p(|x|))$, which is polynomial in $|x|$. We conclude that COUNT(R) has a polynomial-time algorithm. The proof for the case of an FPRAS is completely analogous.

Finally, suppose there exists a polynomial-time randomized algorithm $\mathcal{G}$ for GEN(S), and let q be the polynomial that characterizes its complexity. Now, consider the following procedure for GEN(R) on input x. First, we construct $f(x)$ in time $p(|x|)$. Next, we run $\mathcal{G}(f(x))$, which outputs a solution from $W_S(f(x))$, that is, a solution from $W_R(x)$, uniformly at random. So, the procedure generates an element from $W_R(x)$ uniformly at random and takes time $p(|x|) + q(|f(x)|) \leq p(|x|) + q(p(|x|))$, which is polynomial in $|x|$. We conclude that GEN(R) has a polynomial-time randomized algorithm. The proof for the case of a PPLVUG is completely analogous. □

Therefore, by finding a complete relation S for a class $\mathcal{C}$ under the notion of reduction just defined, we can study the aforementioned problems for S knowing that the obtained results will extend to every relation in the class $\mathcal{C}$. In what follows, we identify complete problems for the classes RelationNL and RelationUL and use them first to establish the good algorithmic properties of RelationUL. Moreover, we prove that the identified problems are self-reducible [21], which will be useful for establishing some of the results of this section as well as for some of the results proved in Section 6 for the class RelationNL.

5.1 Complete Problems for RelationNL and RelationUL

The notion of reduction just defined is useful for us, because RelationNL and RelationUL admit natural complete problems under this notion. These complete relations are defined in terms of NFAs and we call them MEM-NFA and MEM-UFA. We already introduced MEM-NFA in Section 3, and we now define MEM-UFA as

$$\text{MEM-UFA} = \{((A, 0^k), w) \mid A \text{ is an unambiguous NFA with alphabet } \{0,1\}, w \in \{0,1\}^k \text{ and } w \text{ is accepted by } A\},$$

where an NFA is said to be unambiguous if there exists exactly one accepting run for every string accepted by it.

Recall from Section 3 that MEM-NFA $\in$ RelationNL. Besides, it is easy to see that MEM-UFA $\in$ RelationUL. To see why these relations are complete for our classes, consider the following: Take a relation R in RelationNL (the case for RelationUL is the same). We know there is an NL-transducer M that characterizes it. Now run M on some given input x. Since M works in logarithmic space, there is only a polynomial number of different configurations that M can ever be in (polynomial in $|x|$). Hence, we can consider the set of possible configurations as the states of an NFA A_x, which then has only polynomial size. The transitions of A_x are determined by the transitions between the configurations of M. Moreover, a symbol output by the transducer M is interpreted as a symbol read by the automaton A_x. In this way, A_x accepts exactly the language $W_R(x)$. We formalize this idea in the following result:

Proposition 5.2. MEM-NFA *is complete for* RelationNL *and* MEM-UFA *is complete for* RelationUL.

We will prove the result only for the case of RelationUL and MEM-UFA, as the other case is completely analogous. The following lemma is the key ingredient in our argument. The proof of this lemma is given in Appendix A.1.

Lemma 5.3. *Let R be a relation in* RelationUL. *Then there exists a polynomial-time algorithm that, given $x \in \{0,1\}^*$, produces an unambiguous NFA A_x such that $y \in W_R(x)$ if and only if y is accepted by A_x.*

Proof of Proposition 5.2. Let R be a relation in RelationUL and x be a string in $\{0,1\}^*$. We know by Lemma 5.3 that we can construct in polynomial time an unambiguous NFA A_x such that $y \in W_R(x)$ if and only if y is accepted by A_x. Now, since R is a p-relation, there exists a polynomial q such that $|y| = q(|x|)$ for all $y \in W_R(x)$. Thus, we have that all words accepted by A_x have the same length $q(|x|)$. We conclude that $W_R(x) = W_{\text{MEM-UFA}}((A_x, 0^{q(|x|)}))$. Since this works for every $R \in$ RelationUL and every input x, by definition of completeness, we deduce that MEM-UFA is complete for RelationUL. □

In Section 3, we show that SAT-DNF $\in$ RelationNL. Thus, a fundamental question is whether SAT-DNF is complete for the class RelationNL under the notion of reduction considered in this work. Notice that if this holds, then we will obtain that COUNT(SAT-DNF) is SpanL-complete under the notion of polynomial-time parsimonious reduction (introduced in Section 3). However, COUNT(SAT-DNF) is only known to be SpanL-complete under polynomial-time Turing reductions, and it is unknown whether COUNT(SAT-DNF) is complete for SpanL under polynomial-time parsimonious reductions. In fact, it is not even known whether COUNT(SAT-DNF) is SpanL-complete under some notion of reduction that preserves the existence of an FPRAS, so the existence of an FPRAS for COUNT(SAT-DNF) cannot be used to infer the existence of an FPRAS for #NFA. Hence, we leave as an open problem whether SAT-DNF is complete for RelationNL in the sense studied in this article.

5.2 MEM-NFA and MEM-UFA are Self-reducible

Self-reducibility is a property of many natural relations, and it plays a key role in proving some important results, like the tight relationship between counting and uniform generation established in Reference [21]. There are different ways of formalizing this concept, and they can get rather technical, but the intuition is pretty straightforward. We say that a (decision) problem is self-reducible if it can be solved by referring to smaller instances of the same problem. For example, SAT is self-reducible. Given a propositional formula φ, consider its satisfiability problem. We can easily reduce

that problem to smaller instances of SAT as follows: Take the first variable of φ and replace it by 0 to get a new formula φ_0. Do the same with 1 to get a new formula φ_1. Notice that φ is satisfiable if and only if φ_0 or φ_1 is satisfiable. Moreover, both φ_0 and φ_1 have one less variable than φ, so they are smaller instances.

Now, self-reducibility does not imply the existence of a polynomial-time solution for a problem, as SAT well illustrates. It is true that the instances get smaller, until they eventually become trivially easy to solve. But the number of instances is multiplied, so recursively applying self-reducibility can lead to an exponential number of smaller instances to solve. Rather than a solution method, self-reducibility is thought of as a structural feature of a problem.

Now, definitions (and proofs) of self-reducibility can get very technical, partly because they have to formalize the notion of "smaller instance." Hence, they crucially depend on the way that problems are encoded.[4] We now state the main result of this subsection.

Proposition 5.4. MEM-NFA *and* MEM-UFA *are self-reducible.*

To see the intuition behind this result, consider first a **deterministic finite automaton (DFA)** D over the alphabet $\{0,1\}$, and suppose it accepts a string $w = 0 \cdot w'$, where $w' \in \{0,1\}^*$. Then assuming that q_0 is the initial state of D, we know that the accepting run for w moves from q_0 to a state q_1 by reading symbol 0, and then it continues processing w' from q_1. Now, if we change the initial state to q_1 to get a new DFA D_0, then D_0 accepts the string w'. In other words, if $\mathcal{L}(D)$ is the language accepted by D, then we have that:

$$\mathcal{L}(D) \quad = \quad \{0 \cdot w' \mid w' \in \mathcal{L}(D_0)\} \cup \{1 \cdot w' \mid w' \in \mathcal{L}(D_1)\},$$

where DFA D_1 is defined in the same way as D_0. Besides, notice that if the length of the strings to be accepted by D is given as a parameter, as in the case of MEM-NFA, then we can assume that D does not contain any cycles, and each automaton D_i $(i = 0, 1)$ can be made smaller than D by removing q_0 and updating the transition function of D accordingly. Hence, the above equality shows that the language accepted by D can be defined in terms of the languages accepted by smaller deterministic finite automata. The same idea can be applied to an NFA N, although constructing each NFA N_i $(i = 0, 1)$ is a little more complicated, as there can be several transitions from a state that read the same symbol. Intuitively, this shows that MEM-NFA is self-reducible. The precise definition of self-reducibility (with all its technicalities) and the complete proof of Proposition 5.4 can be found in Appendix A.2.

5.3 Establishing the Good Algorithmic Properties of RelationUL

Theorem 3.5 is a consequence of Propositions 5.1 and 5.2 and the following result:

Proposition 5.5. ENUM(MEM-UFA) *can be solved with constant delay, there exists a polynomial-time algorithm for* COUNT(MEM-UFA)*, and there exists a polynomial-time randomized algorithm for* GEN(MEM-UFA).

To sum up all the results just mentioned: MEM-UFA is complete for RelationUL, it has good algorithmic properties, and our notion of reduction (and completeness) preserves all the algorithmic properties we have discussed. In what follows, we prove each of the three results stated in Proposition 5.5.

[4] Thus, saying something like "SAT is self-reducible" is slightly inaccurate. We need to specify the way in which the problem, inputs, and solutions are encoded before we can assert something like that.

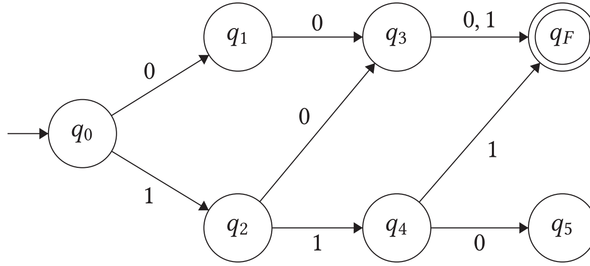

Fig. 1. Unambiguous NFA A.

5.3.1 ENUM(MEM-UFA) *Can Be Solved with Constant Delay.* We now provide a sketch of the constant delay algorithm. The idea is conceptually simple. Remember what we want: to output all strings of a certain length accepted by an unambiguous NFA, without repetition. We may use a preprocessing phase of polynomial time, but afterwards, there can be at most linear time between one string and the next.

Now, let $(A, 0^k)$ be the input, and consider Figure 1 with $k = 3$ as an example. To do constant delay enumeration, we do a depth-first traversal of the NFA, starting from the initial state. As we traverse the NFA, we read the symbols from the transitions and store them in a partial string. When the partial string reaches length k, if we happen to be in a final state, then we output the string.

Basically, that is all you have to do, but there are a few technicalities remaining. First, we mentioned depth-first traversal even though we are not analyzing a graph, but an NFA. The clarification is simple. We will use the preprocessing phase to get a labeled **directed acyclic graph (DAG)** A_{unroll} from A and k and do the depth-first traversal on A_{unroll}. The DAG A_{unroll} is obtained by first unrolling A in the following way:

(1) Cluster all final states of A into a single final state. This is easy to do: Create a new state q_F, make it the unique final state, and create an ε-transition from all previous final states to the new one.
(2) Remove all ε-transitions (this is a standard procedure for an NFA).
(3) Unroll the NFA $k + 1$ times. That is, for each state q create $k + 1$ copies $\{(q, i)\}_{i=0}^{k}$, and for each transition $q \xrightarrow{a} p$ in A, create the transitions $\{(q, i) \xrightarrow{a} (p, i + 1)\}_{i=0}^{k-1}$ in the unrolled automaton. Keep a unique initial state $(q_0, 0)$ and a unique final state (q_F, k).
(4) Remove all nodes that are not a part of an accepting run from the initial to the final state.

See Figure 2 for an example of this kind of transformation. It is easy to see that this can be done in polynomial time and that it produces a new NFA (alternatively, a labeled DAG) A_{unroll} that is still unambiguous and accepts the same words of length k as A. Since there are no ε-transitions, each string of length k accepted by A can be interpreted as a path of length k in A_{unroll} from the initial to the final state and vice versa. Thus, a depth-first traversal of A_{unroll} will go through all words of length k accepted by k, and no more.

The second technicality concerns the following question: Is the enumeration truly repetition-free? It is, for the following reason: Each path of length k from the initial state to the final state is only traversed once (by definition of a depth-first traversal of a graph). Moreover, each one of those paths corresponds to a different string, since A and A_{unroll} are both unambiguous automata.

Finally, does the enumeration phase really have constant delay? That is, does it take time $O(k)$ between enumerating one solution and the next? The answer is yes. Notice that it takes time $O(k)$ to traverse from the initial state to the final state and from one final state visit to the next, because

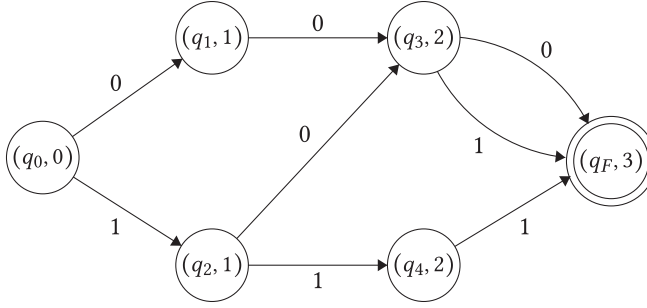

Fig. 2. Graph A_{unroll} obtained from A.

the traversal is depth-first. Also, recall that we removed (in step (4) above) all nodes that were not part of an accepting run from the initial to the final state. Thus, there is no time wasted: Each traversal to the final state produces a new string that we can output.

5.3.2 There Exists a Polynomial-time Algorithm for COUNT(MEM-UFA). Consider the graph A_{unroll} as defined in Section 5.3.1. As we already pointed out, the number of paths of length k from the initial to the final state is exactly what we want to compute: the number of strings of length k accepted by A (the unambiguity assumption is crucial here). Since A_{unroll} is a DAG, we know the number of paths between two given nodes can be computed exactly in polynomial time by dynamic programming. Hence, there exists a polynomial-time algorithm for COUNT(MEM-UFA).

5.3.3 There Exists a Polynomial-time Randomized Algorithm for GEN(MEM-UFA). Consider an input $(A, 0^k)$ for the problem GEN(MEM-UFA). Moreover, as for the case of COUNT(MEM-UFA), consider the graph A_{unroll} defined in Section 5.3.1, which can also be seen as an automaton. Assume that $(q_0, 0)$ is the initial state of A_{unroll} and that $\{(q_1, 1), \dots, (q_\ell, 1)\}$ is the set of states in A_{unroll} reachable from $(q_0, 0)$ by following an edge with label 0. Let N_0 be the number of strings of length k that start with the symbol 0 and are accepted by A. Then, we can compute N_0 in polynomial time by using the counting algorithm mentioned in the previous section, starting from each one of the states in $\{(q_1, 1), \dots, (q_\ell, 1)\}$. Notice that this algorithm works properly, as A is an unambiguous NFA. In the same way, we can compute in polynomial time the number N_1 of strings of length k that start with the symbol 1 and are accepted by A. Given N_0 and N_1, the first symbol w_1 of the string $w = w_1 \dots w_k$ to be generated is chosen according to the probabilities:

$$\mathbf{Pr}(w_1 = 0) = \frac{N_0}{N_0 + N_1} \quad \text{and} \quad \mathbf{Pr}(w_1 = 1) = \frac{N_1}{N_0 + N_1}.$$

Then the algorithm continues in the same way choosing $w_2, \dots, w_k$. It is easy to prove that this algorithm generates uniformly, at random, a string accepted by A of length k.

