Efficient Logspace Classes for Enumeration, Counting, and Uniform Generation

Marcelo Arenas
PUC & IMFD Chile

Luis Alberto Croquevielle
PUC & IMFD Chile

Rajesh Jayaram
Carnegie Mellon University

Cristian Riveros
PUC & IMFD Chile

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 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 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 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.

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 [30]. 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, 30]; 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, 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 a 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 [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. Thus, if Σ denotes a finite alphabet, then by following [21] 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) [21] 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 nondeterministic 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 [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 obtain the first complexity class with a simple and robust 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 complete proofs of these results are given in the Appendix.

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 [21], namely, 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 $\lvert W_R(x)\rvert$

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 [22] and is defined as follows. An enumeration algorithm is of polynomial delay if there exists a polynomial p such that for every input $x \in \Sigma^*$, 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 [9, 12, 30]. 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 to consecutive outputs in a very precise sense. Several notions of constant-delay enumeration has 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 [9, 12, 30].

As it is standard in the literature [30], 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 [30] in two aspects. First, as it was previously mention 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 [9, 12] 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 [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).

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) [21] 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 $\delta \in (0,1)$, 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 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 $W_R(x) = \emptyset$, 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 $W_R(x) \neq \emptyset$, 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 $W_R(x) = \emptyset$, 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 $W_R(x) \neq \emptyset$, 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 element $y \in W_R(x)$ is the same. Finally, we have that $\mathcal{G}(x)$ can be computed in polynomial time in the size of 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 [21]. 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 [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 $[\varphi(x) - \delta, \varphi(x) + \delta]$ for a given error $\delta \in (0,1)$, where φ is defined as in the notion of PLVUG.

3 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. 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 a problem is the satisfiability of a propositional formula in DNF. As a relation, this problem can be represented as follows:

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

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 the problems of 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 [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:

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

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, 7, 19, 25], 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 [8, 28], 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 [21], 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 [21], 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 [21]. 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 [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 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 [31] and UL [27]) 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 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.

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 [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 $d = a_1 \dots a_n$ 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 [15] to state our main results. Given the lack of space, we only recall the main definitions (see [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 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: $q_0 \xrightarrow{X_1} p_0 \xrightarrow{a_1} q_1 \xrightarrow{X_2} p_1 \xrightarrow{a_2} \dots \xrightarrow{a_n} q_n \xrightarrow{X_{n+1}} p_n$ 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 [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 [15] 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 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. $\text{ENUM}(\text{EVAL-eVA})$ *can be enumerated with polynomial delay,* $\text{COUNT}(\text{EVAL-eVA})$ *admits an FPRAS, and* $\text{GEN}(\text{EVAL-eVA})$ *admits a PLVUG.*

In [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 [17], it was given a polynomial-delay enumeration algorithm for $[\![\mathcal{A}]\!](d)$. Thus, only the results about $\text{COUNT}(\text{EVAL-eVA})$ and $\text{GEN}(\text{EVAL-eVA})$ are new.

Regarding efficient enumeration and exact counting, a constant-delay algorithm with polynomial preprocessing was given in in [15] 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 witness 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 [15] for the unambiguous case.

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

Notice that this result give a constant-delay algorithm with polynomial preprocessing for the class of unambiguous functional eVA. Instead, the algorithm in [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, nodes 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 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 $|\pi| = n$. 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 (that is, 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 [11]. 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 that:

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 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. $\text{var}(u) = \perp$) 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 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 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 [13]. 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 $f : \Sigma^* \to$

[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 [6, 7, 25].

Σ^*, 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 restricted, 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:*

- *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 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 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 5.3. 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 [21], and then using a strategy similar to the one described in [21]. 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 NL-TRANSDUCERS: APPROXIMATE COUNTING AND UNIFORM GENERATION

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 [21]. 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 [29]. 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 6.7. 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 6.8.

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 $\{s_{start}, s_{final}\} \cup \left(\bigcup_{t=1}^{n} \{s_1^t, s_2^t, \ldots, s_m^t\}\right)$.

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 *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 $U(s_{start}) = \emptyset$. 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 (1-indexed) 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.

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 $|\mathcal{L}(N)|$ is $R(s_{final})$. We summarize this algorithmic template below in Algorithm 1.

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 $R(s_{final})$.

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 [21], 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 $W^0 = \emptyset$.

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 $\mu > 0$. 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 $|U(s_j^\alpha)|$.

