Efficient Logspace Classes for Enumeration, Counting, and Uniform Generation

Marcelo Arenas
PUC & IMFD Chile
marenas@ing.puc.cl

Luis Alberto Croquevielle
PUC & IMFD Chile
lacroquevielle@uc.cl

Rajesh Jayaram
Carnegie Mellon University
rkjayara@cs.cmu.edu

Cristian Riveros
PUC & IMFD Chile
cristian.riveros@uc.cl

ABSTRACT

In this work, we study two simple yet general complexity classes, based on logspace Turing machines, which provide a unifying framework for efficient query evaluation in areas like 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 nondeterministic 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 for uniform generation. Interestingly, 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* accepted by a nondeterministic 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.*

KEYWORDS

Enumeration, counting, uniform generation.

ACM Reference Format:
Marcelo Arenas, Luis Alberto Croquevielle, Rajesh Jayaram, and Cristian Riveros. 2019. Efficient Logspace Classes for Enumeration, Counting, and Uniform Generation. In *38th ACM SIGMOD-SIGACT-SIGAI Symposium on Principles of Database Systems (PODS'19), June 30-July 5, 2019, Amsterdam, Netherlands.* ACM, New York, NY, USA, 15 pages. https://doi.org/10.1145/3294052.3319704

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 in the area. 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 solutions to a query could also be enormous, so computing the complete set of solutions can be prohibitively expensive. In order 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 [11, 16, 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 the variety. Most notably, 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 to the problem of answering a query.

In this paper, we follow a more 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 [22], 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 other 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. Thus, if Σ denotes a finite alphabet, then by following [22] we represent a problem as a relation $R \subseteq \Sigma^* \times \Sigma^*$, and we say that y is a solution for an input x if $(x, y) \in R$. Note that the problem of enumerating the solutions to a given input x corresponds to the problem of enumerating the elements of the set $\{y \in \Sigma^* \mid (x, y) \in R\}$, while the counting and uniform generation problems correspond to the problems of computing the cardinality of $\{y \in \Sigma^* \mid (x, y) \in R\}$ 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 for studying enumeration, counting and uniform generation. More specifically, given a finite alphabet Σ, an NL-transducer M is a nondeterministic Turing Machine with input and output alphabet Σ, a read-only input tape, a write-only output tape and a work-tape of which, on input $x \in \Sigma^*$, only the first $O(\log(|x|))$ cells can be used. Moreover, a string $y \in \Sigma^*$ 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 of $R \subseteq \Sigma^* \times \Sigma^*$ is said to be accepted by M if for every input x, it holds that $M(x) = \{y \in \Sigma^* \mid (x, y) \in R\}$.

The first complexity class of relations studied in this paper 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) [22] and a polynomial-time Las Vegas algorithm 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 schemata were known for it [19, 24]. 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 obtain the first complexity class with a simple definition based on Turing Machines, and where each problem admits an FPRAS.

Proviso. The main results of the paper are given in Section 3, while the sketches of the proofs of these results are presented in Sections 5 and 6. Due to the lack of space, the full proofs will be given in the journal version of this article.

2 PRELIMINARIES

Relations and problems. Let Σ be a finite alphabet with at least two symbols. As usual, we represent inputs as words $x \in \Sigma^*$ and the length of x is denoted by $|x|$. A problem is represented as a relation $R \subseteq \Sigma^* \times \Sigma^*$. 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 or witness to that input. For each $x \in \Sigma^*$, we define the set $W_R(x) = \{y \in \Sigma^* \mid (x,y) \in R\}$, and call it the witness set for x. Also, if $y \in W_R(x)$, we call y a witness or a solution to x.

This is a very general framework, so mostly we work with relations that meet two additional properties. First, we only work with relations where both the input and the witnesses have a finite encoding. Second, we work with p-relations [22], namely, each R satisfies that (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 $(x,y) \in \Sigma^* \times \Sigma^*$, 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 witnesses can be made to have the same length through padding.

Enumeration, counting and uniform generation. Given a p-relation R, we are interested in the following problems:

Problem:	ENUM(R)
Input:	A word $x \in \Sigma^*$
Output:	Enumerate all $y \in W_R(x)$ without repetitions

Problem:	COUNT(R)
Input:	A word $x \in \Sigma^*$
Output:	The size $|W_R(x)|$

Problem:	GEN(R)
Input:	A word $x \in \Sigma^*$
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 that the elementos 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 $W_R(x) = \varnothing$.

Enumeration with polynomial and constant delay. An enumeration algorithm for ENUM(R) is a procedure that receives an input $x \in \Sigma^*$ 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 paper, 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 [23] and is defined as follows. An enumeration algorithm is of polynomial delay if there exists a polynomial p such that for every input $x \in \Sigma^*$, 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 the last years [10, 13, 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 paper, 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 [10, 13, 31].

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 $R \subseteq \Sigma^* \times \Sigma^*$, 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 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 [31] in two aspects. First, as it was previously mentioned we relax the distinction between query and data in the preprocessing phase, allowing our algorithm to take polynomial time in the input (i.e. combined complexity). Second, our definition of constant-delay is what in [10, 13] is called *linear delay in the size of the output*, namely, writing the next output is linear in its size and not depending 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 [10, 13] means linear in $|y|$. 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).

Approximate counting. Given a relation $R \subseteq \Sigma^* \times \Sigma^*$, the problem COUNT$(R)$ can be solved efficiently if there exists a polynomial-time algorithm that, given $x \in \Sigma^*$, computes $|W_R(x)|$. In other words, if we think of COUNT(R) as a function that maps x to the value $|W_R(x)|$, then COUNT(R) can be computed efficiently if COUNT$(R) \in$ 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 COUNT(R). More precisely, COUNT(R) is said to admit a fully polynomial-time randomized approximation scheme (FPRAS) [22] if there exists a randomized algorithm $\mathcal{A} : \Sigma^* \times (0,1) \to \mathbb{N}$ and a polynomial $q(u,v)$ such that for every $x \in \Sigma^*$ and $\delta \in (0,1)$:

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

and the number of steps 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 $\frac{1}{\delta}$.

Las Vegas uniform generation. The problem GEN(R) can be solved efficiently if there exists a polynomial-time randomized algorithm that, given $x \in \Sigma^*$, generates an element of $W_R(x)$ with uniform probability distribution (if $W_R(x) = \varnothing$, then it returns $\perp$). However, as in the case of 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, GEN(R) is said to admit a polynomial-time Las Vegas uniform generator (PLVUG) if there exists a randomized algorithm $\mathcal{G} : \Sigma^* \to \Sigma^* \cup \{\perp, \mathbf{fail}\}$, a polynomial $q(u)$ and a function $\varphi : \Sigma^* \to (0,1)$ such that for every $x \in \Sigma^*$:

(1) $\mathbf{Pr}(\mathcal{G}(x) \neq \mathbf{fail}) \geq \frac{1}{2}$;
(2) if $W_R(x) \neq \varnothing$, then $\mathbf{Pr}(\mathcal{G}(x) = \perp) = 0$;
(3) for every $(x,y) \in \Sigma^* \times \Sigma^*$:
  (a) if $(x,y) \notin R$, then $\mathbf{Pr}(\mathcal{G}(x) = y) = 0$;
  (b) if $(x,y) \in R$, then $\mathbf{Pr}(\mathcal{G}(x) = y) = \varphi(x)$;
(4) the number of steps needed to compute $\mathcal{G}(x)$ is at most $q(|x|)$.