Notice that the previous idea is essentially the same as the one in Reference [21], that is, we use the fact that the relation MEM-UFA is self-reducible and its counting problem can be solved efficiently. However, a clarifying note should be included here. We claim a polynomial-time randomized algorithm for GEN(MEM-UFA), while an almost-uniform generator is claimed in Reference [21]. Our result is stronger for two reasons. First, we have a stronger counting result (exact polynomial-time algorithm instead of an FPRAS) to use as the basis of our uniform generation algorithm. Second, the computational model considered in Reference [21] (the Probabilistic Turing Machine) is a bit different from the one considered in this work. It cannot, for example, simulate a Bernoulli experiment with a success probability of exactly $\frac{1}{3}$. Essentially, it makes it impossible

to get an exact uniform generation algorithm. We are less strict with our computational model, so we are able to get a polynomial-time randomized algorithm for GEN(MEM-UFA).

6 #NFA ADMITS A FULLY POLYNOMIAL-TIME RANDOMIZED APPROXIMATION SCHEME AND ITS IMPLICATIONS TO THE CLASS RelationNL

The goal of this section is to provide a proof of Theorem 3.2, which considers the class RelationNL defined in terms of NL-transducers. Given that we showed in Proposition 5.2 that MEM-NFA is complete for RelationNL, we have by Propositions 5.1 that Theorem 3.2 is a consequence of the following result:

Theorem 6.1. ENUM(MEM-NFA) *can be solved with polynomial delay,* COUNT(MEM-NFA) *admits an FPRAS, and* GEN(MEM-NFA) *admits a PPLVUG.*

The existence problem for MEM-NFA has as input an NFA A and a value k given in unary (as the string 0^k), and the question to answer is whether $W_{\text{MEM-NFA}}((A, 0^k)) \neq \varnothing$ (that is, whether there are any solutions for $(A, 0^k)$ according to the relation MEM-NFA). It is easy to prove that such a task can be solved in polynomial time, as the nonemptiness problem for NFA can be solved in polynomial time. Moreover, we proved in Section 5.2 that MEM-NFA is a self-reducible relation. Given all of the above, a polynomial delay algorithm for ENUM(MEM-NFA) can be derived from the folklore result that such an enumeration algorithm exists for a self-reducible relation if the associated existence problem for this relation can be solved in polynomial time (a precise statement of this result can be found in Lemma 4.10 in Reference [30]). In this section, we focus on the remaining part of the proof of Theorem 6.1. More specifically, we provide an algorithm that approximately counts the number of words of a given length accepted by an NFA, where this length is given in unary. This constitutes an FPRAS for COUNT(MEM-NFA), as formally stated in the following theorem:

Theorem 6.2. #NFA *(and, thus,* COUNT(MEM-NFA)*) admits a fully polynomial-time randomized approximation scheme.*

The algorithm mentioned in this theorem works by simultaneously counting and doing uniform generation of solutions. Then its existence not only gives us an FPRAS for COUNT(MEM-NFA), but also a PPLVUG for GEN(MEM-NFA), as formally stated in the following theorem:

Theorem 6.3. GEN(MEM-NFA) *admits a preprocessing polynomial-time Las Vegas uniform generator.*

In the rest of this section, we prove Theorems 6.2 and 6.3. More specifically, we start by providing in Section 6.1 an overview of the algorithmic techniques used in the proof of Theorem 6.2. Then, we present in Section 6.2 the template for the FPRAS for #NFA, whose main components are given in Sections 6.3 and 6.4. A complete version of the FPRAS for #NFA is finally given in Section 6.6, where its correctness and polynomial-time complexity are established. Moreover, the proof of Theorem 6.3 is also given in Section 6.6.

6.1 An Overview of the Algorithmic Techniques

We start by providing a high-level overview of our FPRAS for the #NFA problem. To this end, we first set the necessary terminology to refer to this counting problem.

A **non-deterministic finite automaton (NFA)** A over the alphabet $\{0, 1\}$ is given as a tuple $(Q, \{0, 1\}, \Delta, I, F)$, where Q is a finite set of states, $\Delta \subseteq Q \times \{0, 1\} \times Q$ is the transition relation, $I \subseteq Q$ is a set of initial states, and $F \subseteq Q$ is a set of final states. The language of the strings in $\{0, 1\}^*$ that are accepted by A is denoted by $\mathcal{L}(A)$. Moreover, given a natural number n, the language $\mathcal{L}_n(A)$ is

defined as $\mathcal{L}(A) \cap \{0,1\}^n$. With this terminology, we define the counting problem #NFA as follows: The input of #NFA is an NFA A with m states over the alphabet $\{0,1\}$ and a natural number n, and the task is to return $|\mathcal{L}_n(A)|$. Here, n is given in unary (that is, n is given as the string 0^n).[5]

To illustrate the difficulty of #NFA, we first consider the simpler problem of counting the number of strings $|\mathcal{L}_n(A)|$ of length n contained in the language $\mathcal{L}(A)$ accepted by a ***deterministic finite automaton* (DFA)** A. Note that if $w \in \mathcal{L}_n(A)$, there is exactly one accepting path in the DFA for w. So, to count $|\mathcal{L}_n(A)|$, one can simply compute the total number of paths of length n in the DFA, which can be done in polynomial time by a dynamic program. However, if $\mathcal{L}_n(A)$ is instead the language accepted by an NFA, then $w \in \mathcal{L}_n(A)$ can have exponentially many accepting paths, and so counting paths does not lead to a good estimate of $|\mathcal{L}_n(A)|$ for an NFA.

One natural approach to overcome the aforementioned issue is to design an algorithm to estimate the *ambiguity* of the NFA. For instance, the following procedure produces an unbiased estimator of $|\mathcal{L}_n(A)|$: First, sample a random path of length n in the NFA, and let w be the string accepted on that path. Second, count the number of accepting paths P_w that w has in the NFA, and also count the total number of paths P of length n in the NFA. Repeat this process N times, and report the average value of P/P_w. The resulting estimator is indeed unbiased. However, the number of paths $P_w, P_{w'}$ can differ by an exponential factor for different strings w, w', thus the variance of this estimator is exponential. Therefore, this algorithm requires exponentially many samples to obtain a good estimate. Several other similar estimators exist (see, e.g., Reference [23]), which all unfortunately do not lead to polynomial time algorithms for the general #NFA problem.

The basic approach of our FPRAS is to incrementally estimate, for each state q in the NFA, the number of distinct strings w for which there is a path of length α from the starting states to q labeled by w. Call this set of strings $\mathcal{L}(q^\alpha)$. Our high-level approach is similar to dynamic programming. Namely, to estimate $|\mathcal{L}(q^\alpha)|$, we first estimate $|\mathcal{L}(p^{\alpha-1})|$ for each state p such that there is a transition from (p,a,q) in the NFA, where $a \in \{0,1\}$. However, one cannot simply declare

$$|\mathcal{L}(q^\alpha)| \quad = \quad \sum_{p\,:\,(p,a,q)\in\Delta} |\mathcal{L}(p^{\alpha-1})|,$$

because a single string w can be in many of the sets $\mathcal{L}(p^{\alpha-1})$, which would result in over-counting. Therefore, we must also estimate the *intersections* of the sets $\mathcal{L}(p^{\alpha-1})$. This is challenging, as these sets themselves can be exponentially large, so we cannot afford to write them down. Moreover, there are 2^m possible sets that can arise as the intersection of sets of the form $\mathcal{L}(q^\alpha)$ for $q \in Q$, thus we cannot store an estimate of each. The main insight of our FPRAS is to *sketch* the intermediate states $\mathcal{L}(p^{\alpha-1})$ of the dynamic program by replacing the set $\mathcal{L}(p^{\alpha-1})$ with a small (polynomial-sized) uniformly sampled set $S(p^{\alpha-1}) \subseteq \mathcal{L}(p^{\alpha-1})$. Here, the sketch $S(p^{\alpha-1})$ acts as a compact representation of the (possibly) larger set $\mathcal{L}(p^{\alpha-1})$. For instance, to see how such a sketch could be useful, if there were exactly two preceding states (p_1,a,q) and (p_2,a,q), to estimate the relative size of the intersection $|\mathcal{L}(p_1^{\alpha-1}) \cap \mathcal{L}(p_2^{\alpha-1})|/|\mathcal{L}(p_1^{\alpha-1})|$, it will suffice to use the approximation $\tilde{I} = |S(p_1^{\alpha-1}) \cap \mathcal{L}(p_2^{\alpha-1})|/|S(p_1^{\alpha-1})|$. Notice that the quantity $|S(p_1^{\alpha-1}) \cap \mathcal{L}(p_2^{\alpha-1})|$ can be computed in time polynomial in $|S(p_1^{\alpha-1})|$ by checking for each $w \in S(p_1^{\alpha-1})$ if w is contained in $\mathcal{L}(p_2^{\alpha-1})$, which can be accomplished in polynomial time by a membership query for NFAs. If $N(p_1^{\alpha-1}), N(p_1^{\alpha-1})$ are our estimates of $|\mathcal{L}(p_1^{\alpha-1})|, |\mathcal{L}(p_2^{\alpha-1})|$, then we can therefore obtain an estimate of $|\mathcal{L}(q^\alpha)|$ by $N(q^\alpha) = N(p_1^{\alpha-1}) + N(p_2^{\alpha-1}) - \tilde{I} \cdot N(p_1^{\alpha-1})$, avoiding the issue of overcounting the intersection.

[5]As mentioned before, it is known that #NFA belongs to #P. Notice that the fact that n is given in unary is necessary to show this property. If n is given as a binary number, then the value $|\mathcal{L}_n(A)|$ can be double exponential in the size $O(\log n)$ of this input, since $|\mathcal{L}_n(A)|$ can be equal to 2^n. Hence, #NFA cannot be in #P if the input n is given as a binary number, as if a function $f : \{0,1\}^* \to \mathbb{N}$ is in #P, then there exists a polynomial $p(u)$ such that for every $w \in \{0,1\}^*$, it holds that $f(w) \leq 2^{p(|w|)}$.

The main technical hurdle that remains is to determine how to uniformly sample a string w from a set $\mathcal{L}(q^\alpha)$ to construct our sketches $S(p^\alpha)$. This is accomplished by sampling the string w bit-by-bit. We first partition $\mathcal{L}(q^\alpha)$ into the set of strings with last bit equal to 0 and 1. We then estimate the size of both partitions and choose a partition with probability proportional to its size. Finally, we store the bit corresponding to the sampled partition, append it to a *suffix* w' of the string w, and then recurse onto the next bit. In essence, we sample a string w by growing a suffix of w.

To estimate the size of the partitions, we use our sketches $S(p^\beta)$ of $\mathcal{L}(p^\beta)$ for all $\beta \leq \alpha$ and states p. Unfortunately, because of the error in estimating the sets $|\mathcal{L}(p^\beta)|$, there will be some error in the distribution of our sampler. To correct this, and avoid and exponential propagation of this error, we use a rejection sampling technique of Jerrum, Valiant, and Vazirani [21], which normalizes the distribution and results in a perfectly uniform sample. This allows for our construction of the sketches $S(q^\alpha)$ and also gives an algorithm for the *uniform generation* of strings of length n from an NFA.

6.2 The Algorithm Template

The input of #NFA is an NFA $A = (Q, \{0,1\}, \Delta, I, F)$ with m states, a string 0^n that represents a natural number n given in unary, and an error $\varepsilon \in (0,1)$. The problem, then, is to return a value N such that N is a $(1 \pm \varepsilon)$-approximation of $|\mathcal{L}_n(A)|$, that is,

$$(1-\varepsilon)|\mathcal{L}_n(A)| \;\leq\; N \;\leq\; (1+\varepsilon)|\mathcal{L}_n(A)|.$$

Besides, such an approximation should be returned in time polynomial in m, n, and $\frac{1}{\varepsilon}$.

Our algorithm for approximating $|\mathcal{L}_n(A)|$ first involves the construction of a labeled directed acyclic graph from the NFA A. We call this graph A_{unroll}, as it is obtained by unrolling n times the NFA A. Specifically, for every state $q \in Q$ create $n+1$ copies $q^0, q^1, \ldots, q^n$ of q, and include them as vertices of A_{unroll}. Moreover, for every transition (p, b, q) in Δ, create the edge $(p^\alpha, b, q^{\alpha+1})$ in A_{unroll}, for every $\alpha \in [0, n-1]$. We refer to the set $Q^\alpha = \{q^\alpha \mid q \in Q\}$ as the αth layer of A_{unroll}. Furthermore, for every set $P \subseteq Q$, we denote by P^α the copy of P in the αth layer of A_{unroll}. This means that I^0 refers to the initial states of A at the first layer, and F^n refers to the final states of A at the last layer. For the sake of presentation, we will use the terms *vertex* and *state* interchangeably to refer to the vertices of A_{unroll}. Moreover, from now on we assume that A_{unroll} is pruned, that is, for every $q \in Q$ and every $\alpha \in [0, n]$, there exists a path from some vertex of I^0 to q^α. In other words, all states in A_{unroll} are connected to some initial state. The pruning of A_{unroll} can be done in a pre-processing step in polynomial-time in nm without changing the overall time of the algorithm. We remark that in the remainder of the section, whenever we state that we run a procedure for p^α with $p \in Q$ and α some layer, it is implicitly assumed that all pruned states p^α have already been removed. Thus, for the remainder, we will not consider the pruned states at any point, since they cannot be used to derive words of length n in the language.

Given a state q and a layer α, we define $\mathcal{L}(q^\alpha)$ as the set of all strings w such that there exists a path labeled with w from some vertex in I^0 to q^α. Notice that $|w| = \alpha$ for every $w \in \mathcal{L}(q^\alpha)$, and also that $\mathcal{L}(q^\alpha) \neq \emptyset$, since A_{unroll} is pruned. We extend this notation to every set $P \subseteq Q$, namely, $\mathcal{L}(P^\alpha) = \bigcup_{q \in P} \mathcal{L}(q^\alpha)$. The sets of strings $\mathcal{L}(P^\alpha)$ will be crucial for our algorithm. Indeed, finding an approximation for $|\mathcal{L}_n(A)|$ is reduced to finding an estimate for $|\mathcal{L}(F^n)|$, where F^n represents the set of final states of A at the last layer.

The components of our approximation algorithm are as follows: Fix the value $\kappa = \lceil \frac{nm}{\varepsilon} \rceil$ and assume that $n \geq 2$ and $m \geq 2$ (if $n \leq 1$ or $m \leq 1$, then the problem can be easily solved in polynomial time). Then for each layer α and each state q with q^α in A_{unroll}, store a number $N(q^\alpha)$ and a set $S(q^\alpha) \subseteq \mathcal{L}(q^\alpha)$ such that:

- $N(q^\alpha)$ is a $(1 \pm \kappa^{-2})^\alpha$-approximation of $|\mathcal{L}(q^\alpha)|$ and
- $S(q^\alpha)$ is a uniform sample from $\mathcal{L}(q^\alpha)$ of size $2\kappa^7$.