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

(1) If $W^j = \varnothing$, then with probability φ return w^j, otherwise return **fail**.
(2) Else, partition W^j into two sets, those with last bit equal to 0, call this W_0^j, and those with last bit equal to 1, call this W_1^j.
(3) Then choose partition $b \in \{0,1\}$ with probability $p_b = \frac{|W_b^j|}{|W^j|}$, and set $W^{j-1} = \{x \mid x \circ b \in W^j \text{ and } |x| \geq 1\}$, and $w^{j-1} = b \circ w^j$.
(4) Return **SampleTemplate**$(W^{j-1}, w^{j-1}, \frac{\varphi}{p_b})$.

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 could write $W^j_{w^j}$ instead of W^j, but for notational simplicity we omit the subscript, and it is then understand 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:

$$\mathbf{Pr}(w^0 = x) = \frac{|W^{\alpha-1}|}{|W^\alpha|} \cdot \frac{|W^{\alpha-2}|}{|W^{\alpha-1}|} \cdot \frac{|W^{\alpha-3}|}{|W^{\alpha-2}|} \cdot \cdots \cdot \frac{|W^1|}{|W^2|} \cdot \frac{1}{|W^1|} = \frac{1}{|W^\alpha|}.$$

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

Fact 1. *If $0 < \varphi_0 \leq \frac{1}{|W^\alpha|}$ and $w^0 \neq$ **fail** is the output of Algorithm 2, then for every $x \in W^\alpha$, it holds that $\mathbf{Pr}(w^0 = x) = \varphi_0$. 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:

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

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 $X(s_i^\alpha)$ for a fixed state s_i^α is given in Algorithm 3, and our full FPRAS is given in Algorithm 4.

Base Case: For every state s_i^α such that $|U(s_i^\alpha)| \leq k$, 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 the $\{x \circ 0 \mid x \in Y_0\} \cup \{x \circ 1 \mid x \in Y_1\}$, 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_i^α that is not exactly handled, and for which we have not computed $X(s_i^\alpha)$, $R(s_i^\alpha)$, but such that we have computed $X(s_j^t)$, $R(s_j^t)$ for $j = 1, 2, \ldots, m$ and $t = 1, 2, \ldots, \alpha - 1$. To build the set of samples $X(s_i^\alpha)$, 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 φ return w as the sample. Else, with probability $1 - \varphi$, return **fail**.
(3) Else, set $T_0 = \{s_j^{r-1} \in N_{unroll} \mid s_j^{r-1} \xrightarrow{0} s_i^r \text{ for some } s_i^r \in T\}$, and $T_1 = \{s_j^{r-1} \in N_{unroll} \mid s_j^{r-1} \xrightarrow{1} s_i^r \text{ for some } s_i^r \in T\}$ (note $T_0 \cap T_1$ may be non-empty). Then
  (a) For $q \in \{0,1\}$, compute
  $$\widetilde{W}_q = \sum_{s \in T_q} R(s) \cdot \frac{\left|X(s) \setminus \left(\bigcup_{s' \in T_q : s' \prec s} U(s')\right)\right|}{|X(s)|}$$
  (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**$(T_b, b \circ w, \frac{\varphi}{p_b})$