The invocation $\mathcal{G}(x)$ can fail in generating an element of $W_R(x)$, in which case it returns **fail**. By condition (1), we know that this probability of failing is smaller than $\frac{1}{2}$, so that by invoking $\mathcal{G}(x)$ several times we can make this probability arbitrarily small (for example, the probability that $\mathcal{G}(x)$ returns **fail** in 100 consecutive independent invocations is at most $(\frac{1}{2})^{100}$). Assume that the invocation $\mathcal{G}(x)$ does not fail. If $W_R(x) = \varnothing$, then we have by condition 3 (a) that $\mathcal{G}(x) = \perp$, so the randomized algorithm indicates that there is no witness for x in this case. If $W_R(x) \neq \varnothing$, then we have by conditions (2) and (3) that $\mathcal{G}(x)$ returns an element $y \in W_R(x)$. Moreover, we know by condition 3 (b) that the probability of returning such an element y is $\varphi(x)$. Thus, we have a uniform generator in this case, as the probability of returning each $y \in W_R(x)$ is the same. Finally, we have that $\mathcal{G}(x)$ can be computed in polynomial time in the $|x|$.

It is important to notice that the notion of polynomial-time Las Vegas uniform generator corresponds to the notion of uniform generator used in [22]. However, we have decided to use the term "Las Vegas" to emphasize the fact that there is a probability of failing in returning a solution. Moreover, the notion of polynomial-time Las Vegas uniform generator imposes stronger requirements than the notion of fully polynomial-time almost uniform generator introduced in [22]. 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 a string $y \in W_R(x)$ with a probability in an interval $[\varphi(x) - \delta, \varphi(x) + \delta]$ for a given error $\delta \in (0,1)$, where φ is defined as in the notion of PLVUG.

3 NL 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. Given a finite alphabet Σ, an NL-transducer M is a nondeterministic Turing Machine with input and output alphabet Σ, 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 that 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 $y \in \Sigma^*$ 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 $\{(x, y) \in \Sigma^* \times \Sigma^* \mid y \in M(x)\}$.

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

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

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

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 counting and uniformly generating such truth assignments, respectively. It is not difficult to see that SAT-DNF $\in$ RelationNL. Hence, given that COUNT(SAT-DNF) is a #P-complete problem, we cannot expect COUNT(R) to be solvable in polynomial time for every $R \in$ RelationNL. However, COUNT(SAT-DNF) admits an FPRAS [25], 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 $n^{O(\log(n))}$ [24]. In our notation, this problem is represented by the following relation:

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

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.

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

It is worth mentioning a fundamental consequence of this result in computational complexity. The class of function SpanL was introduced in [3] to provide a characterization of some functions that are hard to compute. More specifically, given a finite alphabet Σ, a function $f : \Sigma^* \to \mathbb{N}$ is in SpanL if there exists an NL-transducer M with input alphabet Σ such that $f(x) = |M(x)|$ for every $x \in \Sigma^*$. The class SpanL is contained in #P, and it has been instrumental in proving that some functions are difficult to compute [3, 8, 20, 26], as if a function f is complete for SpanL and $f \in$ FP, then P = NP [3]. Given that #NFA is SpanL-complete under parsimonious reductions [3], and parsimonious reductions preserve the existence of an FPRAS, we obtain the following corollary from Theorem 3.2.

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

Although some classes of functions $\mathcal{C}$ for which every $f \in \mathcal{C}$ admits an FPRAS have been identified before [9, 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 [22], for the class of relations that are *self-reducible*. Thus, one might wonder whether the existence of a PLVUG for GEN(R) in Theorem 3.2 is a corollary of the result in [22], as in this theorem we prove the existence of an FPRAS for COUNT(R). Interestingly, the answer to this question is no, as the notion of PLVUG ask for a uniform generator without an error δ, whose existence cannot be inferred from the results in [22]. Thus, we prove in Section 6 that COUNT(R) admits an FPRAS and GEN(R) admits a PLVUG, for a relation $R \in$ RelationNL, without relaying in the aforementioned result from [22].

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 us to define 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 an 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]) and adapted for our case, namely, problems defined as relations.

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

For RelationUL, we obtain the following result.

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

In particular, it should be noticed that given R in the class RelationUL and an input x, the solutions for x can be enumerated, counted and uniformly generated efficiently.

Classes of problems definable by machine models and that can be enumerated with constant delay have been proposed before. In [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, this is the first class with a simple and robust definition based on Turing Machines.

4 APPLICATIONS OF THE MAIN RESULTS

Before providing proof sketches 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 like information extraction, graph databases, and binary decision diagrams.

4.1 Information extraction

In [15], 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 [16] to state our main results. Given the lack of space, we only recall the main definitions (see [15, 16] 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 open or close by $\mathcal{A}$, respectively. A run ρ over a document $d = a_1 \cdots a_n$ is a sequence of the form: $q_0 \xrightarrow{X_1} p_0 \xrightarrow{a_1} q_1 \xrightarrow{X_2} p_1 \xrightarrow{a_2} \ldots \xrightarrow{a_n} q_n \xrightarrow{X_{n+1}} p_n$ 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 [17, 27], 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 [16] a subclass of eVA is considered in order to recover polynomial-time evaluation. A 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 knows 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 relation:

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

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 PLVUG.*

In [5, 16], it was shown that every functional RGX or functional VA (not necessarily extended) can be converted in polynomial time into a functional eVA. Therefore, Corollary 4.1 also holds for these more general classes. Notice that in [18], it was given a polynomial-delay enumeration algorithm for $[\![\mathcal{A}]\!](d)$. Thus, only the results about COUNT(EVAL-eVA) and GEN(EVAL-eVA) are new.

Regarding efficient enumeration and exact counting, a constant-delay algorithm with polynomial preprocessing was given in [16] for the class of deterministic functional eVA. Here, we can easily extend these results for a more general class, that 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 [16] 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 give a constant-delay algorithm with polynomial preprocessing for the class of unambiguous functional eVA. Instead, the algorithm in [16] 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 [7]. 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, nodes represent pieces of data and edges specify relations between them [7]. 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 nodes $u, v \in V$, one would like to retrieve, count, or uniformly generate all paths[1] in G going from u to v that satisfies 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 $p_1 p_2 \cdots p_n \in \mathcal{L}(R)$. The length of π is defined as $|\pi| = n$. Clearly, between u and v there can be an infinite number of paths that satisfies Q. For this reason, one usually wants to retrieve all paths between u and v of at most 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 $|\pi| \leq n$. This naturally defines the following relation representing the problem of evaluating an RQP 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. Notice that giving a polynomial-delay enumeration algorithm for EVAL-RPQ is straightforward, but the existence of an FPRAS and a PLVUG for EVAL-RPQ was not known before when queries are part of the input (i.e. in combined complexity).

Corollary 4.3. COUNT(EVAL-RPQ) *admits an FPRAS, and* GEN(EVAL-RPQ) *admits a PLVUG.*

4.3 Binary decision diagrams

Binary decision diagrams (OBDDs) are an abstract representation of boolean functions which are widely used in computer science and have found many applications in areas like formal verification [12]. A binary decision diagram (BDD) is a directed acyclic graph $D = (V, E)$ where each node 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 nodes when $\text{var}(v)$ takes values 0 and 1, respectively. D contains only two terminal, or sink nodes, labeled by 0 or 1, and one initial node called v_0. We assume that every path from v_0 to a terminal node 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 node 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 node reached by starting from v_0 and following the values in σ. For Ordered BDDs, 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 node v_0 to a terminal node 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, from which we obtain:

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

[1]Notice that the standard semantics for RPQs is to retrieve pair of nodes. Here we consider a less standard semantics based on paths which is also relevant for graph databases [7, 8, 26].