ALGORITHM 1: Algorithmic Template for our FPRAS

(1) Construct the labeled directed acyclic graph A_{unroll} from an input NFA A and string 0^n, where $A = (Q, \{0,1\}, \Delta, I, F)$.
(2) For layers $\alpha = 0, 1, \ldots, n$ and states $q \in Q$:
  (a) Compute $N(q^\alpha)$ given $\bigcup_{\beta=0}^{\alpha-1} \bigcup_{p \in Q} \{N(p^\beta), S(p^\beta)\}$. For $\alpha = 0$, the value $N(q^\alpha)$ is computed without any additional information.
  (b) Call a subroutine to sample polynomially many uniform elements from $\mathcal{L}(q^\alpha)$ using the value $N(q^\alpha)$ and the elements $\bigcup_{\beta=0}^{\alpha-1} \bigcup_{p \in Q} \{N(p^\beta), S(p^\beta)\}$.
  (c) Let $S(q^\alpha) \subseteq \mathcal{L}(q^\alpha)$ be the multiset of uniform samples obtained.
(3) Return $N(F^n)$ given $\bigcup_{\beta=0}^{n} \bigcup_{p \in Q} \{N(p^\beta), S(p^\beta)\}$.

For the first requirement, we mean that

$$(1 - \kappa^{-2})^\alpha |\mathcal{L}(q^\alpha)| \leq N(q^\alpha) \leq (1 + \kappa^{-2})^\alpha |\mathcal{L}(q^\alpha)|.$$

In particular, if $\alpha = 0$, then we should have that $N(q^\alpha) = |\mathcal{L}(q^\alpha)|$. For the last requirement, we mean that each $w \in S(q^\alpha)$ is a uniform and independent sample from $\mathcal{L}(q^\alpha)$. Given this condition on the samples, it is possible that we will obtain duplicates of a given $w \in \mathcal{L}(q^\alpha)$. Besides, if $|\mathcal{L}(q^\alpha)| < 2\kappa^7$, then we know that $S(q^\alpha)$ has to contain duplicate elements. Therefore, we allow $S(q^\alpha)$ to be a multiset (meaning that the strings w in $S(q^\alpha)$ are not necessarily distinct). The number $N(q^\alpha)$ and the set $S(q^\alpha)$ can be understood as a "sketch" of $\mathcal{L}(q^\alpha)$ that will be used to compute other estimates for A_{unroll}.

The algorithm proceeds like a dynamic programming algorithm, computing $N(q^\alpha)$ and $S(q^\alpha)$ for every state q^α in A_{unroll} in a breadth-first search ordering. We first compute $N(q^0), S(q^0)$ for all states q^0 at layer 0. Then, given $\bigcup_{\beta=0}^{\alpha-1} \bigcup_{p \in Q} \{N(p^\beta), S(p^\beta)\}$, we compute $N(q^\alpha), S(q^\alpha)$ for each vertex q^α. So, the value $N(q^\alpha)$ and the set $S(q^\alpha)$ are computed layer-by-layer. The final estimate for $|\mathcal{L}(F^n)|$ is $N(F^n)$. We summarize this algorithmic template in Algorithm 1. For the rest of this section, we show how to instantiate the template of our algorithm. For a layer α, we show in Section 6.3 how to compute the estimate $N(q^\alpha)$ given estimates $N(q^\beta)$ and sets $S(q^\beta)$ for all $\beta < \alpha$. In fact, for this we need to assume a strong condition (introduced in the next section), which states that the samples in our set $S(q^\alpha)$ satisfy good concentration properties. Next, given $N(q^\alpha)$ and the prior estimates $N(q^\beta)$ and sets $S(q^\beta)$, we demonstrate in Section 6.4 how to generate a uniform sample from the set $\mathcal{L}(q^\alpha)$, proving how to compute $S(q^\alpha)$. In particular, again, we will show that the strong condition used as an induction hypothesis holds for the sets $S(q^\alpha)$ with exponentially large probability over κ (Section 6.5). In the last section, we put all pieces together and show the correctness of the algorithm.

6.3 Computing an Estimate for a Set of Vertices

Recall that the input of the problem is an NFA $A = (Q, \{0,1\}, \Delta, I, F)$ with m states and a string 0^n, and that we assume that $m \geq 2$ and $n \geq 2$. Then fix a layer α and define a sketch data structure such that $\text{sketch}[\alpha] := \{N(p^\beta), S(p^\beta)\}_{p \in Q, \beta \leq \alpha}$. Moreover, assume that $\text{sketch}[\alpha]$ has already been computed. In particular, $N(p^\beta)$ is a $(1 \pm \kappa^{-2})^\beta$-approximation of $|\mathcal{L}(p^\beta)|$, and $S(p^\beta)$ is a uniform sample from $\mathcal{L}(p^\beta)$ of size $2\kappa^7$ for each $\beta \leq \alpha$. The goal of this section is twofold: We first show how to compute an estimate of $|\mathcal{L}(P^\alpha)|$ for every $P \subseteq Q$, which is denoted by $N(P^\alpha)$, and then we show how to compute an estimate for $N(q^{\alpha+1})$. These values $N(P^\alpha)$ will play a crucial role for

computing not only $N(q^{\alpha+1})$, but also the set of uniform samples $S(q^{\alpha+1})$ and the final estimate $N(F^n)$ for $|\mathcal{L}(F^n)|$ (see Sections 6.4 and 6.5).

Let P be a non-empty subset of Q, and suppose that we want to find an estimate $N(P^\alpha)$ for $|\mathcal{L}(P^\alpha)|$. If A is deterministic, then the sets $\{\mathcal{L}(p^\alpha) \mid p \in P\}$ are disjoint, and then we can easily compute the size of $|\mathcal{L}(P^\alpha)|$ as $\sum_{p\in P} |\mathcal{L}(p^\alpha)|$. Unfortunately, given that A can be non-deterministic, this sum will over-approximate the size of $|\mathcal{L}(p^\alpha)|$, and we need to find a way to deal with the intersections of the sets $\{\mathcal{L}(p^\alpha) \mid p \in P\}$. For this, fix a total order $\prec$ over the set P, and consider the following way to compute $|\mathcal{L}(P^\alpha)|$:

$$|\mathcal{L}(P^\alpha)| \;=\; \sum_{p\in P} |\mathcal{L}(p^\alpha)| \cdot \frac{|\mathcal{L}(p^\alpha) \smallsetminus \bigcup_{q\in P\,:\,q\prec p} \mathcal{L}(q^\alpha)|}{|\mathcal{L}(p^\alpha)|}. \tag{†}$$

With the ratio $|\mathcal{L}(p^\alpha) \smallsetminus \bigcup_{q\in P\,:\,q\prec p} \mathcal{L}(q^\alpha)|/|\mathcal{L}(p^\alpha)|$, we removed from $\mathcal{L}(p^\alpha)$ its intersection with all sets $\mathcal{L}(q^\alpha)$ such that $q \prec p$. In fact, one can easily check that $|\mathcal{L}(P^\alpha)| = \sum_{p\in P} |\mathcal{L}(p^\alpha) \smallsetminus \bigcup_{q\in P\,:\,q\prec p} \mathcal{L}(q^\alpha)|$ and, thus, equation (†) trivially holds. We call the above ratio the *intersection rate* of p^α in P given $\prec$ (or just the intersection rate of p^α).

Inspired by equation (†), we can estimate $|\mathcal{L}(P^\alpha)|$ by using $N(p^\alpha)$ to estimate $|\mathcal{L}(p^\alpha)|$ and $S(p^\alpha)$ to estimate the intersection rate of p^α. More precisely, we define the estimate $N(P^\alpha)$ for $|\mathcal{L}(P^\alpha)|$ as follows:

$$N(P^\alpha) \;=\; \sum_{p\in P} N(p^\alpha) \cdot \frac{|S(p^\alpha) \smallsetminus \bigcup_{q\in P\,:\,q\prec p} \mathcal{L}(q^\alpha)|}{|S(p^\alpha)|}. \tag{‡}$$

It is important to note that $N(P^\alpha)$ can be computed in polynomial time in the size of sketch$[\alpha]$. Indeed, the set $S(p^\alpha) \smallsetminus \bigcup_{q\in P\,:\,q\prec p} \mathcal{L}(q^\alpha)$ can be computed by iterating over each string $w \in S(p^\alpha)$ and checking whether $w \in \mathcal{L}(\{q^\alpha \mid q \in P \text{ and } q \prec p\})$. Given that verifying if a string is in $\mathcal{L}(\{q^\alpha \mid q \in P \text{ and } q \prec p\})$ can be done in polynomial time, computing $N(P^\alpha)$ takes polynomial time as well. We call the ratio $|S(p^\alpha) \smallsetminus \bigcup_{q\in P\,:\,q\prec p} \mathcal{L}(q^\alpha)|/|S(p^\alpha)|$ the *estimate of the intersection rate of* p^α.

To show that $N(P^\alpha)$ is a good estimate for $|\mathcal{L}(P^\alpha)|$, we need that the estimate of the intersection rate is a good approximation of the real intersection rate in each layer. By a good approximation, we mean that the following condition holds at level α:

$$\mathcal{E}(\alpha) \;:=\; \forall q \in Q\ \forall P \subseteq Q. \quad \left| \frac{|\mathcal{L}(q^\alpha) \smallsetminus \bigcup_{p\in P} \mathcal{L}(p^\alpha)|}{|\mathcal{L}(q^\alpha)|} - \frac{|S(q^\alpha) \smallsetminus \bigcup_{p\in P} \mathcal{L}(p^\alpha)|}{|S(q^\alpha)|} \right| \;<\; \frac{1}{\kappa^3}.$$

This condition is crucial for the next results, and most of our analysis in this and next section will assume that this condition holds. Towards the end, in Section 6.6, we will show that, by Hoeffding's inequality, the condition $\mathcal{E}(\alpha)$ holds for all layers α with exponentially high probability over κ. Next, we prove that, if condition $\mathcal{E}(\alpha)$ holds, then $N(P^\alpha)$ is a good estimate for $|\mathcal{L}(P^\alpha)|$.

Proposition 6.4. *Assume that $\mathcal{E}(\alpha)$ holds and $N(p^\alpha)$ is a $(1 \pm \kappa^{-2})^\alpha$-approximation of $|\mathcal{L}(p^\alpha)|$ for every $p \in Q$. Then $N(P^\alpha)$ is a $(1 \pm \kappa^{-2})^{\alpha+1}$-approximation of $|\mathcal{L}(P^\alpha)|$ for every $P \subseteq Q$.*

Proof. Given that condition $\mathcal{E}(\alpha)$ holds, we know that for each $p \in P$:

$$\frac{|\mathcal{L}(p^\alpha) \smallsetminus \bigcup_{q\in P\,:\,q\prec p} \mathcal{L}(q^\alpha)|}{|\mathcal{L}(p^\alpha)|} - \kappa^{-3} \;<\; \frac{|S(p^\alpha) \smallsetminus \bigcup_{q\in P\,:\,q\prec p} \mathcal{L}(q^\alpha)|}{|S(p^\alpha)|} \;<\; \frac{|\mathcal{L}(p^\alpha) \smallsetminus \bigcup_{q\in P\,:\,q\prec p} \mathcal{L}(q^\alpha)|}{|\mathcal{L}(p^\alpha)|} + \kappa^{-3}.$$

Moreover, given that $N(p^\alpha)$ is a $(1 \pm \kappa^{-2})^\alpha$-approximation of $|\mathcal{L}(p^\alpha)|$, it holds that:

$$(1-\kappa^{-2})^\alpha |\mathcal{L}(p^\alpha)| \;\leq\; N(p^\alpha) \;\leq\; (1+\kappa^{-2})^\alpha |\mathcal{L}(p^\alpha)|.$$

Putting these two bounds together, we obtain the following bounds from the definition of $N(P^\alpha)$ in equation (‡):

$$(1-\kappa^{-2})^\alpha \sum_{p \in P} \left(|\mathcal{L}(p^\alpha) \setminus \bigcup_{q \in P\,:\, q \prec p} \mathcal{L}(q^\alpha)| - \kappa^{-3} |\mathcal{L}(p^\alpha)| \right) \;<\; N(P^\alpha)$$

$$\;<\; (1+\kappa^{-2})^\alpha \sum_{p \in P} \left(|\mathcal{L}(p^\alpha) \setminus \bigcup_{q \in P\,:\, q \prec p} \mathcal{L}(q^\alpha)| + \kappa^{-3} |\mathcal{L}(p^\alpha)| \right).$$

Recall from the discussion of the intersection rate that $|\mathcal{L}(P^\alpha)| = \sum_{p \in P} |\mathcal{L}(p^\alpha) \setminus \bigcup_{q \in P\,:\, q \prec p} \mathcal{L}(q^\alpha)|$. Moreover, given that $\mathcal{L}(p^\alpha) \subseteq \mathcal{L}(P^\alpha)|$ and $|P| \leq m \leq \kappa$, we have that $\sum_{p \in P} |\mathcal{L}(p^\alpha)| \leq \sum_{p \in P} |\mathcal{L}(P^\alpha)| = |P| \cdot |\mathcal{L}(P^\alpha)| \leq \kappa \cdot |\mathcal{L}(P^\alpha)|$. Replacing both statements in the previous inequality, we obtain

$$(1-\kappa^{-2})^\alpha (|\mathcal{L}(P^\alpha)| - \kappa^{-3} \cdot \kappa |\mathcal{L}(P^\alpha)|) \;<\; N(P^\alpha) \;<\; (1+\kappa^{-2})^\alpha (|\mathcal{L}(P^\alpha)| + \kappa^{-3} \cdot \kappa |\mathcal{L}(P^\alpha)|),$$

which is equivalent to

$$(1-\kappa^{-2})^{\alpha+1} |\mathcal{L}(P^\alpha)| \;<\; N(P^\alpha) \;<\; (1+\kappa^{-2})^{\alpha+1} |\mathcal{L}(P^\alpha)|.$$

This concludes the proof of the proposition. □

With the estimates of $|\mathcal{L}(P^\alpha)|$ for every $P \subseteq Q$ at the αth layer, we are ready to give a good estimate for the size $|\mathcal{L}(q^{\alpha+1})|$ of a single vertex in the next layer $\alpha + 1$. Let $q^{\alpha+1}$ be an arbitrary vertex at layer $\alpha + 1$. For $b \in \{0, 1\}$, define the set of vertices $R_b = \{p^\alpha \in Q^\alpha \mid (p^\alpha, b, q^{\alpha+1})$ is an edge in $A_{unroll}\}$, namely, the set of all vertices in the αth layer from which $q^{\alpha+1}$ can be reached by reading symbol b. Notice that sets R_0 and R_1 partition $\mathcal{L}(q^{\alpha+1})$ in the following sense:

$$\mathcal{L}(q^{\alpha+1}) \;=\; \mathcal{L}(R_0) \cdot \{0\} \uplus \mathcal{L}(R_1) \cdot \{1\}, \tag{1}$$

where given two sets S_1, S_2 of strings, $S_1 \cdot S_2$ is defined as the set consisting of the concatenation of each string of S_1 with each string of S_2 (in particular, $\mathcal{L}(R_b) \cdot \{b\} = \{w \in \{0,1\}^* \mid w = v \cdot b$ with $v \in \mathcal{L}(R_b)\}$ for $b \in \{0, 1\}$). Equation (1) implies that $|\mathcal{L}(q^{\alpha+1})| = |\mathcal{L}(R_0)| + |\mathcal{L}(R_1)|$. Notice that, if we assume $\mathcal{E}(\alpha)$ holds, then by Proposition 6.4, we have that $N(R_b)$ is a $(1 \pm \kappa^{-2})^{\alpha+1}$-approximation of $|\mathcal{L}(R_b)|$ for $b \in \{0, 1\}$, from which we obtain that $N(q^{\alpha+1}) = N(R_0) + N(R_1)$ is a $(1 \pm \kappa^{-2})^{\alpha+1}$-approximation of $|\mathcal{L}(q^{\alpha+1})|$. Therefore, we can derive an estimate $N(q^{\alpha+1})$ for $|\mathcal{L}(q^{\alpha+1})|$ by using previous estimates $\{p^\alpha\}_{p \in Q}$.