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

(1) If $n \leq 12$, then return $|\{x \in \{0,1\}^n \mid x \in \mathcal{L}(N)\}|$ (notice that 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 $\alpha = 1, 2, \ldots, n$ and for $i = 1, 2, \ldots, m$:
   (a) For $b \in \{0,1\}$, let $T_b(s_i^\alpha) = \{s \in N_{unroll} \mid \text{there is an edge } s \xrightarrow{b} s_i^\alpha \text{ in } N_{unroll}\}$. Let $T(s_i^\alpha) = T_0(s_i^\alpha) \cup T_1(s_i^\alpha)$, and for $b \in \{0,1\}$, assume that $T_b(s_i^\alpha) = \{v_1^b, \ldots, v_{r_b}^b\}$ where $r_b = |T_b(s_i^\alpha)|$.
   (b) If $T(s_i^\alpha) = \{s_{start}\}$ (meaning if $\alpha = 1$), set $X(s_i^\alpha) = \{b \in \{0,1\} \mid s_{start} \xrightarrow{b} s_i^\alpha \text{ is an edge in } N_{unroll}\}$. Moreover, set $R(s_i^\alpha) = |X(s_i^\alpha)|$, and declare the state s_i^α to be exactly handled.
   (c) Else, if s is exactly-handled for all $s \in T(s_i^\alpha)$, set
   $$R(s_i^\alpha) \quad = \quad \left(\sum_{j=1}^{r_0} \left| X(v_j^0) \setminus \bigcup_{t=1}^{j-1} X(v_t^0) \right| \right) + \left(\sum_{j=1}^{r_1} \left| X(v_j^1) \setminus \bigcup_{t=1}^{j-1} X(v_t^1) \right| \right),$$
   and then if $R(s_i^\alpha) \leq k$, declare s_i^α to be exactly handled, and set
   $$X(s_i^\alpha) \quad = \quad \left(\bigcup_{t=1}^{r_0} \{x \circ 0 \mid x \in X(v_t^0)\} \right) \bigcup \left(\bigcup_{t=1}^{r_1} \{x \circ 1 \mid x \in X(v_t^1)\} \right).$$
   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 $\alpha = 1, 2, \ldots, n$ and for $i = 1, 2, \ldots, m$:
   (a) If s_i^α is exactly handled, then $R(s_i^\alpha)$ and $X(s_i^\alpha)$ are already computed. Otherwise, for $b \in \{0,1\}$ set
   $$\widetilde{W}_b(s_i^\alpha) \quad = \quad \sum_{s \in T_b(s_i^\alpha)} R(s) \cdot \frac{\left| X(s) \setminus \left(\bigcup_{s' \in T_b(s_i^\alpha) : s' \prec s} U(s') \right) \right|}{|X(s)|}$$
   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 $|\mathcal{L}_n(N)|$.

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.

By invoking the recursive 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

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

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: To summarize out algorithm, 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 formally in Algorithm 4.

6.4 The Analysis of the Algotihm

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 [21], 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* **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, the algorithm 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 union bound and condition on 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:

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

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 $\left|X(s_i^\alpha) \setminus \left(\bigcup_{j \in L} U(s_j^\alpha)\right)\right|/|X(s_i^\alpha)|$ 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 $k = \lceil (\frac{mm}{\delta})^{64} \rceil$. 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. Note that since $k = n^c$, and since $|U(s^r)| \leq 2^r$ just by the bound on the number of bit-strings of length r, it follows that all layers r up to $r = c\log(n)$ will be exactly handled. We will now need the well-known Hoeffding inequality:

Proposition 6.5 (Hoeffding inequality [20]). *Let $X_1, \ldots, X_n$ be independent random variables supported on $[0,1]$. Let $S = \frac{1}{n}\sum_{i=1}^{n} X_n$. Then for every $t > 0$, we have that $\mathbf{Pr}(|S - \mathbb{E}[S]| \geq t) \leq 2e^{-2nt^2}$.*

The following Lemma 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 6.6. *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 6.7. *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:*

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

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 6.7, it is possible to conclude that GEN(MEM-NFA) admits a PLVUG. More precisely, we have the following result.

Corollary 6.8. *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 $W_{\text{MEM-NFA}}((N, 0^n)) = \emptyset$, then $\mathcal{G}(N,n)$ returns $\perp$.
(2) If $W_{\text{MEM-NFA}}((N, 0^n)) \neq \emptyset$, 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.