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 is to use UL transducers to realize the good algorithmic properties that a data structure has.

Some non-deterministic variants of BDDs have been studied in the literature [6]. In particular, an nOBDD extends an OBDD with vertices u without variables (i.e. $\text{var}(u) = \perp$) 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 node v_0 to a terminal node with labeled 0 or 1. Without loss 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 called 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 looses the single witness property, and now an assignment σ can have several paths from the initial node to the 1 terminal node. Thus, it is not clear whether EVAL-nOBDD is in RelationUL. Still one can easily show that EVAL-nOBDD $\in$ RelationNL, from which the following results follow.

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

It is important to stress that the existence of an FPRAS and a PLVUG for EVAL-nOBDD was not known before.

5 A SIMPLE NOTION OF COMPLETENESS, AND ITS APPLICATION TO RelationUL

The goal of this section is to define a simple notion of reduction for the classes RelationNL and RelationUL, and then to show how it can be used to prove Theorem 3.5. In Section 6, we use this notion again when proving Theorem 3.2.

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 [14]. However, we do not intent to discuss them here; instead, we use an idea of completeness that is very restricted, but that turns out to be useful for the classes we defined. Let $\mathcal{C}$ be a complexity class of relations, and let $R, S \in \mathcal{C}$. We say R is reducible to S if there exists a function $f : \Sigma^* \to \Sigma^*$, computable in polynomial time, such that for every $x \in \Sigma^*$: $W_R(x) = W_S(f(x))$. Also, if T is reducible to S for every $T \in \mathcal{C}$, we say S is complete for $\mathcal{C}$. Notice that this definition is very strict, since the notion of reduction requires the witness set to be exactly the same for both relations (is not sufficient that they have the same size, for example). The benefit behind 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 the following result.

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

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

Therefore, by finding a complete relation S for a class $\mathcal{C}$, we can just study the aforementioned problems for S, and we know that the obtained results will extend to every relation in the class $\mathcal{C}$.

5.1 Complete problems for RelationNL and RelationUL

The notion of reduction just defined above is useful for us as RelationNL and RelationUL admit complete problems under this notion. These complete relations are defined in terms of NFAs, and the idea behind them is the following. Take a relation R in RelationNL (the case for RelationUL is very similar). We know there is an NL-transducer M that characterizes it. Consider now some input x. Since M is a non-deterministic logspace Turing Machine, there is only a polynomial number of different configurations that M can be in (polynomial on $|x|$). So we can consider the set of possible configurations as the states of an NFA N_x, which has polynomial size, and whose transitions are determined by the transitions between the configurations of M. Moreover, whenever a symbol is output by the transducer M, that symbol is read by the automaton N_x. In this way, N_x accepts exactly the language $W_R(x)$. We formalize this idea in the following result, where

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

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

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

5.2 Algorithmic properties of RelationUL

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

Proposition 5.3. 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).

The results for COUNT(MEM-UFA) is a corollary of the fact that there exists a polynomial time algorithm that, given an input string x, returns the number of accepting runs of a non-deterministic logspace Turing Machine with input x [3]. Moreover, the result for GEN(MEM-UFA) can be obtained by considering that COUNT(MEM-UFA) can be solved in polynomial time and MEM-UFA is a self-reducible relation [22], and then using a strategy similar to the one described in [22]. On the other hand, the result for ENUM(MEM-UFA) does require a more elaborated proof that we we outline here.

Let $(N, 0^k)$ be an input of ENUM(MEM-UFA). In the preprocessing phase of the constant-delay enumeration algorithm for this problem, the NFA N is unrolled to get rid of any cycles it might have, and keep only the accepted words of length exactly k, which are the ones we want to enumerate. For the unrolling, we create $k+1$ layers of nodes, being each layer a copy of the set of states of N. And for each transition in N, we connect each layer with the next one, by joining the corresponding nodes with a directed edge, and labeling the edge according to the symbol in the transition. Given that N is an unambiguous NFA, this gives us a directed acyclic graph G, where each word w of length k accepted by N has a unique corresponding path between a fixed start node and a fixed end node in G, such that the labels read along the way form the string w. From that, it is not difficult to see how to enumerate with constant delay. We just have to go through G, beginning in the "start node", and traversing it in a depth-first search manner. During this process, we store the symbols read, and output them any time we reach the end node of G.

6 APPROXIMATE COUNTING AND UNIFORM GENERATION OF 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 show 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 PLVUG*

As mentioned in Section 5.2, we have that MEM-NFA is a self-reducible relation [22]. Besides, the existence problem for MEM-NFA (that is, for a given input $(N, 0^k)$, decide whether there are any witnesses) can be solved in polynomial time. With all that, we can derive the existence of a polynomial delay algorithm for ENUM(MEM-NFA) as a direct application of Theorem 4.9 from [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 Theorem 6.6. As this algorithm works by simultaneously counting and doing uniform generation of witnesses, its existence not only gives us an FPRAS for COUNT(MEM-NFA), but also a PLVUG for GEN(MEM-NFA), as formally stated in Corollary 6.7.

6.1 The Algorithm Template

As mentioned in Section 3, we consider the following approximation problem. The input of the problem is an NFA N on the alphabet $\{0,1\}$ with m states (and no epsilon transitions), a string 0^n that represents an integer $n \geq 1$ given in unary, and an error $\delta \in (0,1)$. The problem then is to return R such that R is a $(1 \pm \delta)$-approximation of $|\mathcal{L}_n(N)|$, that is, $(1-\delta)|\mathcal{L}_n(N)| \leq R \leq (1+\delta)|\mathcal{L}_n(N)|$, where $\mathcal{L}(N)$ is the set of strings accepted by N and $\mathcal{L}_n(N) = \{w \in \{0,1\}^* \mid w \in \mathcal{L}(N) \text{ and } |w| = n\}$. Besides, such an approximation should be returned in time polynomial in m, n and $\frac{1}{\delta}$ (notice that the size of NFA N is $O(m^2)$, so being polynomial in m means being polynomial in the size of N).

Our algorithm for approximating $\mathcal{L}_n(N)$ will involve the construction of a directed acyclic graph from the NFA N. We call this directed acyclic graph N_{unroll}, as it is obtained by unrolling n times the NFA N . Formally, assume that $N = \{s_1, \ldots, s_m\}$ and s_1 is the initial state of N. Then for every state $s_i \in N$ create n states $s_i^1, \ldots, s_i^n$ in N_{unroll}, and for every transition $s_i \xrightarrow{b} s_j$ in N and $b \in \{0,1\}$, create the transition $s_i^t \xrightarrow{b} s_j^{t+1}$ for all $t = 1, 2, \ldots, n-1$ in N_{unroll}. Moreover, include a vertex s_{start} in N_{unroll} with transitions $s_{start} \xrightarrow{b} s_i^1$ if there is a transition $s_1 \xrightarrow{b} s_i$ in N (recall that s_1 is the initial state of N). Finally, create a unique final state s_{final} for N_{unroll}, and for every accepting state s_j of N, add to N_{unroll} the transition $s_j^n \xrightarrow{1} s_{final}$. We will use the terms *vertex* and *state* interchangeably to refer to the vertices of N_{unroll}. We refer to the set $\{s_1^t, s_2^t, \ldots, s_m^t\}$ as the t-th layer of N_{unroll}. The vertex set of N_{unroll} is precisely $\{s_{start}, s_{final}\} \cup (\bigcup_{t=1}^{n} \{s_1^t, s_2^t, \ldots, s_m^t\})$.