Note that the computation of $N(q^{\alpha+1})$ is deterministic by assuming that $\mathcal{E}(\beta)$ holds for all $\beta \leq \alpha$. Specifically, the estimates $N(q^0)$ are exact for the initial layer. Next, for each layer α, we assume that $\mathcal{E}(\alpha)$ holds and we can compute $N(q^{\alpha+1})$ by using $\{N(p^\alpha)\}_{p \in Q}$ (in fact, by using $\{N(P^\alpha)\}_{P \subseteq Q}$). Then, we assume that $\mathcal{E}(\alpha + 1)$ holds and so on. Therefore, by filling the sets $\{S(p^\beta)\}_{p \in Q, \beta \leq \alpha}$ with uniform samples and assuming that $\mathcal{E}(\beta)$ holds for all $\beta \leq \alpha$, we can compute each estimate $N(q^{\alpha+1})$. Moreover, we can guarantee that it is a $(1 \pm \kappa^{-2})^{\alpha+1}$-approximation of $|\mathcal{L}(q^{\alpha+1})|$. We summarize this fact in the following proposition:

Proposition 6.5. *Assume that $\mathcal{E}(\beta)$ holds for all $\beta \leq \alpha$. Then $N(p^{\alpha+1})$ is a $(1 \pm \kappa^{-2})^{\alpha+1}$-approximation of $|\mathcal{L}(p^{\alpha+1})|$ for every $p \in Q$.*

After all, at some point, we will reach the last layer n, and we would like to compute the $(1 \pm \varepsilon)$-approximation for $|\mathcal{L}_n(A)|$. For this, we can use $N(F^n)$ for estimating $|\mathcal{L}_n(A)|$, which achieves the ultimate goal of our algorithm.

Proposition 6.6. *If $\mathcal{E}(\beta)$ holds for all $\beta \leq n$, then $N(F^n)$ is a $(1 \pm \varepsilon)$-approximation for $|\mathcal{L}_n(A)|$.*

Proof. Assume that $N(F^n)$ is a $(1 \pm \kappa^{-2})^{n+1}$-approximation of $|\mathcal{L}(F^n)| = |\mathcal{L}_n(A)|$, that is,

$$(1-\kappa^{-2})^{n+1}|\mathcal{L}_n(A)| \;\leq\; N(F^n) \;\leq\; (1+\kappa^{-2})^{n+1}|\mathcal{L}_n(A)|.$$

But we have that:

$$\begin{aligned}
(1+\kappa^{-2})^{n+1} &\leq \left(1+\left(\frac{\varepsilon}{mn}\right)^2\right)^{n+1} \\
&= \left[\left(1+\left(\frac{1}{(\frac{nm}{\varepsilon})^2}\right)\right)^{(\frac{nm}{\varepsilon})^2}\right]^{\frac{(n+1)\varepsilon^2}{n^2m^2}} \\
&\leq e^{\frac{\varepsilon^2}{m^2}} \\
&\leq 1+2\frac{\varepsilon^2}{m^2} \quad \text{since } e^x \leq (1+2x) \text{ for } x \in [0,1] \\
&= 1+\varepsilon\cdot\frac{2\varepsilon}{m^2} \\
&\leq 1+\varepsilon \qquad \text{since } m \geq 2 \text{ and } \varepsilon \in (0,1),
\end{aligned}$$

and we also have that:

$$\begin{aligned}
(1-\kappa^{-2})^{n+1} &\geq \left(1-\left(\frac{\varepsilon}{mn}\right)^2\right)^{n+1} \\
&= \left[\left(1-\left(\frac{1}{(\frac{nm}{\varepsilon})^2}\right)\right)^{(\frac{nm}{\varepsilon})^2}\right]^{\frac{(n+1)\varepsilon^2}{n^2m^2}} \\
&\geq (e^{-2})^{\frac{\varepsilon^2}{m^2}} \quad \text{since } \left(1-\frac{1}{x}\right)^x \geq e^{-2} \text{ for } x \geq 2 \\
&\geq 1-\frac{2\varepsilon^2}{m^2} \qquad \text{since } e^{-x} \geq 1-x \text{ for } x \geq 0 \\
&= 1-\varepsilon\cdot\frac{2\varepsilon}{m^2} \\
&\geq 1-\varepsilon \qquad \text{since } m \geq 2 \text{ and } \varepsilon \in (0,1).
\end{aligned}$$

Thus, we conclude that:

$$(1-\varepsilon)|\mathcal{L}_n(A)| \;\leq\; N(F^n) \;\leq\; (1+\varepsilon)|\mathcal{L}_n(A)|. \qquad \square$$

In the following section, we show how to compute the set $S(q^{\alpha+1})$ using sketch$[\alpha]$, namely, how to generate a uniform sample from $\mathcal{L}(q^{\alpha+1})$. Specifically, we show that, assuming $\mathcal{E}(\beta)$ holds for all $\beta \leq \alpha$, we can obtain uniform samples from the sets $\mathcal{L}(q^{\alpha+1})$ such that property $\mathcal{E}(\alpha+1)$ will hold with high probability.

6.4 Uniform Sampling from a Vertex

To carry out our main approximation algorithm, we must implement the algorithm template given in Algorithm 1, whose input is assumed to be an NFA $A = (Q, \{0,1\}, \Delta, I, F)$ with m states and a string 0^n, where $m \geq 2$ and $n \geq 2$. In the previous section, we implemented Step 2(a) of this algorithm and, thus, the goal of this section is to implement the sampling subroutine in Step 2(b). This procedure is based on a sample technique proposed in Reference [21], but modified to suit our setting.

Take a state $q \in Q$ and layer $\alpha \leq n$, and assume that for all layers $\beta < \alpha$ the condition $\mathcal{E}(\beta)$ holds. Notice that by Proposition 6.5, once we have $\mathcal{E}(\beta)$ and estimates for all levels $\beta < \alpha$, we immediately get the estimates $N(p^\alpha)$ for the level α as well.

The procedure to sample a uniform element of the set $\mathcal{L}(q^\alpha)$ is as follows: We initialize a string w^α to be the empty string. Then, we construct a sequence of strings $w^\alpha, w^{\alpha-1}, \ldots, w^1, w^0$, where each element w^β is of the form $b_\beta \cdot w^{\beta+1}$ with $b_\beta \in \{0,1\}$, and we define the result of the sample procedure to be w^0. In other words, we sample a string w^0 of $\mathcal{L}(q^\alpha)$ by building a *suffix* of the sample, bit-by-bit. To ensure that w^0 is an element of $\mathcal{L}(q^\alpha)$ chosen uniformly, we also consider a sequence of sets $P^\alpha, P^{\alpha-1}, \ldots, P^1, P^0$ constructed as follows: The first set is $P^\alpha = \{q^\alpha\}$. Then, we consider the set of vertices at layer $\alpha - 1$ that can reach the set P^α by reading letter b, namely, for $b \in \{0,1\}$ define:

$$P^\alpha_b = \{p^{\alpha-1} \in Q^{\alpha-1} \mid \text{there exists } r^\alpha \in P^\alpha \text{ such that } (p^{\alpha-1}, b, r^\alpha) \text{ is an edge in } A_{unroll}\}.$$

Notice that, although we use the superscript α, the set P^α_b is a subset of vertices in the $(\alpha-1)$-th layer. Similar to the previous section, the sets P^α_0 and P^α_1 induce a partition of the set $\mathcal{L}(P^\alpha)$ in the following sense:

$$\mathcal{L}(P^\alpha) = \mathcal{L}(P^\alpha_0) \cdot \{0\} \uplus \mathcal{L}(P^\alpha_1) \cdot \{1\}.$$

Therefore, our sampling algorithm estimates the size $N(P^\alpha_b)$ of $\mathcal{L}(P^\alpha_b)$ for $b \in \{0,1\}$ and chooses one of P^α_0, P^α_1 with probability proportional to its size, namely, $N(P^\alpha_0)/(N(P^\alpha_0) + N(P^\alpha_1))$ and $N(P^\alpha_1)/(N(P^\alpha_0) + N(P^\alpha_1))$. Say we choose P^α_b. Then, we define $b_{\alpha-1} = b$, append the bit $b_{\alpha-1}$ as a prefix of w^α to obtain $w^{\alpha-1} = b_{\alpha-1} \cdot w^\alpha$, define $P^{\alpha-1}$ as P^α_b, and continue with the recursion on $w^{\alpha-1}$ and $P^{\alpha-1}$. Hence, we have that P^β is the set of vertices such that there exists a path labeled by w^β that connects some state of P^β with q^α. Notice that $\mathcal{L}(P^\beta) \neq \emptyset$ for every layer β (and, thus, $P^\beta \neq \emptyset$). Indeed, given that A_{unroll} is pruned, we know that $\mathcal{L}(P^\alpha) = \mathcal{L}(\{q^\alpha\}) \neq \emptyset$. By induction, if for some level β we have that $P^\beta_0 = \emptyset$ (similar when $P^\beta_1 = \emptyset$), then $\mathcal{L}(P^\beta_1) \cdot \{1\} = \mathcal{L}(P^\beta)$ and the next level $\beta - 1$ will be chosen with probability 1. In particular, $\mathcal{L}(P^\beta_1) = \mathcal{L}(P^{\beta-1}) \neq \emptyset$.

Since there could be an error in estimating the sizes of the partitions, it may be the case that some items were chosen with slightly larger probability than others. To remedy this and obtain a perfectly uniform sampler, at every step of the algorithm, we store the probability with which we chose a partition. Thus, at the end, we have computed exactly the probability φ with which we sampled the string w. We can then reject this sample with probability proportional to φ, which gives a perfect sampler. As long as no string is too much more likely than another to be sampled, the probability of rejection will be a constant, and we can simply run our sampler $O(\log(\frac{1}{\mu}))$-times to get a sample with probability $1 - \mu$ for every $\mu > 0$.

This procedure then is given in Algorithm 2. We call it with the initial parameters **Sample**$(\alpha, \{q^\alpha\}, \lambda, \varphi_0)$, where λ is the empty string, corresponding to the goal of sampling a uniform element of $\mathcal{L}(P^\alpha) = \mathcal{L}(q^\alpha)$. Here, φ_0 is a value that we will later choose. Notice that at every step of Algorithm 2, we have that $|\mathcal{L}(P^\beta)|$ is precisely the number of strings in $\mathcal{L}(q^\alpha)$ that have the suffix w^β, as $\mathcal{L}(P^\beta)$ is the set of strings x such that $x \cdot w^\beta \in \mathcal{L}(q^\alpha)$. Observe then that the set P^β depends on the random string w^β, so in fact we could write $P^\beta_{w^\beta}$ instead of P^β. For notational simplicity, we omit the subscript, and it is then understood that P^β is a function of w^β.

To get some intuition of Algorithm 2, assume for the moment that we can compute each p_b exactly, namely, $p_b = |\mathcal{L}(P^\beta_b)|/|\mathcal{L}(P^\beta)|$. Now the probability of choosing a given element $x \in \mathcal{L}(q^\alpha)$ can be computed as follows: Ignoring for a moment the possibility of returning **fail**, we have that

ALGORITHM 2: Sample$(\beta, P^\beta, w^\beta, \varphi)$

(1) If $\beta = 0$, then with probability φ return w^0, otherwise return **fail**.
(2) Else, compute the set $P^\alpha_b = \{p^{\alpha-1} \in Q^{\alpha-1} \mid$ there exists $r^\alpha \in P^\alpha$ such that $(p^{\alpha-1}, b, r^\alpha)$ is an edge in $A_{unroll}\}$ for every $b \in \{0,1\}$.
(3) Choose a partition $b \in \{0,1\}$ with probability $p_b = \dfrac{N(P^\beta_b)}{N(P^\beta_0) + N(P^\beta_1)}$.
(4) Set $P^{\beta-1} = P^\beta_b$, and $w^{\beta-1} = b \cdot w^\beta$.
(5) Return **Sample**$(\beta - 1, P^{\beta-1}, w^{\beta-1}, \frac{\varphi}{p_b})$.

w^0 is the string returned by **Sample**$(\alpha, \{q^\alpha\}, \lambda, \varphi_0)$. Thus, the probability we choose x is:

$$\mathbf{Pr}(w^0 = x) \;=\; \frac{|\mathcal{L}(P^{\alpha-1})|}{|\mathcal{L}(P^\alpha)|} \cdot \frac{|\mathcal{L}(P^{\alpha-2})|}{|\mathcal{L}(P^{\alpha-1})|} \cdot \frac{|\mathcal{L}(P^{\alpha-3})|}{|\mathcal{L}(P^{\alpha-2})|} \cdot \ldots \cdot \frac{|\mathcal{L}(P^1)|}{|\mathcal{L}(P^2)|} \cdot \frac{1}{|\mathcal{L}(P^1)|} \;=\; \frac{1}{|\mathcal{L}(P^\alpha)|}.$$

Now at the point of return, we also have that $\varphi = \varphi_0/\mathbf{Pr}(w^0 = x)$. Thus, if $\varphi_0/\mathbf{Pr}(w^0 = x) \leq 1$, then the probability that x is output is simply φ_0. The following is then easily seen:

Fact 1. Assume that each probability p_b in Algorithm 2 satisfies that

$$p_b \;=\; \frac{|\mathcal{L}(P^\beta_b)|}{|\mathcal{L}(P^\beta)|}.$$

If $0 < \varphi_0 \leq \frac{1}{|\mathcal{L}(P^\alpha)|}$ and $w^0 \neq$ **fail** is the output of Algorithm 2, then for every $x \in \mathcal{L}(P^\alpha)$, it holds that $\mathbf{Pr}(w^0 = x) = \varphi_0$. Moreover, the algorithm outputs $w^0 =$ **fail** with probability $1 - |\mathcal{L}(P^\alpha)|\varphi_0$.

This shows that, conditioned on not failing, the above is a uniform sampler. Repeating the procedure $\ell \cdot (|\mathcal{L}(P^\alpha)|\varphi_0)^{-1}$ times, we get a sample with probability $1 - e^{-\ell}$, since:

$$\begin{aligned}(1 - |\mathcal{L}(P^\alpha)|\varphi_0)^{\ell \cdot (|\mathcal{L}(P^\alpha)|\varphi_0)^{-1}} &\leq (e^{-|\mathcal{L}(P^\alpha)|\varphi_0})^{\ell \cdot (|\mathcal{L}(P^\alpha)|\varphi_0)^{-1}} \qquad \text{(by using } (1-x) \leq e^{-x}\text{)} \\ &= e^{-|\mathcal{L}(P^\alpha)|\varphi_0 \cdot \ell \cdot (|\mathcal{L}(P^\alpha)|\varphi_0)^{-1}} = e^{-\ell}.\end{aligned}$$

However, Fact 1 was obtained under the strong assumption that each probability p_b can be computed exactly. Hence, in what follows, we focus on showing that with high probability the same result holds if we approximate p_b with $N(P^\beta_b)/(N(P^\beta_0) + N(P^\beta_1))$ (instead of assuming that $p_b = |\mathcal{L}(P^\beta_b)|/|\mathcal{L}(P^\beta)|$).

Proposition 6.7. *Assume that condition $\mathcal{E}(\beta)$ holds for every layer $\beta < \alpha$. If $w \neq$ **fail** is the output of* **Sample**$(\alpha, \{q^\alpha\}, \lambda, \frac{e^{-5}}{N(q^\alpha)})$*, then for every $x \in \mathcal{L}(q^\alpha)$:*

$$\mathbf{Pr}(w = x) \;=\; \frac{e^{-5}}{N(q^\alpha)}.$$

Moreover, the algorithm outputs **fail** *with probability at most $1 - e^{-9}$. Thus, conditioned on not failing,* **Sample**$(\alpha, \{q^\alpha\}, \lambda, \frac{e^{-5}}{N(q^\alpha)})$ *returns a uniform element $x \in \mathcal{L}(q^\alpha)$.*

Proof. First, we show that every recursive call to **Sample** satisfies that $\varphi \in (0,1)$. Since $\varphi_0 = \frac{e^{-5}}{N(q^\alpha)} > 0$ and with each call φ does not decrease (because it is divided by a probability), we know that $\varphi > 0$ at each subsequent call. It remains to show that $\varphi < 1$ for every recursive call to the

Sample procedure. Since φ does not decrease after each recursive call, it suffices to show this for the final value of φ. Notice that at the call β, with β from n to 1, we have that φ is divided by a factor

$$\frac{N\left(P^{\beta}_{w[\beta]}\right)}{N\left(P^{\beta}_{0}\right)+N\left(P^{\beta}_{1}\right)},$$

where $w[\beta]$ is the βth letter of w. So, in the final call, φ has the value:

$$\begin{aligned}\varphi &= \left(\prod_{\beta=1}^{\alpha}\frac{N\left(P^{\beta}_{w[\beta]}\right)}{N\left(P^{\beta}_{0}\right)+N\left(P^{\beta}_{1}\right)}\right)^{-1}\cdot\varphi_0\\ &= \left(\prod_{\beta=1}^{\alpha}\frac{N\left(P^{\beta}_{0}\right)+N\left(P^{\beta}_{1}\right)}{N\left(P^{\beta}_{w[\beta]}\right)}\right)\cdot\frac{e^{-5}}{N(q^{\alpha})}.\end{aligned}$$

Given that condition $\mathcal{E}(\beta)$ holds for every layer $\beta < \alpha$, by Proposition 6.4, we know that $N(P^{\beta}_{b})$ is a $(1\pm\kappa^{-2})^{\beta}$-approximation of $|\mathcal{L}(P^{\beta}_{b})|$ for all $b\in\{0,1\}$ and $\beta\in[1,\alpha]$ (recall that P^{β}_{b} is a subset of states at layer $\beta-1$). It follows that at the final recursive call to **Sample**, we have that:

$$\begin{aligned}\varphi &= \left(\prod_{\beta=1}^{\alpha}\frac{N\left(P^{\beta}_{0}\right)+N\left(P^{\beta}_{1}\right)}{N\left(P^{\beta}_{w[\beta]}\right)}\right)\cdot\frac{e^{-5}}{N(q^{\alpha})}\\ &\le \left(\prod_{\beta=1}^{\alpha}\frac{(1+\kappa^{-2})^{\beta}\cdot\left(\left|\mathcal{L}\left(P^{\beta}_{0}\right)\right|+\left|\mathcal{L}\left(P^{\beta}_{1}\right)\right|\right)}{(1-\kappa^{-2})^{\beta}\cdot\left|\mathcal{L}\left(P^{\beta}_{w[\beta]}\right)\right|}\right)\cdot\frac{e^{-5}}{N(q^{\alpha})}\\ &= \left(\prod_{\beta=1}^{\alpha}\frac{(1+\kappa^{-2})^{\beta}}{(1-\kappa^{-2})^{\beta}}\right)\cdot\left(\prod_{\beta=1}^{\alpha}\frac{\left(\left|\mathcal{L}\left(P^{\beta}_{0}\right)\right|+\left|\mathcal{L}\left(P^{\beta}_{1}\right)\right|\right)}{\left|\mathcal{L}\left(P^{\beta}_{w[\beta]}\right)\right|}\right)\cdot\frac{e^{-5}}{N(q^{\alpha})}.\end{aligned}$$

Recall that $|\mathcal{L}(P^{\beta}_{0})|+|\mathcal{L}(P^{\beta}_{1})| = |\mathcal{L}(P^{\beta+1}_{w[\beta+1]})|$ for every $\beta\in[1,\alpha-1]$. Also note that $|\mathcal{L}(P^{1}_{w[1]})| = 1$, since $\mathcal{L}(P^{1})$ is the set of strings $x\in\{0,1\}$ such that $x\cdot w^{1}\in\mathcal{L}(q^{\alpha})$, and $\mathcal{L}(P^{1}_{w[1]})$ is the subset of $\mathcal{L}(P^{1})$ with the last bit equal to $w[1]$ (of which there is just one). Thus, given that $|\mathcal{L}(q^{\alpha})| = |\mathcal{L}(P^{\alpha})| = |\mathcal{L}(P^{\alpha}_{0})|+|\mathcal{L}(P^{\alpha}_{1})|$, we have that:

$$\prod_{\beta=1}^{\alpha}\frac{\left(\left|\mathcal{L}\left(P^{\beta}_{0}\right)\right|+\left|\mathcal{L}\left(P^{\beta}_{1}\right)\right|\right)}{\left|\mathcal{L}\left(P^{\beta}_{w[\beta]}\right)\right|} = |\mathcal{L}(q^{\alpha})|,$$

and so

$$\begin{aligned}\varphi &\le \left(\prod_{\beta=1}^{\alpha}\frac{(1+\kappa^{-2})^{\beta}}{(1-\kappa^{-2})^{\beta}}\right)\cdot|\mathcal{L}(P^{\alpha})|\cdot\frac{e^{-5}}{N(q^{\alpha})}\\ &= \left(\frac{1+\kappa^{-2}}{1-\kappa^{-2}}\right)^{\frac{\alpha(\alpha+1)}{2}}\cdot|\mathcal{L}(q^{\alpha})|\cdot\frac{e^{-5}}{N(q^{\alpha})}\\ &\le \left(\frac{1+\kappa^{-2}}{1-\kappa^{-2}}\right)^{\alpha^2}\cdot|\mathcal{L}(q^{\alpha})|\cdot\frac{e^{-5}}{N(q^{\alpha})}\end{aligned}$$

$$\leq \left(\frac{1+\kappa^{-2}}{1-\kappa^{-2}}\right)^{n^2} \cdot |\mathcal{L}(q^\alpha)| \cdot \frac{e^{-5}}{N(q^\alpha)}.$$

Furthermore, $N(q^\alpha)$ is a $(1 \pm \kappa^{-2})^\alpha$-approximation of $|\mathcal{L}(q^\alpha)|$ by Proposition 6.5. Then, we know that $N(q^\alpha) \geq (1-\kappa^{-2})^\alpha |\mathcal{L}(q^\alpha)| \geq (1-\kappa^{-2})^{n^2} |\mathcal{L}(q^\alpha)|$ and, therefore,

$$\begin{aligned}
\varphi &\leq \left(\frac{1+\kappa^{-2}}{1-\kappa^{-2}}\right)^{n^2} \cdot |\mathcal{L}(q^\alpha)| \cdot \frac{e^{-5}}{N(q^\alpha)} \\
&\leq \left(\frac{1+\kappa^{-2}}{1-\kappa^{-2}}\right)^{n^2} \cdot |\mathcal{L}(q^\alpha)| \cdot \frac{e^{-5}}{(1-\kappa^{-2})^{n^2} |\mathcal{L}(q^\alpha)|} \\
&= \frac{(1+\kappa^{-2})^{n^2}}{(1-\kappa^{-2})^{n^2} \cdot (1-\kappa^{-2})^{n^2}} \cdot e^{-5} < \frac{e}{e^{-2} \cdot e^{-2}} \cdot e^{-5} = 1,
\end{aligned}$$

where the last inequality holds, because $\kappa = \lceil \frac{nm}{\varepsilon} \rceil \geq n \geq 2$, $(1+\ell^{-1})^\ell < e$, and $(1-\ell^{-1})^\ell \geq e^{-2}$ for every $\ell \geq 2$. Hence, we know that under the assumptions stated for this proposition, on each call and, in particular, on the last call, we have that $\varphi \leq 1$.

As a second step in the proof of the proposition, we show that the algorithm outputs **fail** with probability at most $1 - e^{-9}$. Notice that this probability is only due to Step (1) in Algorithm 2. That is, the probability we output fail is at most $(1-\varphi)$, where φ is as computed in the previous part of the proof. Thus, to show that the failure probability is at most $1 - e^{-9}$, we compute a lower bound for φ in a similar way as we computed an upper bound for it:

$$\begin{aligned}
\varphi &= \left(\prod_{\beta=1}^{\alpha} \frac{N\left(P_0^\beta\right) + N\left(P_1^\beta\right)}{N\left(P_{w[\beta]}^\beta\right)}\right) \cdot \frac{e^{-5}}{N(q^\alpha)} \\
&\geq \left(\prod_{\beta=1}^{\alpha} \frac{(1-\kappa^{-2})^\beta \cdot \left(\left|\mathcal{L}\left(P_0^\beta\right)\right| + \left|\mathcal{L}\left(P_1^\beta\right)\right|\right)}{(1+\kappa^{-2})^\beta \cdot \left|\mathcal{L}\left(P_{w[\beta]}^\beta\right)\right|}\right) \cdot \frac{e^{-5}}{N(q^\alpha)} \\
&= \left(\prod_{\beta=1}^{\alpha} \frac{(1-\kappa^{-2})^\beta}{(1+\kappa^{-2})^\beta}\right) \cdot \left(\prod_{\beta=1}^{\alpha} \frac{\left(\left|\mathcal{L}\left(P_0^\beta\right)\right| + \left|\mathcal{L}\left(P_1^\beta\right)\right|\right)}{\left|\mathcal{L}\left(P_{w[\beta]}^\beta\right)\right|}\right) \cdot \frac{e^{-5}}{N(q^\alpha)} \\
&\geq \left(\frac{1-\kappa^{-2}}{1+\kappa^{-2}}\right)^{n^2} \cdot |\mathcal{L}(q^\alpha)| \cdot \frac{e^{-5}}{(1+\kappa^{-2})^{n^2} \cdot |\mathcal{L}(q^\alpha)|} \\
&= \frac{(1-\kappa^{-2})^{n^2}}{(1+\kappa^{-2})^{n^2} \cdot (1+\kappa^{-2})^{n^2}} \cdot e^{-5} \geq \frac{e^{-2}}{e \cdot e} \cdot e^{-5} = e^{-9}.
\end{aligned}$$

Note that, in the fourth step, we use the fact that $N(q^\alpha)$ is a $(1 \pm \kappa^{-2})^\alpha$-approximation of $|\mathcal{L}(q^\alpha)|$ by Proposition 6.5 (indeed, implied by the assumption that $\mathcal{E}(\beta)$ holds for each $\beta < \alpha$), so $N(q^\alpha) \leq (1+\kappa^{-2})^\alpha |\mathcal{L}(q^\alpha)| \leq (1+\kappa^{-2})^{n^2} |\mathcal{L}(q^\alpha)|$.

As the final step of the proof, we need to show that if the output of the algorithm is $w \neq$ **fail**, then $\mathbf{Pr}(w = x) = e^{-5}/N(q^\alpha)$ for every $x \in \mathcal{L}(q^\alpha)$. Now, the probability of w being a particular $x \in \mathcal{L}(q^\alpha)$ is given by the following expression:

$$\begin{aligned}
\mathbf{Pr}(w = x) &= \mathbf{Pr}(w^0 = x \wedge \text{ the last call to } \textbf{Sample} \text{ does not fail}) \\
&= \mathbf{Pr}(\text{last call to } \textbf{Sample} \text{ does not fail} \mid w^0 = x) \cdot \mathbf{Pr}(w^0 = x)
\end{aligned}$$

$$= \left(\left(\prod_{\beta=1}^{\alpha} \frac{N\left(P^{\beta}_{w[\beta]}\right)}{N\left(P^{\beta}_{0}\right) + N\left(P^{\beta}_{1}\right)} \right)^{-1} \cdot \frac{e^{-5}}{N(q^{\alpha})} \right) \cdot \left(\prod_{\beta=1}^{\alpha} \frac{N\left(P^{\beta}_{w[\beta]}\right)}{N\left(P^{\beta}_{0}\right) + N\left(P^{\beta}_{1}\right)} \right)$$

$$= \frac{e^{-5}}{N(q^{\alpha})},$$

as desired. This concludes the proof of the proposition. □

We would like to remark that, for Proposition 6.7 to be correct, we need that $\mathcal{E}(\beta)$ holds for every layer $\beta < \alpha$. Indeed, the sampling procedure uses values $N(P^{\beta}_{b})/(N(P^{\beta}_{0}) + N(P^{\beta}_{1}))$ for approximating the real probabilities $|\mathcal{L}(P^{\beta}_{b})|/|\mathcal{L}(P^{\beta})|$. For this, we need that each value $N(P^{\beta})$ is a good estimate for $|\mathcal{L}(P^{\beta})|$ and this is implied by Proposition 6.5 if $\mathcal{E}(\beta)$ holds for every layer $\beta < \alpha$. In the next section, we prove that indeed this happens with exponentially high probability.