Remark 1. *Notice that s_{final} is included in N_{unroll} to have a unique final state. Besides, notice that for each final state s_j of*

N, the last occurrence of such a state when processing a string of length n is connected with s_{final} via the same symbol 1, that is, the transition $s_j^n \xrightarrow{1} s_{final}$ is included in N_{unroll}. Hence, the size of the accepted language is not changed, as the number of distinct strings which give a path from s_{start} to s_{final} in N_{unroll} is precisely $|\mathcal{L}_n(N)|$.

We say that a string w is a *member of* a vertex $s \in N_{unroll}$ if there is a path from s_{start} to s in N_{unroll} where the string of ordered labels of the edges is precisely w. We write $U(s)$ to denote the set of strings which are members of a state s. Note that $|U(s_{final})| = |\mathcal{L}_n(N)|$, and $U(s_{start}) = \{\varepsilon\}$. Thus, our goal is to produce a good estimate of the value $|U(s_{final})|$. For a string w, let $w[t]$ denote the t-th bit in w. Thus if $w = 100101$, we have $w[1] = 1, w[2] = 0, w[3] = 0$, and so on. For strings w, v, let $w \circ v$ denote their concatenation.

Algorithm 1: Algorithmic Template for our FPRAS

```
(1) Construct the directed acyclic graph $N_{unroll}$ from the NFA $N$.
(2) For layers $i = 1, 2, \ldots, n$ and $j = 1, 2, \ldots, m$:
    (a) Compute $R(s_j^i)$ given $\bigcup_{t=1}^{i-1}\bigcup_{j=1}^{m}\{R(s_j^t), X(s_j^t)\}$.
        For $i = 1$, we have that $R(s_j^i)$ is computed without any additional information.
    (b) Call a subroutine to sample $k$ uniform elements of $U(s_j^i)$ using the value $R(s_j^i)$ and
        $\bigcup_{t=1}^{i-1}\bigcup_{j=1}^{m}\{R(s_j^t), X(s_j^t)\}$.
    (c) Let $X(s_j^i) \subseteq U(s_j^i)$ be the multi-set of the $k$ uniform samples obtained.
(3) Return $R(s_{final})$.
```

The components of the main algorithm are as follows. We set $k = (\frac{nm}{\delta})^c$ for some sufficiently large constant c (to be defined later). Then for each vertex $s \in N_{unroll}$ (where $s = s_j^t$ for some $j \in \{1, \ldots, m\}$ and $t \in \{1, \ldots, n\}$), we store k strings $x_1, \ldots, x_k$, such that each $x_i \in U(s)$. Specifically, the x_i's are uniform samples of the set $U(s)$. We denote this set of k samples for the vertex s by $X(s) \subseteq U(s)$ (if $|U(s)| \leq k$, we set $X(s) = U(s)$). Since the samples will be uniform and independent, it is possible that we will obtain duplicates samples of a given $x \in U(s)$. Therefore, we allow $X(s)$ to be a multi-set (meaning that $X(s) = \{x_1, \ldots, x_k\}$, and the strings x_i are not necessarily distinct). Second, we store a value $R(s)$ which is a $(1 \pm \delta)$-approximation of $|U(s)|$. The algorithm proceeds like a dynamic programming algorithm, computing $R(s)$ and $X(s)$ for every state s in N_{unroll} in a breadth-first search ordering. We first compute $R(s), X(s)$ for all states s in layer 1, meaning $\{s_1^1, s_2^1, \ldots, s_m^1\}$. Then for any layer i, given the values $\bigcup_{t=1}^{i-1}\bigcup_{j=1}^{m}\{R(s_j^t), X(s_j^t)\}$, we compute the corresponding values $R(s_j^i), X(s_j^i)$ for each vertex s_j^i in layer i. So the values $R(s), X(s)$ are computed layer by layer. The final estimate for $|\mathcal{L}_n(N)|$ is $R(s_{final})$. We summarize this algorithmic template in Algorithm 1.

6.2 The Sampling Template

To carry out our main approximation algorithm, we must implement the algorithmic template given in Algorithm 1. In particular, we must implement the sampling subroutine in step 2 (b). We begin by describing a generic sampling template for this step, which will be used by our main algorithm as a subroutine. The procedure is essentially that of [22], but modified to suit our setting. The procedure to sample a uniform element of a set $U(s_j^\alpha)$ is as follows. We initialize a string w^α to be the empty string, we construct a sequence of strings $w^\alpha, w^{\alpha-1}, \ldots, w^1, w^0$, where each string w^t is of the form $b_t \circ w^{t+1}$ with $b_t \in \{0, 1\}$, and we define the result of the sample procedure to be w^0. To ensure that w^0 is an element of $U(s_j^\alpha)$ chosen with uniform distribution, we also consider a sequence of sets of strings $W^\alpha, W^{\alpha-1}, \ldots, W^1, W^0$ constructed as follows. We have that $W^\alpha = U(s_j^\alpha)$. Then we partition W^α into two sets of strings: those with last bit equal to 0 and with last bit equal to 1, which are called W_0^α and W_1^α, respectively. We estimate the size of each partition, and choose one of them with probability proportional to its size, say W_b^α. We then append the bit b the prefix of w^α to obtain $w^{\alpha-1} = b \circ w^\alpha$, we define $W^{\alpha-1}$ as $\{x \mid x \circ b \in W^\alpha \text{ and } |x| \geq 1\}$, and we recurse on $w^{\alpha-1}$ and $W^{\alpha-1}$. Thus, in general, we have that W^t is the set of strings x such that $x \circ w^t \in U(s_j^\alpha)$, and we also have that $W^0 = \varnothing$.

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 $1 - \varphi$, 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 $\mu > 0$. For the sake of simplicity, we first assume that we have perfect estimates of the sizes of the partitions in question. This procedure is given below in Algorithm 2. We call it with the initial parameters **SampleTemplate**$(W^\alpha, \varepsilon, \varphi_0)$, where ε is the empty string, corresponding to the goal of sampling a uniform element of $W^\alpha = U(s_j^\alpha)$. Here, φ_0 is a value that we will later choose. Specifically, φ_0 will be a constant times a $(1 \pm \delta)$-approximation of $|U(s_j^\alpha)|$.

At every step j of Algorithm 2, we have that $|W^j|$ is precisely the number of strings in W^α which have the suffix

w^j, as W^j is the set of strings x such that $x \circ w^j \in W^\alpha$. Note then that the set W^j *depends* on the random string w^j, so in fact we should write $W^j_{w^j}$ instead of W^j, but for notational simplicity we omit the subscript, and it is then understood that W^j is a function of w^j.

Now the probability of choosing a given element $x \in W^\alpha$ can be computed as follows. Ignoring for a moment the possibility of returning **fail**, we have that w^0 is the string returned by **SampleTemplate**$(W^\alpha, \varepsilon, \varphi_0)$ since $W^0 = \varnothing$. Thus, we probability we chose x is:

$$\mathbf{Pr}(w^0 = x) = \frac{|W^{\alpha-1}|}{|W^\alpha|} \cdot \frac{|W^{\alpha-2}|}{|W^{\alpha-1}|} \cdot \dots \cdot \frac{|W^1|}{|W^2|} \cdot \frac{1}{|W^1|} = \frac{1}{|W^\alpha|}$$

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

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

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

$$\begin{aligned}(1 - |W^\alpha|\varphi_0)^{\ell\cdot(|W^\alpha|\varphi_0)^{-1}} &\le (e^{-|W^\alpha|\varphi_0})^{\ell\cdot(|W^\alpha|\varphi_0)^{-1}} \\ &= e^{-|W^\alpha|\varphi_0\cdot\ell\cdot(|W^\alpha|\varphi_0)^{-1}} = e^{-\ell}.\end{aligned}$$

Algorithm 2: SampleTemplate(W^j, w^j, φ)

```
(1) If $W^j = \varnothing$, then with probability $\varphi$ return $w^j$,
    otherwise return fail.
(2) Else, partition $W^j$ into two sets, those with last
    bit equal to 0, call this $W^j_0$, and those with last bit
    equal to 1, call this $W^j_1$.
(3) Then choose partition $b \in \{0,1\}$ with probability
    $p_b = \frac{|W^j_b|}{|W^j|}$, and set $W^{j-1} = \{x \mid x \circ b \in W^j$ and $|x| \ge 1\}$, and $w^{j-1} = b \circ w^j$.
(4) Return SampleTemplate$(W^{j-1}, w^{j-1}, \frac{\varphi}{p_b})$.
```

6.3 The Main Algorithm

We now describe our main algorithm formally. As previously mentioned, the algorithm computes the values of $R(s)$, $X(s)$ in a breadth-first search order on the graph N_{unroll}. Thus we first compute $R(s)$, $X(s)$ for all s in layer i, and then move on to layer $i+1$. Our algorithm for computing the samples needed in $X(s^\alpha_i)$ for a fixed state s^α_i is given in Algorithm 3, and our full FPRAS is given in Algorithm 4.

Base Case: For every state s^α_i such that $|U(s^\alpha_i)| \le k$, compute and store $R(s) = |U(s)|$ exactly, and store the entire set $U(s) = X(s)$. We call these states *exactly handled*. To do this, we perform a breadth-first search from s_{start}. At every new state s we see, we check if all the states in the prior layer with edges into s are exactly handled (if not, then s is not exactly handled). If so, then we compute $|U(s)|$ by computing the union Y_0 of all $X(s')$, where s' ranges over all states with edges into s labeled with a 0, and then computing the union Y_1 of all $X(s'')$, where s'' ranges over all states with edges into s labeled with a 1. If $|Y_0| + |Y_1|$ is at most k, then we set $X(s)$ to be $\{x \circ 0 \mid x \in Y_0\} \cup \{x \circ 1 \mid x \in Y_1\}$, and we set $R(s) = |X(s)|$. Otherwise, we conclude that $|U(s)| > k$, and thus s is not exactly handled.

Inductive Case: Suppose we have a state s^α_i that is not exactly handled, and for which we have not computed $X(s^\alpha_i)$, $R(s^\alpha_i)$, but such that we have computed $X(s^t_j)$, $R(s^t_j)$ for $j = 1, 2, \dots, m$ and $t = 1, 2, \dots, \alpha - 1$. To build the set of samples $X(s^\alpha_i)$, we call the procedure **Sample**(T, w, φ) a total of k times, where T is some subset of states (all in the same layer), w is a string suffix, and $\varphi > 0$ is some small value (T, w and φ will be specified later). Notice that **Sample** is the instantiation of the procedure **SampleTemplate** described in the previous section to the specific requirements of our main algorithm. Given any fixed arbitrary linear ordering $\prec$ on the states of N_{unroll}, the procedure **Sample** is defined as shown in Algorithm 3.

Algorithm 3: Sample(T, w, φ)

```
(1) If $\varphi \notin (0,1)$ return fail.
(2) If $T = \{s_{start}\}$, then with probability $\varphi$ return $w$.
    Else, with probability $1 - \varphi$, return fail.
(3) Else, define:
        $T_0 = \{s^{r-1}_j \in N_{unroll} \mid s^{r-1}_j \xrightarrow{0} s^r_i$ for some $s^r_i \in T\}$
        $T_1 = \{s^{r-1}_j \in N_{unroll} \mid s^{r-1}_j \xrightarrow{1} s^r_i$ for some $s^r_i \in T\}$
    Note $T_0 \cap T_1$ may be non-empty. Then:
    (a) For $q \in \{0,1\}$, compute
            $\widetilde{W}_q = \sum_{s \in T_q} R(s) \cdot \frac{|X(s) \setminus (\bigcup_{s' \in T_q : s' \prec s} U(s'))|}{|X(s)|}$
    (b) For $q \in \{0,1\}$, set $p_q = (\widetilde{W}_q)/(\widetilde{W}_0 + \widetilde{W}_1)$, and
        then choose $b \in \{0,1\}$ with probability $p_b$.
(4) Return Sample$(T_b, b \circ w, \frac{\varphi}{p_b})$
```

It is important to note that $X(s) \setminus (\bigcup_{s' \in T_q, s' \prec s} U(s'))$ in step 3 (a) can be computed in polynomial time by simply iterating through each $x \in X(s)$, and checking whether there is a path from s_{start} to some $s' \in T_q$, with $s' \prec s$, where the string of ordered labels of the edges is precisely x, which can be done by a breadth-first search.

Algorithm 4: FPRAS to estimate $|\mathcal{L}_n(N)|$ for a NFA N with $m \geq 1$ states, $n \geq 1$ given in unary and error $\delta \in (0,1)$

```
(1) If $n \leq 12$, then return $|\{x \in \{0,1\}^n \mid x \in \mathcal{L}(N)\}|$ (this can be done in polynomial time by an exhaustive search)
(2) Construct the directed acyclic graph $N_{unroll}$ from $N$, and set $k = \lceil (\frac{nm}{\delta})^{64} \rceil$.
(3) For each vertex $s \in N_{unroll}$, if there is not a path from the starting vertex $s_{start}$ to $s$, remove $s$ from $N_{unroll}$.
(4) For layers $\alpha = 1, 2, \dots, n$ and for $i = 1, 2, \dots, m$:
    (a) For $b \in \{0,1\}$, let $T_b(s_i^\alpha) = \{s \in N_{unroll} \mid$ there is an edge $s \xrightarrow{b} s_i^\alpha$ in $N_{unroll}\}$. Let $T(s_i^\alpha) = T_0(s_i^\alpha) \cup T_1(s_i^\alpha)$, and for
        $b \in \{0,1\}$, assume that $T_b(s_i^\alpha) = \{v_1^b, \dots, v_{r_b}^b\}$ where $r_b = |T_b(s_i^\alpha)|$.
    (b) If $T(s_i^\alpha) = \{s_{start}\}$ (meaning if $\alpha = 1$), set $X(s_i^\alpha) = \{b \in \{0,1\} \mid s_{start} \xrightarrow{b} s_i^\alpha$ is an edge in $N_{unroll}\}$. Moreover, set
        $R(s_i^\alpha) = |X(s_i^\alpha)|$, and declare the state $s_i^\alpha$ to be exactly handled.
    (c) Else, if $s$ is exactly-handled for all $s \in T(s_i^\alpha)$, set
            $$R(s_i^\alpha) = \left( \sum_{j=1}^{r_0} \left| X(v_j^0) \setminus \bigcup_{t=1}^{j-1} X(v_t^0) \right| \right) + \left( \sum_{j=1}^{r_1} \left| X(v_j^1) \setminus \bigcup_{t=1}^{j-1} X(v_t^1) \right| \right),$$
        and then if $R(s_i^\alpha) \leq k$, declare $s_i^\alpha$ to be exactly handled, and set
            $$X(s_i^\alpha) = \left( \bigcup_{t=1}^{r_0} \{x \circ 0 \mid x \in X(v_t^0)\} \right) \bigcup \left( \bigcup_{t=1}^{r_1} \{x \circ 1 \mid x \in X(v_t^1)\} \right).$$
        Otherwise, (that is, if $R(s_i^\alpha) > k$), do nothing.
    (d) Else (that is, if $s$ is not exactly handled for at least one state $s \in T(s_i^\alpha)$) do nothing.
(5) For layers $\alpha = 1, 2, \dots, n$ and for $i = 1, 2, \dots, m$:
    (a) If $s_i^\alpha$ is exactly handled, then $R(s_i^\alpha)$ and $X(s_i^\alpha)$ are already computed. Otherwise, for $b \in \{0,1\}$ set
            $$\widetilde{W}_b(s_i^\alpha) = \sum_{s \in T_b(s_i^\alpha)} R(s) \cdot \frac{\left| X(s) \setminus \left( \bigcup_{s' \in T_b(s_i^\alpha) : s' \prec s} U(s') \right) \right|}{|X(s)|}$$
        and $R(s_i^\alpha) = \widetilde{W}_0(s_i^\alpha) + \widetilde{W}_1(s_i^\alpha)$.
    (b) If $R(s_i^\alpha) = 0$, terminate the algorithm and output 0 as the estimate (failure event).
    (c) Else, set $X(s_i^\alpha) = \varnothing$. Then while $|X(s_i^\alpha)| < k$
        (i) Set $w$ = **fail**
        (ii) Run **Sample**$(\{s_i^\alpha\}, \varepsilon, \frac{e^{-4}}{R(s_i^\alpha)})$ until it returns a string $w \neq$ **fail**, and at most $\lceil (\frac{nm}{\delta})^4 \rceil$ times
        (iii) If $w$ = **fail** (that is, none of the $\lceil (\frac{nm}{\delta})^4 \rceil$ calls returned a string $w \neq$ **fail**), then terminate the algorithm and
              output 0 as the estimate (failure event).
        (iv) Otherwise, a sample $w \in \{0,1\}^*$ was returned, and set $X(s_i^\alpha)$ as $X(s_i^\alpha) \cup \{w\}$ (recall we allow $X(s_i^\alpha)$ to contain
              duplicates).
(6) Return $R(s_{final})$ as an estimate for $|\mathcal{L}_n(N)|$.
```