6.5 Bounding the Probability of Breaking the Main Assumption

As it was previously discussed, the computation of the sketch composed by the estimates $N(q^{\alpha})$ and sets $S(q^{\alpha})$ is subject that conditions $\mathcal{E}(\alpha)$ hold for all layers $\alpha \leq n$. Therefore, this section is aimed to bound the probability that $\mathcal{E}(\alpha)$ is false for some layer α and show that, indeed, this probability is exponentially low.

First, assume that we are back to a layer α, condition $\mathcal{E}(\beta)$ holds for all layers $\beta < \alpha$, and we want to check the probability that $\mathcal{E}(\alpha)$ holds for α. In other words, we want to bound $\mathbf{Pr}(\mathcal{E}(\alpha) \mid \bigwedge_{\beta=0}^{\alpha-1} \mathcal{E}(\beta))$, for which we need Hoeffding's inequality.

Proposition 6.8 (Hoeffding's Inequality [20]). *Let $X_1, \ldots, X_t$ be independent random variables bounded by the interval $[0, 1]$ such that $\mathbb{E}[X_i] = \mu$. Then for every $\delta > 0$, it holds that*

$$\mathbf{Pr}\left(\left| \frac{1}{t} \sum_{i=1}^{t} X_i - \mu \right| \geq \delta \right) \leq 2e^{-2t\delta^2}.$$

For the first layer $\alpha = 0$, the condition $\mathcal{E}(0)$ certainly holds. Now, if we are at any layer α and $\bigwedge_{\beta=0}^{\alpha-1} \mathcal{E}(\beta)$ holds, then we know by Proposition 6.7 that for each $q \in Q$, it is possible to fill $S(q^{\alpha})$ with $2\kappa^7$ uniform samples of $\mathcal{L}(q^{\alpha})$. Consider the case of any subset $P \subseteq Q$, and let $S(q^{\alpha}) = \{w_1, \ldots, w_t\}$ be the uniform sample of $\mathcal{L}(q^{\alpha})$ of size $t = 2\kappa^7$. For each w_i, consider the random variable X_i such that $X_i = 1$ if $w_i \in (\mathcal{L}(q^{\alpha}) \setminus \bigcup_{p \in P} \mathcal{L}(p^{\alpha}))$, and 0 otherwise. Then, we have that:

$$\begin{aligned}
\mathbb{E}[X_i] &= \frac{|\mathcal{L}(q^{\alpha}) \setminus \bigcup_{p \in P} \mathcal{L}(p^{\alpha})|}{|\mathcal{L}(q^{\alpha})|} \\
\sum_{i=1}^{t} X_i &= |S(q^{\alpha}) \setminus \bigcup_{p \in P} \mathcal{L}(p^{\alpha})|, \quad \text{and} \\
t &= |S(q^{\alpha})|.
\end{aligned}$$

Therefore, by Hoeffding's inequality we infer that:

$$\mathbf{Pr}\left(\left| \frac{|\mathcal{L}(q^{\alpha}) \setminus \bigcup_{p \in P} \mathcal{L}(p^{\alpha})|}{|\mathcal{L}(q^{\alpha})|} - \frac{|S(q^{\alpha}) \setminus \bigcup_{p \in P} \mathcal{L}(p^{\alpha})|}{|S(q^{\alpha})|} \right| \geq \frac{1}{\kappa^3} \;\middle|\; \bigwedge_{\beta=0}^{\alpha-1} \mathcal{E}(\beta) \right) \leq 2e^{-4\kappa}.$$

Note that in the previous inequality the condition $\bigwedge_{\beta=0}^{\alpha-1} \mathcal{E}(\beta)$ does not change the assumptions of Hoeffding's inequality. We can bound $\mathbf{Pr}(\neg\mathcal{E}(\alpha) \mid \bigwedge_{\beta=0}^{\alpha-1} \mathcal{E}(\beta))$ by taking the union bound over all states q and all possible subsets $P \subseteq Q$:

$$\mathbf{Pr}\left(\exists q \in Q \;\; \exists P \subseteq Q \;\; \left| \frac{|\mathcal{L}(q^\alpha) \setminus \bigcup_{p \in P} \mathcal{L}(p^\alpha)|}{|\mathcal{L}(q^\alpha)|} - \frac{|S(q^\alpha) \setminus \bigcup_{p \in P} \mathcal{L}(p^\alpha)|}{|S(q^\alpha)|} \right| \geq \frac{1}{\kappa^3} \;\middle|\; \bigwedge_{\beta=0}^{\alpha-1} \mathcal{E}(\beta) \right)$$
$$\leq m2^m \cdot 2e^{-4\kappa} \;\leq\; e^{2nm} \cdot e^{-4\kappa} \;\leq\; e^{-2\kappa}.$$

We conclude that, at layer α, the probability $\mathbf{Pr}(\mathcal{E}(\alpha) \mid \wedge_{\beta=0}^{\alpha-1} \mathcal{E}(\beta)) \geq 1 - e^{-2\kappa}$.

To extend the previous implications over all layers, we can use that:

$$\begin{aligned} \mathbf{Pr}(\mathcal{E}(0) \wedge \cdots \wedge \mathcal{E}(n)) &= \prod_{\alpha=1}^{n} \mathbf{Pr}\left(\mathcal{E}(\alpha) \mid \bigwedge_{\beta=0}^{\alpha-1} \mathcal{E}(\beta)\right) \\ &\geq \prod_{\alpha=1}^{n} (1 - e^{-2\kappa}) \;=\; (1 - e^{-2\kappa})^n. \end{aligned}$$

Moreover, we have that:

$$\begin{aligned} (1 - e^{-2\kappa})^n &= 1 + \sum_{j=1}^{n} \binom{n}{j} (-1)^j e^{-2\kappa \cdot j} \\ &\geq 1 - \sum_{j=1}^{n} \binom{n}{j} e^{-2\kappa \cdot j} \\ &\geq 1 - e^{-2\kappa} \cdot \sum_{j=1}^{n} \binom{n}{j} \\ &\geq 1 - e^{-2\kappa} \cdot 2^n \\ &\geq 1 - e^{-2\kappa} \cdot e^{\kappa} = 1 - e^{-\kappa}. \end{aligned}$$

We conclude by stating the main purpose of this section in the following proposition:

Proposition 6.9. *The probability that $\mathcal{E}(\alpha)$ holds for all layers $\alpha \leq n$ is bounded below by $1 - e^{-\kappa}$.*

6.6 The Main Algorithm, Its Correctness, and Its Complexity

In Algorithm 3, we give all the steps of the FPRAS that has been discussed in the previous subsections. This algorithm follows the same structure of Algorithm 1, but now the computation of $N(q^\alpha)$ and $S(q^\alpha)$ is fully described. The algorithm proceeds as mentioned before, layer-by-layer, computing the estimates $N(q^\alpha)$ and the sample sets $S(q^\alpha)$. For each state q^0 at the initial layer $\alpha = 0$, the pair $N(q^\alpha), S(q^\alpha)$ is computed without considering any additional information (Step (4) of Algorithm 3). Then for each layer $\alpha > 0$ and each vertex q^α, we compute $N(q^\alpha) = N(R_0)+N(R_1)$, as was discussed in Section 6.3. After $N(q^\alpha)$ is computed, the set $S(q^\alpha)$ is filled with $2\kappa^7$ uniform and independent samples. For this, $\mathbf{Sample}(\alpha, \{q^\alpha\}, \lambda, \frac{e^{-5}}{N(q^\alpha)})$ is run at most $\Theta(\log(\kappa))$ times or until a string $w \neq$ **fail** is output. If $w =$ **fail**, then the algorithm terminates and outputs 0. Otherwise, the sample is added to $S(q^\alpha)$, and the computation continues. Finally, the algorithm computes and returns $N(F^n)$, as it was discussed in Section 6.3.

Given that the value κ is polynomial in m, n, and $\frac{1}{\varepsilon}$, it is clear that Algorithm 3 works in time polynomial in m, n, and $\frac{1}{\varepsilon}$. Hence, it only remains to show that Algorithm 3 is correct, namely, that $N(F^n)$ is a $(1 \pm \varepsilon)$-approximation of $|\mathcal{L}_n(A)|$ with probability greater than $\frac{3}{4}$. First, assume that the algorithm returns a good estimate, that is, condition $\mathcal{E}(\alpha)$ holds for all layers $\alpha \leq n$ during the run of the algorithm and w is never equal to **fail** after Step (i) of the algorithm. Then by Proposition 6.6, we obtain that $N(F^n)$ is a $(1 \pm \varepsilon)$-approximation of $|\mathcal{L}_n(A)|$, as desired.

To conclude the proof of the correctness of Algorithm 3, we need to bound the probability that the algorithm does not give a good estimate, namely, that either condition $\mathcal{E}(\alpha)$ is false for

ALGORITHM 3: FPRAS to estimate $|\mathcal{L}_n(A)|$ for an NFA $A = (Q, \{0,1\}, \Delta, I, F)$ with $m \geq 2$ states, integer $n \geq 2$ given in unary and error $\varepsilon \in (0,1)$

(1) If $\mathcal{L}_n(A) = \varnothing$, then return 0.
(2) Else, construct the directed acyclic graph A_{unroll} from A, and set $\kappa = \lceil \frac{nm}{\varepsilon} \rceil$.
(3) For each vertex q^α of A_{unroll}, if there is no path from a vertex in I^0 to q^α, then remove q^α from A_{unroll}.
(4) For each $q^0 \in I^0$, set $N(q^0) = 1$ and $S(q^0) = \{\lambda\}$.
(5) For layers $\alpha = 1, 2, \ldots, n$ and for each vertex q^α in A_{unroll}:
  (a) Let $R_b = \{p^{\alpha-1} \in Q^{\alpha-1} \mid (p^{\alpha-1}, b, q^\alpha) \text{ is an edge in } A_{unroll}\}$ for $b = 0, 1$.
  (b) Set $N(q^\alpha) = N(R_0) + N(R_1)$.
  (c) Set $S(q^\alpha) = \varnothing$. Then while $|S(q^\alpha)| < 2\kappa^7$:
    (i) Run **Sample**$(\alpha, \{q^\alpha\}, \lambda, \frac{e^{-5}}{N(q^\alpha)})$ until it returns a string $w \neq$ **fail**, and at most $\Theta(\log(\kappa))$ times
    (ii) If $w =$ **fail**, then terminate the algorithm and output 0 as the estimate (failure event).
    (iii) Otherwise, a sample $w \in \{0,1\}^\alpha$ was returned, and set $S(q^\alpha) = S(q^\alpha) \cup \{w\}$ (recall $S(q^\alpha)$ allows duplicates).
(6) Return $N(F^n)$ as an estimate for $|\mathcal{L}_n(A)|$.

some layer α or the sampling algorithm fails $c(\kappa)$ times at Step (i), where $c(\kappa)$ is the number of repetitions performed in this step. Therefore, it remains to show that this probability of giving a wrong output is at most $\frac{1}{4}$ considering a value for $c(\kappa) \in \Theta(\log(\kappa))$. Let $\mathcal{E}_{\mathbf{fail}}(\alpha, q, j)$ be the event that the call **Sample**$(\alpha, \{q^\alpha\}, \lambda, e^{-5}/N(q^\alpha))$ fails $c(\kappa)$ consecutive times at layer α, state q^α, and the jth sample of $S(q^\alpha)$, where $j \in [1, 2\kappa^7]$. We know by Proposition 6.7 that the probability that **Sample** fails is at most $1 - e^{-9}$ and, therefore, $\mathbf{Pr}(\mathcal{E}_{\mathbf{fail}}(\alpha, q, j)) \leq (1 - e^{-9})^{c(\kappa)}$. Furthermore, by Proposition 6.9, we already know that $\mathbf{Pr}(\neg\mathcal{E}(0) \vee \cdots \vee \neg\mathcal{E}(n)) \leq e^{-\kappa}$. By the union bound, we conclude that the probability that the algorithm gives the wrong output is:

$$\begin{aligned} \mathbf{Pr}\left(\neg \bigvee_{\alpha=1}^{n} \bigvee_{q \in Q} \bigvee_{j=1}^{2\kappa^7} \mathcal{E}_{\mathbf{fail}}(\alpha, q, j) \vee \neg\mathcal{E}(0) \vee \cdots \vee \neg\mathcal{E}(n)\right) &\leq \sum_{\alpha=1}^{n} \sum_{q \in Q} \sum_{j=1}^{2\kappa^7} (1 - e^{-9})^{c(\kappa)} + e^{-\kappa} \\ &\leq 2nm\kappa^7 (1 - e^{-9})^{c(\kappa)} + e^{-2} \\ &\leq 2\kappa^8 (1 - e^{-9})^{c(\kappa)} + e^{-2}. \end{aligned}$$

Finally, if we take

$$c(\kappa) = \left\lceil \frac{2 + \log(4) + 8\log(\kappa)}{\log((1 - e^{-9})^{-1})} \right\rceil,$$

then we obtain that $2\kappa^8 (1 - e^{-9})^{c(\kappa)} \leq \frac{1}{2} \cdot e^{-2}$. Therefore, we have that the probability that the algorithm returns a wrong estimate is at most $\frac{3}{2} \cdot e^{-2} < \frac{1}{4}$. As $c(\kappa)$ is $\Theta(\log(\kappa))$, this concludes the proof of the correctness of Algorithm 3 as stated in Theorem 6.2.

To finish this section, we need to prove Theorem 6.3, that is, we need to show that GEN(MEM-NFA) admits a **preprocessing polynomial-time Las Vegas uniform generator (PPLVUG)**. Notice that our algorithm results in a sampler satisfying the conditions of Theorem 6.3. Specifically, given an NFA A and a natural number n, Algorithm 3 builds a structure $(A_{unroll}, \{N(p^\alpha), S(p^\alpha)\}_{p \in Q, \alpha \leq n})$. Furthermore, conditioned on all the above events (i.e.,

$\mathcal{E}(0), \ldots, \mathcal{E}(n)$, and $\mathcal{E}_{\mathbf{fail}}$) we can use $(A_{unroll}, \{N(p^\alpha), S(p^\alpha)\}_{p \in Q, \alpha \leq n})$ for getting a uniform sample from $\mathcal{L}_n(A)$ by using Algorithm 2, and this algorithm fails with probability $1 - e^{-9}$. In other words, Algorithm 3 and Algorithm 2 are the randomized algorithms $\mathcal{P}$ and $\mathcal{G}$, respectively, and $(A_{unroll}, \{N(p^\alpha), S(p^\alpha)\}_{p \in Q, \alpha \leq n})$ is the string $\mathcal{D}$ that is good-for-generation (i.e., the advice) from the PPLVUG's definition. The probability of success of the above events can be amplified to $1 - \delta$ for any $\delta > 0$ by scaling κ up by a factor of $\log(1/\delta)$. The overall runtime is now polynomial in $(n, m, \log(1/\delta))$ as needed. Note that for the purposes of a uniform sampler, the parameter δ can be set to a constant, as it does not appear in the definition of the sampler in Theorem 6.3. Now by Proposition 6.7, we can obtain truly uniform samples from the set $\mathcal{L}_n(A)$ in time polynomial in $(n, m, \log(1/\delta))$ time. The probability that **fail** is returned by the sampler is at most $1 - e^{-9}$, which can be amplified to at most $1/2$ by repeating it a constant number of times to satisfy the condition required in Theorem 6.3, which completes the proof of this theorem.