By calling the procedure **Sample** on $(\{s_i^\alpha\}, \varepsilon, \varphi_0)$ until it returns k samples (where ε the empty string, and φ_0 is a value we will later choose) we obtain the samples for $X(s_i^\alpha)$. Note that it is possible that duplicate samples will be returned by the **Sample** procedure. This will not be an issue for us, and we can instead assume that $X(s_i^\alpha)$ is a multi-set (thus, $X(s_i^\alpha)$ can have more than one copy of the same element in $U(s_i^\alpha)$). The value of φ_0 that we choose will depend on our estimate $R(s_i^\alpha)$, so before invoking the above recursive procedure to obtain $X(s_i^\alpha)$, we first show how to compute $R(s_i^\alpha)$. To do so, set $T_0(s_i^\alpha) = \{s_q^{\alpha-1} \in N_{unroll} \mid s_q^{\alpha-1} \xrightarrow{0} s_i^\alpha\}$, and $T_1(s_i^\alpha) = \{s_q^{\alpha-1} \in N_{unroll} \mid s_q^{\alpha-1} \xrightarrow{1} s_i^\alpha\}$, and define the linear ordering $\prec$ as above. Then for $b \in \{0,1\}$, compute

$$\widetilde{W}_b = \sum_{s \in T_b(s_i^\alpha)} R(s) \cdot \frac{\left| X(s) \setminus \left(\bigcup_{s' \in T_b(s_i^\alpha) : s' \prec s} U(s') \right) \right|}{|X(s)|},$$

and define $R(s_i^\alpha) = \widetilde{W}_0 + \widetilde{W}_1$. We then set the parameter $\varphi_0 = \frac{e^{-4}}{R(s_i^\alpha)}$, which we use in our calls to **Sample**. This completes the procedure to obtain the desired $X(s_i^\alpha), R(s_i^\alpha)$ pair. After computing $X(s_i^\alpha), R(s_i^\alpha)$ for all states, the final output of the algorithm is $R(s_{final})$ as the approximation to $|\mathcal{L}_n(N)|$.

Summary: We compute the sample set and estimate pair $X(s_j^t)$, $R(s_j^t)$ for all states s_j in layers $t = 1, 2, \ldots, \alpha - 1$. For each state s_i^α such that $|U(s_i^\alpha)| \leq k$, we declare s_i^α to be *exactly handled*. For such exactly handled states, we store $X(s_i^\alpha) = U(s_i^\alpha)$ and $R(s_i^\alpha) = |U(s_i^\alpha)|$ exactly. Otherwise, we compute $R(s_i^\alpha)$ from the samples and estimates in the prior layers $t < \alpha$. Finally, using $R(s_i^\alpha)$ and $X(s_i^t)$, $R(s_i^t)$ for all $t < \alpha$, we invoke **Sample**$(\{s_i^\alpha\}, \varepsilon, \frac{e^{-4}}{R(s_i^\alpha)})$ repeatedly to obtain all k samples needed for $X(s_i^\alpha)$. Once this has been completed for all states in N_{unroll}, we output $R(s_{final})$ as our final estimate. Our full algorithm is given in Algorithm 4.

6.4 The Analysis of the Algorithm

We start by showing that our sampling algorithm **Sample** of Algorithm 3 performs nearly the same procedure as the one described in the template Algorithm 2. Consider the notation used in these algorithms, and fix a state s_i^α in layer α. Let T^t, T_0^t and T_1^t be the set T, T_0 and T_1, respectively, in the $(\alpha - t)$-th recursive call to **Sample**, where the original call to **Sample**$(\{s_i^\alpha\}, \varepsilon, \frac{e^{-4}}{R(s_i^\alpha)})$ is counted as the first, that is, $t = 0$. Moreover, let w^t be the (possibly empty) string in this call, and let $\widetilde{W}_q^t$ for $q \in \{0, 1\}$ be the value of $\widetilde{W}_q$ in this call. Thus, we have that $w^\alpha = \varepsilon$, and $T^\alpha = \{s_i^\alpha\}$. We define the index t in this way so that T^t is a subset of states in the t-th layer (i.e. T^t is a set of states of the form s_j^t for some $j \in \{1, \ldots, m\}$). Notice that the sets T_0^t and T_1^t will be in layers $t-1$ by definition, and $R(s_i^\alpha) = \widetilde{W}_0^\alpha + \widetilde{W}_1^\alpha$. By construction, for $t < \alpha$, we have the property that T^t is the set of states s in layer t such that there is an edge labeled with the bit $w^{\alpha-t}[1]$ to some state $s' \in T^{t+1}$. Given this, the only difference between our sampling algorithm of Algorithm 3 and the template Algorithm 2 is that the sizes of the sets $|W^t|$ are replaced with approximations $\widetilde{W}^t$, since we no longer know $|W^t|$ exactly. We now demonstrate that the procedure of Algorithm 3 does in fact follow the template of Algorithm 2, up to the fact that it uses approximations $\widetilde{W}^t$ of $|W^t|$.

Proposition 6.2. *For every t, it holds that $(\bigcup_{s \in T^t} U(s)) = W^t$, where W^t is defined as in Algorithm 2 as the set of strings x such that $x \circ w^t \in U(s_j^\alpha)$.*

Recall that for $q \in \{0, 1\}$, we defined W_q^t as the set of strings in W^t with last bit equal to q, and that W^t is the set of strings x such that $x \circ w^t \in U(s_i^\alpha)$. Also recall that we initialized $w^\alpha = \varepsilon$, so in general $w^{\alpha-i}$ is a string of length i for every $i = 0, 1, 2, \ldots, \alpha$. As noted, the only difference between our algorithm **Sample** and the template **SampleTemplate** is that at any layer t, instead of choosing w^{t-1} to be $q \circ w^t$ and recursing into the set W_q^t with probability $\frac{|W_q^t|}{|W^t|}$ exactly, we choose q with the approximation probability $\frac{\widetilde{W}_q^t}{\widetilde{W}_0^t + \widetilde{W}_1^t}$ (since we do not know the exact value of $\frac{|W_q^t|}{|W^t|} = \frac{|W_q^t|}{|W_0^t + W_1^t|}$). Recall that the approximation $\widetilde{W}_q^t$ of $|W_q^t|$ is the value of $\widetilde{W}_q$ in the t-th call to **Sample** as in Algorithm 3. The following result can be found in [22], however we provide a proof here to consider the specificities of our setting. The result says that at the point where we attempt to compute uniform samples of the set $U(s_i^\alpha)$, in order to build the sample set $X(s_i^\alpha)$, assuming that we have a good estimate $R(s_i^\alpha)$ of $|U(s_i^\alpha)|$ and good estimates $\widetilde{W}_q^t$ of the sizes of the partitions $|W_q^t|$, our sampling procedure will in fact output a uniformly random sample of $U(s_i^\alpha)$ (and only output **fail** with at most $1 - O(1)$ probability).

Proposition 6.3. *Set $k = \lceil (\frac{mn}{\delta})^{64} \rceil$, where $n \geq 2$, and suppose that we have estimates $\widetilde{W}_q^t = (1 \pm k^{-1/4})^t |W_q^t|$ for all $t = 1, 2, \ldots, \alpha$ and $q \in \{0, 1\}$, and an estimate $R(s_i^\alpha) = (1 \pm k^{-1/4})^\alpha |U(s_i^\alpha)|$. If $w \neq$ **fail** is the output of the procedure* **Sample**$(\{s_i^\alpha\}, \varepsilon, \frac{e^{-4}}{R(s_i^\alpha)})$*, then for every $x \in U(s_i^\alpha)$:*

$$\mathbf{Pr}(w = x) \quad = \quad \frac{e^{-4}}{R(s_i^\alpha)}.$$

Moreover, it outputs **fail** *with probability at most $1 - e^{-5}$. Thus, conditioned on not failing,* **Sample**$(\{s_i^\alpha\}, \varepsilon, \frac{e^{-4}}{R(s_i^\alpha)})$ *returns a uniform element $x \in U(s_i^\alpha)$.*

Proposition 6.3 demonstrates that our sampler is indeed uniform, provided our estimates $R(s_i^\alpha)$ and $\widetilde{W}_q^t$ satisfy the stated assumptions. Our next goal is to show that, when tasked with computing samples for the set $X(s_i^\alpha)$, the conditions of Proposition 6.3 will indeed hold. Note that while our sampler only returns a sample with probability e^{-5}, by repeating the procedure some τ times, at least one sample will be returned with probability $1 - e^{-c \cdot \tau}$, where $c > 0$ is a fixed constant. Since our algorithm needs only nmk samples, we can bound the probability of error by the union of getting at least one sample out of every τ attempts. This blows up the complexity of our algorithm by a τ factor only, preserving the polynomial time if τ is polynomial in $\frac{nm}{\delta}$.

To facilitate our analysis, we introduce two properties. On termination of our algorithm, we define the following properties for each state s_i^α:

Property 1: $R(s_i^\alpha) = (1 \pm k^{-1/4})^\alpha |U(s_i^\alpha)|$,

Property 2: for every subset $L \subseteq \{1, \ldots, m\}$, it holds:

$$\left| \frac{|X(s_i^\alpha) \setminus (\bigcup_{j \in L} U(s_j^\alpha))|}{|X(s_i^\alpha)|} - \frac{|U(s_i^\alpha) \setminus (\bigcup_{j \in L} U(s_j^\alpha))|}{|U(s_i^\alpha)|} \right| < k^{-1/3}$$

In other words, Property 1 means that our estimate $R(s_i^\alpha)$ for the size of the set $U(s_i^\alpha)$ is within our desired error bounds. Property 2 asserts that the sampled subset $X(s_i^\alpha) \subseteq$

$U(s_i^\alpha)$ is a *good* approximation of the set $U(s_i^\alpha)$ in the following sense: for every set of the form $U(s_i^\alpha) \setminus \left(\bigcup_{j\in L} U(s_j^\alpha)\right)$ such that our algorithm may at some point attempt to estimate the ratio $|U(s_i^\alpha) \setminus \left(\bigcup_{j\in L} U(s_j^\alpha)\right)|/|U(s_i^\alpha)|$ as in step $3(a)$ of Algorithm 3, we will get a good approximation of this ratio by using $|X(s_i^\alpha) \setminus \left(\bigcup_{j\in L} U(s_j^\alpha)\right)|/|X(s_i^\alpha)|$ instead. We now consider a fixed point in the execution of the algorithm, and show that if Properties 1 and 2 hold for all nodes in N_{unroll} at depth $t = 1, 2, \ldots, \alpha - 1$, then on a call to sample a string from $U(s_i^\alpha)$ for a fixed s_i^α, the assumptions of Proposition 6.3 will be satisfied.

Proposition 6.4. *Fix a state s_i^α for $i \in \{1, \ldots, m\}$ and $\alpha \in \{1, \ldots, n\}$, and set $k = \lceil (\frac{mn}{\delta})^{64} \rceil$. Suppose that for every $t \in \{1, \ldots, \alpha-1\}$ and $j \in \{1, \ldots, m\}$, the states s_j^t satisfy both Properties 1 and 2. Then on query to* **Sample**$(\{s_i^\alpha\}, \varepsilon, \frac{e^{-4}}{R(s_i^\alpha)})$ *for each $i \in \{1, \ldots, m\}$, the conditions of Proposition 6.3 hold: namely that $\widetilde{W}_q^t = (1 \pm k^{-1/4})^t |W_q^t|$ for every $t \in \{1, \ldots, \alpha\}$ and $q \in \{0, 1\}$, and $R(s_i^\alpha) = (1 \pm k^{-1/4})^\alpha |U(s_i^\alpha)|$. In particular, this implies that s_i^α satisfies Property 1 for all $i \in \{1, \ldots, m\}$.*

Let $\mathcal{E}^r$ be the event that Properties 1 and 2 hold for s_j^r for all $j \in \{1, \ldots, m\}$. Note for every layer r where s_j^r is exactly handled for all $j \in \{1, \ldots, m\}$, the event $\mathcal{E}^r$ holds with probability 1. Call a layer exactly handled if all the states in it are exactly handled.

The following Lemma, which is a consequence of Hoeffding inequality [21], demonstrates that if Properties 1 and 2 hold for all states s_j^t in layers $t = 1, 2, \ldots, \alpha - 1$, then after completion of the sampling procedure which constructs $X(s_i^\alpha)$ and the estimate $R(s_i^\alpha)$ for a fixed state s_i^α, we will have that s_i^α satisfies both Properties 1 and 2. This result will allow us to inductively show that all vertices in the graph N_{unroll} satisfy Properties 1 and 2. In particular, this means that Property 1 will hold for the final state s_{final}, which implies that $R(s_{final}) = (1 \pm k^{-1/4})^n |U(s_{final})| = (1 \pm \delta)\mathcal{L}_n(N)$, which is our desired approximation.