7 CONCLUDING REMARKS

We consider this work as a first step towards the definition of classes of problems with good properties in terms of enumeration, counting, and uniform generation of solutions. In this sense, there is plenty of room for extensions and improvements, and many problems need to be studied further. First, for each one of the classes RelationNL and RelationUL, we have identified a single problem that is complete for it. An important question, then, is whether such classes admit other natural and well-studied complete problems; for instance, we leave as an open problem whether SAT-DNF is complete for RelationNL under the notion of reduction introduced in Section 5. Second, the different components of the FPRAS for #NFA were designed to facilitate its proof of correctness. As such, we already know of some optimizations that significantly reduce its runtime, and we plan to develop more such optimizations to make this FPRAS usable in practice. Finally, an interesting area to explore is to extend the results of this article to the class of context-free languages. In particular, it will be interesting to understand if relations based on context-free languages have good properties regarding enumeration, counting, and uniform generation. Here, it is natural to ask whether the problem #CFG (i.e., to count the number of words of a given length accepted by a context-free grammar) admits an FPRAS or not.

APPENDIX

A PROOFS OF INTERMEDIATE RESULTS

A.1 Proof of Lemma 5.3

Let x be any element in $\{0,1\}^*$. Since R is in RelationUL, we know there exists a UL-transducer M such that $W_R(x) = M(x)$. Without loss of generality, we can assume that M has only one accepting state, so it can be written as a tuple $M = (Q, \Gamma, \mathsf{B}, \{0,1\}, \delta, q_0, \{q_F\})$. If it has more than one accepting state, say, a set F of accepting states, then we can define a new transducer M' that is identical to M with one difference. It has only one final state q_F and whenever it reaches a state in F, it makes one last transition to q_F and stops. It is clear that $M(x) = M'(x)$, so we do not lose any generality with this assumption.

Let $n = |x|$, $f(n)$ be the function that bounds the number of cells in the work tape that can be used, and assume that $f(n)$ is $O(\log(n))$. Consider now an execution of M on input x. Since the input tape never changes (its content is always x), we can completely characterize the configuration of the machine at any given moment as a tuple $(q, i, j, w) \in Q \times \{1, \ldots, n\} \times \{1, \ldots, f(n)\} \times \Gamma^{f(n)}$ where

- q stores the state the machine is in.
- i indicates the position of the head on the input tape.

- j indicates the position of the head on the work tape.
- w stores the contents of the work tape.

With the previous notation, the initial configuration of M on input x is represented by $c_I = (q_0, 1, 1, \mathsf{B}^{f(n)})$, that is, M is in its initial state, the heads are at the first position of their respective tapes, and the work tape is empty (that is, it only contains the blank symbol B). The accepting configuration is represented by a tuple of the form $c_F = (q_F, i_F, j_F, w_F)$. Notice that without loss of generality, we can assume the accepting configuration to be unique by changing M so it runs for a little longer to reach it. If C_x is the set of possible configuration tuples, then we have that

$$\begin{aligned}|C_x| &\leq |Q| \cdot n \cdot f(n) \cdot |\Gamma|^{f(n)} \\ &= |Q| \cdot n \cdot f(n) \cdot |\Gamma|^{O(\log(n))} \\ &= |Q| \cdot n \cdot f(n) \cdot O(n^{\ell}), \text{ where } \ell \text{ is a constant} \\ &= O(n^{\ell+1} \log(n)),\end{aligned}$$

which is polynomial in $|x|$. Recall the notation for NFAs introduced in Section 6.1. We now define the NFA $A_x = (C_x, \{0, 1\}, \Delta_x, c_I, \{c_F\})$ where C_x, c_I and c_F are defined as above and the transition relation Δ_x is constructed in the following way:

- Let $c, d \in C_x$. Consider any possible run of M on input x. Suppose there is a valid transition, during that run, that goes from c to d while outputting symbol $\gamma \in \Gamma$. Then, (c, γ, d) is in Δ_x.
- Let $c, d \in C_x$. Consider any possible run of M on input x. Suppose there is a valid transition, during that run, that goes from c to d while making no output. Then, (c, ε, d) is in Δ_x.

We already showed that C_x has polynomial size in $|x|$, and it clearly can be constructed explicitly in polynomial time. The same is true for Δ_x. Given a pair of configurations $c, d \in C_x$, it can be checked in polynomial time whether there is a possible transition from c to d during an execution of M on input x (it suffices to check δ, the transition relation for M). And there are just $|C_x|^2$ such pairs of configurations that we need to check, so the whole construction of A_x can be done in polynomial time. It only rests to show that $W_R(x) = \mathcal{L}(A_x)$ and that A_x is unambiguous.

Let $y \in W_R(x)$. That means there is an accepting run of M on input x that yields y as output. Equivalently, there is a sequence of configurations $\{c_k\}_{k=0}^m$ and a sequence $\{w_k\}_{k=0}^m$ of symbols such that:

- $c_0 = c_I$.
- $c_m = c_F$.
- For each $k \in \{0, \ldots, m-1\}$, the transition from c_k to c_{k+1} is valid on input x given the transition relation δ of M.
- For each $k \in \{0, \ldots, m-1\}$, we have that w_k is equal to the symbol output when going from configuration c_k to c_{k+1} if a symbol was output. Otherwise, $w_k = \varepsilon$.
- $y = w_0 \cdot w_1 \cdot \ldots \cdot w_m$.

By definition, this means that y is accepted by A_x. That is, $y \in \mathcal{L}(A_x)$ and so we can conclude that $W_R(x) \subseteq \mathcal{L}(A_x)$. Since all the previous implications are clearly equivalences, we can also conclude that $\mathcal{L}(A_x) \subseteq W_R(x)$. Hence, $W_R(x) = \mathcal{L}(A_x)$ as needed. What the previous argument is saying is that every accepting run of M that outputs a string y has a unique corresponding accepting run of A_x on input y. That implies that A_x is unambiguous. Otherwise, there would be some $y \in \mathcal{L}(A_x)$ such that two different runs of A_x accept y. But that would mean that there are two different runs of M on input x that output y, which cannot occur, since M is a UL-transducer.

Finally, notice that A_x is actually not an NFA (under the definition given in Section 3), since we explicitly allowed for the possibility of ε-transitions. But recall that the ε-transitions of any NFA

can be removed in polynomial time without changing the accepted language, which is a standard result from automata theory. This concludes the proof of the lemma.

A.2 Proof of Proposition 5.4

We focus on the case of MEM-NFA (it extends easily to MEM-UFA). To show this result, we need to include a little more detail in our definition of MEM-NFA to consider some corner cases. First, we have to consider the cases where the string in unary is empty. That is, the case where $k = 0$ in input $(N, 0^k)$. This just amounts to the following: If the starting state is a final state, then we consider that the automaton does accept the empty string. So, if $k = 0$ and N is an NFA that has all the properties stated in the definition of MEM-NFA, plus its starting state is the accepting state, then $((N, 0^k), \varepsilon) \in$ MEM-NFA. Also, we need to consider the cases where N does not have all the properties stated in the definition of MEM-NFA. In those cases, we consider that $(N, 0^k)$, for any k, does not have any solutions. Also—and this gets more technical—we consider that any input that has an invalid encoding does not have any solutions either. We will not be completely precise about which encoding should be used (although during the proof we will mention some important points regarding that). But we will ask that the correctness of the encoding can be checked in polynomial time (this is a mild requirement, as any reasonable encoding will allow for it). And it is important to have in mind that for some technical concepts like self-reducibility, the encoding of the problem is critical.

We use the notion of self-reducibility stated in Reference [30], adapted to our situation, since Reference [30] uses a slightly different framework to define an enumeration problem. We say a relation $R \subseteq \{0,1\}^* \times \{0,1\}^*$ is self reducible if there exist polynomial-time computable functions $\psi : \{0,1\}^* \times \{0,1\}^* \to \{0,1\}^*$, $\sigma : \{0,1\}^* \to \mathbb{N}$ and $\ell : \{0,1\}^* \to \mathbb{N}$ such that for every $x, y, w \in \{0,1\}^*$:

(1) if $(x, y) \in R$, then $|y| = \ell(x)$,

(2) if $\ell(x) = 0$, then it can be tested in polynomial time in $|x|$, whether the empty string is a solution for x.

(3) $\sigma(x) \in O(\log |x|)$,

(4) $\ell(x) > 0$ if and only if $\sigma(x) > 0$,

(5) $|\psi(x, w)| \leq |x|$,

(6) $\ell(\psi(x, w)) = \max\{\ell(x) - |w|, 0\}$, and

(7) $W_R(x) = \bigcup_{w \in \{0,1\}^{\sigma(x)}} \{w \cdot y \mid y \in W_R(\psi(x, w))\}$.

The last condition intuitively says that all solutions for a given input can be constructed from the solutions of (a polynomial number of) smaller instances. It can be equivalently stated in the following way, which is how we will use it:

(8) if $y = y_1 y_2 \ldots y_m$, then it holds that $(x, y) \in R$ if and only if $(\psi(x, y_1 \ldots y_{\sigma(x)}), y_{\sigma(x)+1} \ldots y_m) \in R$.

As we already stated, the empty string is a solution only when the input is correctly encoded and the initial and final states of the automaton coincide. So, condition (2) from above is satisfied regardless of our definition of ℓ. We will focus from now on on the other six conditions. Let $\mathcal{N} = \{N \mid N$ is an NFA with a unique final state and no ε-transitions$\}$. Following the previous notation, we define the functions ℓ, σ, and ψ that characterize self-reducibility. The only interesting cases, of course, are those where the automaton in the input is in $\mathcal{N}$ (and the input is correctly encoded).

In all the others, the input is not correct, so the set of solutions is empty, and we do not need to worry about self-reducibility. That said, we define

$$\ell((N, 0^k)) = \begin{cases} k & \text{if the input is correctly encoded and } N \in \mathcal{N} \\ 0 & \text{in any other case;} \end{cases}$$

$$\sigma((N, 0^k)) = \begin{cases} 1 & \text{if the input is correctly encoded, } k > 0 \text{ and } N \in \mathcal{N} \\ 0 & \text{in any other case.} \end{cases}$$

Both functions are clearly computable in polynomial time. The definition of ℓ is just saying that on input $(N, 0^k)$, any solution will have length k, which comes directly from the definition of MEM-NFA. The definition of σ indicates that, for any input, as long as its solutions have positive length, we can create another input that has the same solutions, but with the first character removed. Notice that with these definitions, conditions (3) and (4) for self-reducibility are trivially met. Condition (1) is also met, which is easy to see from the definitions of MEM-NFA and ℓ. The only task left is to define ψ and prove conditions (5), (6), and (8). We now proceed in that direction.

Let $N = (Q, \{0, 1\}, \delta, q_0, \{q_F\})$ be an automaton in $\mathcal{N}$. Notice we are making the assumption that N has a unique final state, since it makes the idea clearer and the proof only has to be modified slightly for the general case. We will mention some points about the exact encoding soon (which is key for condition (5) to hold). But first, consider an input $x = (N, 0^k)$ that is incorrectly encoded or where N is not in $\mathcal{N}$. Then, it has no solutions and it is enough to set $\psi(x, w) = x$ for all $w \in \{0, 1\}^*$ (which is clearly computable in polynomial time). In that case, notice that condition (5) is trivially true. Also, notice that, since N is not in $\mathcal{N}$ (or is encoded in an incorrect format), we have $\ell(x) = \sigma(x) = 0$, so for any w it holds that

$$\ell(\psi(x, w)) = \ell(x) = 0 = \max\{-|w|, 0\} = \max\{\ell(x) - |w|, 0\},$$

so condition (6) is also true. And given that $\ell(x) = \sigma(x) = 0$, condition (8) amounts to checking that for every $y \in \{0, 1\}^*$, it holds that $(x, y) \in$ MEM-NFA if and only if $(x, y) \in$ MEM-NFA, which is obviously true. Now, consider the case of an input $x = (N, 0^k)$ that is correctly encoded and where N is in $\mathcal{N}$. There are two main cases to consider.

First, the case where $k = 0$. This case is also simple, because we can set $\psi(x, w) = x$ for all $w \in \{0, 1\}^*$ (which is computable in polynomial time and implies that condition (5) is trivially true), and, since $\ell(x) = \sigma(x) = 0$, it is possible to prove as before that conditions (6) and (8) hold. Second, we need to consider the case where $k > 0$. Then, we have $\sigma(x) = 1$, so $\psi(x, w)$ only needs to be defined when w is a single symbol. Then, for both $w \in \{0, 1\}$, we set $\psi((N, 0^k), w) = (N', 0^{k-1})$, where N' is defined as follows: Let Q_w be the set

$$Q_w = \{q \in Q \mid (q_0, w, q) \in \delta\}.$$

Thus, Q_w is the set of states that can be reached (with one transition) from the initial state by reading the symbol w. Now, we define $N' = (Q', \{0, 1\}, \delta', q_0', \{q_F'\})$ where q_0' is a new state not contained in Q, and:

$$\begin{aligned} Q' &= (Q \setminus Q_w) \cup \{q_0'\}, \\ \delta' &= \{(q, a, p) \mid (q, a, p) \in \delta \text{ and } q, p \in Q'\} \cup \{(q, a, q_0') \mid (q, a, p) \in \delta \text{ and } q \in Q', p \in Q_w\} \cup \\ &\quad \{(q_0', a, p) \mid (q, a, p) \in \delta \text{ and } q \in Q_w, p \in Q'\} \cup \{(q_0', a, q_0') \mid (q, a, p) \in \delta \text{ and } q, p \in Q_w\}, \\ q_F' &= \begin{cases} q_F & \text{if } q_F \in Q' \\ q_0' & \text{if } q_F \notin Q'. \end{cases} \end{aligned}$$

Notice that this construction takes only polynomial time. What we are doing, basically, is the following: Imagine Q_w as a first "layer" of states reachable from q_0 in one step. We want to merge all of Q_w in a single new initial state q_0', while ensuring that from q_0' we can reach the same states as were previously reachable from Q_w. The definitions are a little complicated, because we have to account for some special cases. For example, we would maybe want to remove q_0 (since now we have a new initial state) but there is the possibility that q_0 is part of the acceptance runs of some strings, and not only as an initial state. The same goes for the states in Q_w, and that is why we have many different cases to consider in the definition of δ'. We have to make sure not to lose any accepting runs with the removal of Q_w.