Lemma 6.5. *Conditioned on $\mathcal{E}^1 \wedge \cdots \wedge \mathcal{E}^{\alpha-1}$, for every $i \in \{1, \ldots, m\}$, state s_i^α will satisfy Properties 1 and 2 with probability at least $1 - 2e^{-k^{1/3}}$. In other words,*

$$\mathbf{Pr}(\mathcal{E}^\alpha \mid \mathcal{E}^1 \wedge \cdots \wedge \mathcal{E}^{\alpha-1}) \geq 1 - 2e^{-k^{1/3}}$$

Putting together all the previous results, we obtain the main result of this section.

Theorem 6.6. *Given an NFA N with $m \geq 1$ states over the alphabet $\{0, 1\}$, an integer $n \geq 1$ given in unary and $\delta \in (0, 1)$, there exists a randomized algorithm that receives as input N, n and δ, and returns a value R such that:*

$$\mathbf{Pr}\Big(\big|R - |\mathcal{L}_n(N)|\big| \leq \delta|\mathcal{L}_n(N)|\Big) \quad \geq \quad 1 - e^{-\tau n m},$$

where $\tau > 0$ is a fixed constant. Moreover, the algorithm runs in time $O((\frac{nm}{\delta})^c)$, where c is a fixed constant. Thus, we have that #NFA *admits an FPRAS.*

From the existence of Algorithm 4 and the form it is defined, and from the proof of Theorem 6.6, it is possible to conclude that GEN(MEM-NFA) admits a PLVUG. More precisely, we have the following result.

Corollary 6.7. *Given an NFA N with $m \geq 1$ states over the alphabet $\{0, 1\}$ and an integer $n \geq 1$ given in unary, there exists a polynomial $q(u, v)$ and a randomized algorithm $\mathcal{G}$ that receives as input N and n, and satisfies the following conditions.*

(1) If $W_{\text{MEM-NFA}}((N, 0^n)) = \varnothing$, then $\mathcal{G}(N, n)$ returns $\perp$.
(2) If $W_{\text{MEM-NFA}}((N, 0^n)) \neq \varnothing$, then
   (a) $\mathcal{G}(N, n)$ returns **fail** *with a probability $p_{N,n} < \frac{1}{2}$.*
   (b) $\mathcal{G}(N, n)$ returns $w \in W_{\text{MEM-NFA}}((N, 0^n))$ with a probability $(1 - p_{N,n})/|W_{\text{MEM-NFA}}((N, 0^n))|$.
(3) The number of steps needed to compute $\mathcal{G}(N, n)$ is at most $q(m, n)$.

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. In particular, 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 also plan on developing more such optimizations so to make this FPRAS usable in practice.

ACKNOWLEDGMENTS

M. Arenas and C. Riveros was partially funded by the Fondecyt grant 1161473 and the Fondecyt grant 11150653, respectively. M. Arenas, L. A. Croquevielle and C. Riveros were partially funded by the Millennium Institute for Foundational Research on Data.

REFERENCES

[1] Serge Abiteboul, Gerome Miklau, Julia Stoyanovich, and Gerhard Weikum. 2016. Data, Responsibly (Dagstuhl Seminar 16291). *Dagstuhl Reports* 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. *Theoretical Computer Science* 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, Pierre Bourhis, Stefan Mengel, and Matthias Niewerth. 2019. Constant-Delay Enumeration for Nondeterministic Document Spanners. In *Proceedings of ICDT*.

[6] Antoine Amarilli, Florent Capelli, Mikaël Monet, and Pierre Senellart. 2018. Connecting Knowledge Compilation Classes and Width Parameters. *CoRR* abs/1811.02944 (2018).

[7] Renzo Angles, Marcelo Arenas, Pablo Barceló, Aidan Hogan, Juan Reutter, and Domagoj Vrgoč. 2017. Foundations of modern query languages for graph databases. *ACM Computing Surveys (CSUR)* 50, 5 (2017), 68.

[8] 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.

[9] Marcelo Arenas, Martin Muñoz, and Cristian Riveros. 2017. Descriptive Complexity for counting complexity classes. In *32nd Annual ACM/IEEE Symposium on Logic in Computer Science, LICS 2017, Reykjavik, Iceland, June 20-23, 2017*. 1–12.

[10] Guillaume Bagan. 2006. MSO queries on tree decomposable structures are computable with linear delay. In *Proceedings of CSL*. 167–181.

[11] Guillaume Bagan, Arnaud Durand, and Etienne Grandjean. 2007. On acyclic conjunctive queries and constant delay enumeration. In *Proceedings of CSL*. 208–222.

[12] Randal E Bryant. 1992. Symbolic Boolean manipulation with ordered binary-decision diagrams. *ACM Computing Surveys (CSUR)* 24, 3 (1992), 293–318.

[13] Bruno Courcelle. 2009. Linear delay enumeration and monadic second-order logic. *Discrete Applied Mathematics* 157, 12 (2009), 2675–2700.

[14] 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.

[15] Ronald Fagin, Benny Kimelfeld, Frederick Reiss, and Stijn Vansummeren. 2015. Document spanners: A formal approach to information extraction. *J. ACM* 62, 2 (2015), 12.

[16] 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).

[17] Dominik D. Freydenberger. 2017. A Logic for Document Spanners. In *Proceedings of ICDT*. 13:1–13:18.

[18] Dominik D Freydenberger, Benny Kimelfeld, and Liat Peterfreund. 2018. Joining extractions of regular expressions. In *Proceedings of PODS*. 137–149.

[19] 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.

[20] 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.

[21] Wassily Hoeffding. 1963. Probability inequalities for sums of bounded random variables. *J. Amer. Statist. Assoc.* 58, 301 (1963), 13–30.

[22] 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.

[23] David S Johnson, Mihalis Yannakakis, and Christos H Papadimitriou. 1988. On generating all maximal independent sets. *Inform. Process. Lett.* 27, 3 (1988), 119–123.

[24] Sampath Kannan, Z. Sweedyk, and Stephen R. Mahaney. 1995. Counting and Random Generation of Strings in Regular Languages. In *Proceedings of SODA*. 551–557.

[25] Richard M. Karp and Michael Luby. 1983. Monte-Carlo Algorithms for Enumeration and Reliability Problems. In *Proceedings of FOCS*. 56–64.

[26] Katja Losemann and Wim Martens. 2013. The complexity of regular expressions and property paths in SPARQL. *ACM Trans. Database Syst.* 38, 4 (2013), 24:1–24:39.

[27] Francisco Maturana, Cristian Riveros, and Domagoj Vrgoc. 2018. Document Spanners for Extracting Incomplete Information: Expressiveness and Complexity. In *Proceedings of PODS*. ACM, 125–136.

[28] Klaus Reinhardt and Eric Allender. 2000. Making Nondeterminism Unambiguous. *SIAM J. Comput.* 29, 4 (2000), 1118–1131.

[29] Sanjeev Saluja, KV Subrahmanyam, and Madhukar N Thakur. 1995. Descriptive complexity of #P functions. *J. Comput. System Sci.* 50, 3 (1995), 493–505.

[30] Johannes Schmidt. 2009. Enumeration: Algorithms and complexity. *Preprint, available at https://www.thi.uni-hannover.de/fileadmin/forschung/arbeiten/schmidt-da.pdf* (2009).

[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.