Now, we make some observations about N'. To construct Q', we are removing at least one state from Q, but we are adding at most one new state, q_0'. This implies that $|Q'| \leq |Q|$ (notation here indicates set cardinality). Similarly for the construction of δ'. Notice that each transition we add to construct δ' (besides the ones that come directly from δ) corresponds to a transition that already existed and that involved at least one state from Q_w. So, all in all, we have not really added any new transitions, just simulated the ones where states in Q_w appeared. That means that $|\delta'| \leq |\delta|$. So, as a whole, N' contains at most as many states and transitions as N, and maybe less. Does that mean that (notation here indicates encoding sizes) $|\psi((N, 0^k), w)| \leq |(N, 0^k)|$? It will depend on the type of encoding used, of course. So, we will consider that the NFA in the input is encoded in the following (natural) way: First, a list of all states, followed by the list of all tuples in the transition relation, and at the end the initial and final states. Also, we assume that all states have an encoding of the same size (which is easy to achieve through padding). And the same goes for all transitions. With that encoding, since N' has less (or equal) number of states and transitions than N, it is clear that $|N'| \leq |N|$. Of course, it is also true that $|0^{k-1}| \leq |0^k|$. We can then conclude that $|\psi((N, 0^k), w)| = |(N', 0^{k-1})| \leq |(N, 0^k)|$, that is, condition (5) is satisfied. We also have by definition of ℓ that $\ell((N, 0^k)) = k$ and $\ell((N', 0^{k-1})) = k - 1$. Since $|w| = 1$, condition (6) is also true:

$$\begin{aligned}\ell(\psi((N, 0^k), w)) &= \ell((N', 0^{k-1})) = k - 1 \\ &= \ell((N, 0^k)) - 1 = \ell((N, 0^k)) - |w| = \max\{\ell((N, 0^k)) - |w|, 0\}.\end{aligned}$$

Finally, we turn to condition (8). Let $y = y_1 y_2 \ldots y_m \in \{0, 1\}^*$. Since $\sigma(x) = 1$, condition (8) amounts to checking that

$$((N, 0^k), y) \in \text{MEM-NFA} \text{ if and only if } ((N', 0^{k-1}), y_2 \ldots y_m) \in \text{MEM-NFA},$$

where N' is constructed by considering $w = y_1$, that is, $N' = \psi((N, 0^k), y_1)$. Notice that if $m \neq k$, then both sides of the equivalence above are immediately false (and thus the equivalence is true), so we need only consider the case where $m = k$. We will now prove both directions of the equivalence. First, suppose $((N, 0^k), y) \in$ MEM-NFA. Then, by definition, we know there is an accepting run ρ of N on input y such that

$$\rho : p_0 \xrightarrow{y_1} p_1 \xrightarrow{y_2} p_2 \xrightarrow{y_3} \cdots \xrightarrow{y_{k-1}} p_{k-1} \xrightarrow{y_k} p_k,$$

where $p_0 = q_0$, $p_k = q_F$ and $(p_{i-1}, y_i, p_i) \in \delta$ for all $i \in \{1, \ldots, k\}$. Now, we will show that $((N', 0^{k-1}), y_2 \ldots y_k) \in$ MEM-NFA, that is, $y_2 \ldots y_k$ is accepted by N'. To do that, we first show by induction the following property: For all $i \in \{2, \ldots, k\}$ there is a valid run of N' on input $y_2 \ldots y_i$ (although the run is not necessarily accepting) that looks like this:

$$\rho_i : s_1 \xrightarrow{y_2} s_2 \xrightarrow{y_3} s_3 \xrightarrow{y_4} \cdots \xrightarrow{y_{i-1}} s_{i-1} \xrightarrow{y_i} s_i,$$

where $s_1 = q'_0$ and for all $j \in \{2, \dots, i\}$, we have that $(s_{j-1}, y_j, s_j) \in \delta'$ and

$$s_j = \begin{cases} p_j & \text{if } p_j \notin Q_{y_1} \\ q'_0 & \text{if } p_j \in Q_{y_1}. \end{cases}$$

To prove this fact by induction, consider first the case of $i = 2$. By definition, we know that $p_1 \in Q_{y_1}$ and $(p_1, y_1, p_2) \in \delta$. There are now two different possibilities. First, if $p_2 \notin Q_{y_1}$, then by definition of δ', we know that $(q'_0, y_2, p_2) \in \delta'$. Second, if $p_2 \in Q_{y_1}$, then by definition of δ', we know that $(q'_0, y_2, q'_0) \in \delta'$. So, the property is true when $i = 2$.

Now, suppose the property holds for some $i < k$, and consider the case for $i + 1$. By the induction hypothesis, we know there is a valid run ρ_i such that

$$\rho_i : s_1 \xrightarrow{y_2} s_2 \xrightarrow{y_3} s_3 \xrightarrow{y_4} \cdots \xrightarrow{y_{i-1}} s_{i-1} \xrightarrow{y_i} s_i,$$

where $s_1 = q'_0$ and $(s_{j-1}, y_j, s_j) \in \delta'$ for all $j \in \{2, \dots, i\}$. Now, by the induction hypothesis, there are four possibilities (where each possibility is represented in one of the four sets that form the definition of δ'):

- $s_i = p_i$ and $p_{i+1} \notin Q_{y_1}$. In that case, if we set $s_{i+1} = p_{i+1}$, then, by definition, we know that $(s_i, y_{i+1}, s_{i+1}) \in \delta'$.
- $s_i = p_i$ and $p_{i+1} \in Q_{y_1}$. In that case, if we set $s_{i+1} = q'_0$, then, by definition, we know that $(s_i, y_{i+1}, s_{i+1}) \in \delta'$.
- $s_i = q'_0$ and $p_{i+1} \notin Q_{y_1}$. In that case, if we set $s_{i+1} = p_{i+1}$, then, by definition, we know that $(s_i, y_{i+1}, s_{i+1}) \in \delta'$.
- $s_i = q'_0$ and $p_{i+1} \in Q_{y_1}$. In that case, if we set $s_{i+1} = q'_0$, then, by definition, we know that $(s_i, y_{i+1}, s_{i+1}) \in \delta'$.

All that means that we can add one more transition to ρ_i to form a valid run ρ_{i+1} given by

$$\rho_{i+1} : s_1 \xrightarrow{y_2} s_2 \xrightarrow{y_3} s_3 \xrightarrow{y_4} \cdots \xrightarrow{y_i} s_i \xrightarrow{y_{i+1}} s_{i+1},$$

where $s_1 = q'_0$ and for all $j \in \{2, \dots, i+1\}$, we have that $(s_{j-1}, y_j, s_j) \in \delta'$ and

$$s_j = \begin{cases} p_j & \text{if } p_j \notin Q_{y_1} \\ q'_0 & \text{if } p_j \in Q_{y_1}. \end{cases}$$

The property is thus proved. Now, consider a valid run of that type for $i = k$ that looks like

$$\rho' : s_1 \xrightarrow{y_2} s_2 \xrightarrow{y_3} s_3 \xrightarrow{y_4} \cdots \xrightarrow{y_{k-1}} s_{k-1} \xrightarrow{y_k} s_k,$$

where $s_1 = q'_0$ and for all $j \in \{2, \dots, k\}$, we have that $(s_{j-1}, y_j, s_j) \in \delta'$. Now, by the property just proved, we know there are two possibilities. First, if $p_k \notin Q_{y_1}$, then we know $s_k = p_k = q_F$. Since $q_F = p_k \notin Q_{y_1}$, we have that $q'_F = q_F$ and thus ρ' is an accepting run, which means that $y_2 \dots y_k$ is accepted by N'. Second, if $p_k \in Q_{y_1}$, then we know $s_k = q'_0$. And, since $q_F = p_k \in Q_{y_1}$, we have that $q'_F = q'_0$ and thus ρ' is again an accepting run, which means that $y_2 \dots y_k$ is accepted by N'. All this proves that if $((N, 0^k), y) \in$ MEM-NFA, then $((N', 0^{k-1}), y_2 \dots y_k) \in$ MEM-NFA. The proof for the other direction is analogous.

We conclude this section by pointing out that the same proof works for the case of MEM-UFA. That is, the same definitions of ℓ, σ, and ψ work for that proof. The only difference is that we also need to show that ψ produces a valid automaton for the relation, that is, an unambiguous NFA. But that is not hard to show from the previous proof. Making similar use of the notation of valid runs, it can be shown that if $\psi((N, 0^k), w)$ had two different accepting runs for some word y, then N would have two different accepting runs for $w \circ y$, and so it would not be unambiguous.

ACKNOWLEDGMENTS

The authors would like to thank the anonymous reviewers for their many helpful comments.

REFERENCES

[1] Serge Abiteboul, Gerome Miklau, Julia Stoyanovich, and Gerhard Weikum. 2016. Data, responsibly (Dagstuhl seminar 16291). *Dagstuhl Rep.* 6, 7 (2016), 42–71.
[2] Alfred V. Aho and John E. Hopcroft. 1974. *The Design and Analysis of Computer Algorithms.* Pearson Education India.
[3] Carme Álvarez and Birgit Jenner. 1993. A very hard log-space counting class. *Theor. Comput. Sci.* 107, 1 (1993), 3–30.
[4] Antoine Amarilli, Pierre Bourhis, Louis Jachiet, and Stefan Mengel. 2017. A circuit-based approach to efficient enumeration. In *Proceedings of ICALP.* 111:1–111:15.
[5] Antoine Amarilli, Florent Capelli, Mikaël Monet, and Pierre Senellart. 2018. Connecting knowledge compilation classes and width parameters. *CoRR* abs/1811.02944 (2018).
[6] Renzo Angles, Marcelo Arenas, Pablo Barceló, Aidan Hogan, Juan Reutter, and Domagoj Vrgoč. 2017. Foundations of modern query languages for graph databases. *ACM Comput. Surv.* 50, 5 (2017), 68.
[7] Marcelo Arenas, Sebastián Conca, and Jorge Pérez. 2012. Counting beyond a Yottabyte, or how SPARQL 1.1 property paths will prevent adoption of the standard. In *Proceedings of WWW.* 629–638.
[8] Marcelo Arenas, Martin Muñoz, and Cristian Riveros. 2017. Descriptive complexity for counting complexity classes. In *Proceedings of LICS.* 1–12.
[9] Guillaume Bagan. 2006. MSO queries on tree decomposable structures are computable with linear delay. In *Proceedings of CSL.* 167–181.
[10] Guillaume Bagan, Arnaud Durand, and Etienne Grandjean. 2007. On acyclic conjunctive queries and constant delay enumeration. In *Proceedings of CSL.* 208–222.
[11] Randal E. Bryant. 1992. Symbolic Boolean manipulation with ordered binary-decision diagrams. *ACM Comput. Surv.* 24, 3 (1992), 293–318.
[12] Bruno Courcelle. 2009. Linear delay enumeration and monadic second-order logic. *Discr. Appl. Math.* 157, 12 (2009), 2675–2700.
[13] Constantinos Daskalakis, Paul W. Goldberg, and Christos H. Papadimitriou. 2009. The complexity of computing a Nash equilibrium. *SIAM J. Comput.* 39, 1 (2009), 195–259.
[14] Ronald Fagin, Benny Kimelfeld, Frederick Reiss, and Stijn Vansummeren. 2015. Document spanners: A formal approach to information extraction. *J. ACM* 62, 2 (2015), 12.
[15] Fernando Florenzano, Cristian Riveros, Martin Ugarte, Stijn Vansummeren, and Domagoj Vrgoc. 2018. Constant delay algorithms for regular document spanners. *arXiv preprint arXiv:1803.05277* (2018).
[16] Dominik D. Freydenberger. 2017. A logic for document spanners. In *Proceedings of ICDT.* 13:1–13:18.
[17] Dominik D. Freydenberger, Benny Kimelfeld, and Liat Peterfreund. 2018. Joining extractions of regular expressions. In *Proceedings of PODS.* 137–149.
[18] Vivek Gore, Mark Jerrum, Sampath Kannan, Z. Sweedyk, and Stephen R. Mahaney. 1997. A quasi-polynomial-time algorithm for sampling words from a context-free language. *Inf. Comput.* 134, 1 (1997), 59–74.
[19] Lane A. Hemaspaandra and Heribert Vollmer. 1995. The satanic notations: Counting classes beyond #P and other definitional adventures. *SIGACT News* 26, 1 (1995), 2–13.
[20] Wassily Hoeffding. 1963. Probability inequalities for sums of bounded random variables. *J. Amer. Statist. Assoc.* 58, 301 (1963), 13–30.
[21] Mark R. Jerrum, Leslie G. Valiant, and Vijay V. Vazirani. 1986. Random generation of combinatorial structures from a uniform distribution. *Theor. Comput. Sci.* 43 (1986), 169–188.
[22] David S. Johnson, Mihalis Yannakakis, and Christos H. Papadimitriou. 1988. On generating all maximal independent sets. *Inform. Process. Lett.* 27, 3 (1988), 119–123.
[23] Sampath Kannan, Z. Sweedyk, and Stephen R. Mahaney. 1995. Counting and random generation of strings in regular languages. In *Proceedings of SODA.* 551–557.
[24] Richard M. Karp and Michael Luby. 1983. Monte Carlo algorithms for enumeration and reliability problems. In *Proceedings of FOCS.* 56–64.
[25] Katja Losemann and Wim Martens. 2013. The complexity of regular expressions and property paths in SPARQL. *ACM Trans. Datab. Syst.* 38, 4 (2013), 24:1–24:39.
[26] Francisco Maturana, Cristian Riveros, and Domagoj Vrgoc. 2018. Document spanners for extracting incomplete information: Expressiveness and complexity. In *Proceedings of PODS.* ACM, 125–136.
[27] J. Scott Provan and Michael O. Ball. 1983. The complexity of counting cuts and of computing the probability that a graph is connected. *SIAM J. Comput.* 12, 4 (1983), 777–788.
[28] Klaus Reinhardt and Eric Allender. 2000. Making nondeterminism unambiguous. *SIAM J. Comput.* 29, 4 (2000), 1118–1131.

[29] Sanjeev Saluja, K. V. Subrahmanyam, and Madhukar N. Thakur. 1995. Descriptive complexity of# P functions. *J. Comput. Syst. Sci.* 50, 3 (1995), 493–505.
[30] Johannes Schmidt. 2009. Enumeration: Algorithms and complexity. Retrieved from https://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.582.8008&rep=rep1&type=pdf.
[31] Luc Segoufin. 2013. Enumerating with constant delay the answers to a query. In *Proceedings of ICDT*. 10–20.
[32] Leslie G. Valiant. 1976. Relative complexity of checking and evaluating. *Inf. Process. Lett.* 5, 1 (1976), 20–23.
[33] Moshe Y. Vardi. 1982. The complexity of relational query languages (extended abstract). In *Proceedings of the 14th Annual ACM Symposium on Theory of Computing*. 137–146.

Received April 2020; revised April 2021; accepted July 2021