REFERENCES

[1] S. Abiteboul, G. Miklau, J. Stoyanovich, and G. Weikum. Data, responsibly (dagstuhl seminar 16291). *Dagstuhl Reports*, 6(7):42–71, 2016.
[2] A. V. Aho and J. E. Hopcroft. *The design and analysis of computer algorithms*. Pearson Education India, 1974.
[3] C. Álvarez and B. Jenner. A very hard log-space counting class. *Theoretical Computer Science*, 107(1):3–30, 1993.
[4] A. Amarilli, P. Bourhis, L. Jachiet, and S. Mengel. A circuit-based approach to efficient enumeration. In *Proceedings of ICALP*, pages 111:1–111:15, 2017.
[5] A. Amarilli, F. Capelli, M. Monet, and P. Senellart. Connecting knowledge compilation classes and width parameters. *CoRR*, abs/1811.02944, 2018.
[6] R. Angles, M. Arenas, P. Barceló, A. Hogan, J. Reutter, and D. Vrgoč. Foundations of modern query languages for graph databases. *ACM Computing Surveys (CSUR)*, 50(5):68, 2017.
[7] M. Arenas, S. Conca, and J. Pérez. Counting beyond a yottabyte, or how SPARQL 1.1 property paths will prevent adoption of the standard. In *Proceedings of WWW*, pages 629–638, 2012.
[8] M. Arenas, M. Muñoz, and C. Riveros. 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*, pages 1–12, 2017.
[9] G. Bagan. MSO queries on tree decomposable structures are computable with linear delay. In *Proceedings of CSL*, pages 167–181, 2006.
[10] G. Bagan, A. Durand, and E. Grandjean. On acyclic conjunctive queries and constant delay enumeration. In *Proceedings of CSL*, pages 208–222, 2007.
[11] R. E. Bryant. Symbolic boolean manipulation with ordered binary-decision diagrams. *ACM Computing Surveys (CSUR)*, 24(3):293–318, 1992.
[12] B. Courcelle. Linear delay enumeration and monadic second-order logic. *Discrete Applied Mathematics*, 157(12):2675–2700, 2009.
[13] C. Daskalakis, P. W. Goldberg, and C. H. Papadimitriou. The complexity of computing a nash equilibrium. *SIAM J. Comput.*, 39(1):195–259, 2009.
[14] R. Fagin, B. Kimelfeld, F. Reiss, and S. Vansummeren. Document spanners: A formal approach to information extraction. *Journal of the ACM*, 62(2):12, 2015.
[15] F. Florenzano, C. Riveros, M. Ugarte, S. Vansummeren, and D. Vrgoc. Constant delay algorithms for regular document spanners. *arXiv preprint arXiv:1803.05277*, 2018.
[16] D. D. Freydenberger. A logic for document spanners. In *Proceedings of ICDT*, pages 13:1–13:18, 2017.
[17] D. D. Freydenberger, B. Kimelfeld, and L. Peterfreund. Joining extractions of regular expressions. In *Proceedings of PODS*, pages 137–149, 2018.
[18] V. Gore, M. Jerrum, S. Kannan, Z. Sweedyk, and S. R. Mahaney. A quasi-polynomial-time algorithm for sampling words from a context-free language. *Inf. Comput.*, 134(1):59–74, 1997.
[19] L. A. Hemaspaandra and H. Vollmer. The satanic notations: counting classes beyond #p and other definitional adventures. *SIGACT News*, 26(1):2–13, 1995.
[20] W. Hoeffding. Probability inequalities for sums of bounded random variables. *Journal of the American Statistical Association*, 58(301):13–30, 1963.
[21] M. R. Jerrum, L. G. Valiant, and V. V. Vazirani. Random generation of combinatorial structures from a uniform distribution. *Theor. Comput. Sci.*, 43:169–188, 1986.
[22] D. S. Johnson, M. Yannakakis, and C. H. Papadimitriou. On generating all maximal independent sets. *Information Processing Letters*, 27(3):119–123, 1988.
[23] S. Kannan, Z. Sweedyk, and S. R. Mahaney. Counting and random generation of strings in regular languages. In *Proceedings of SODA*, pages 551–557, 1995.
[24] R. M. Karp and M. Luby. Monte-carlo algorithms for enumeration and reliability problems. In *Proceedings of FOCS*, pages 56–64, 1983.
[25] K. Losemann and W. Martens. The complexity of regular expressions and property paths in SPARQL. *ACM Trans. Database Syst.*, 38(4):24:1–24:39, 2013.
[26] F. Maturana, C. Riveros, and D. Vrgoc. Document spanners for extracting incomplete information: Expressiveness and complexity. In *Proceedings of PODS*, pages 125–136. ACM, 2018.
[27] K. Reinhardt and E. Allender. Making nondeterminism unambiguous. *SIAM J. Comput.*, 29(4):1118–1131, 2000.
[28] S. Saluja, K. Subrahmanyam, and M. N. Thakur. Descriptive complexity of# p functions. *Journal of Computer and System Sciences*, 50(3):493–505, 1995.
[29] J. Schmidt. Enumeration: Algorithms and complexity. *Preprint, available at https://www.thi.uni-hannover.de/fileadmin/forschung/arbeiten/schmidt-da.pdf*, 2009.
[30] L. Segoufin. Enumerating with constant delay the answers to a query. In *Proceedings of ICDT*, pages 10–20, 2013.
[31] L. G. Valiant. Relative complexity of checking and evaluating. *Inf. Process. Lett.*, 5(1):20–23, 1976.

A PROOFS FROM SECTION 5

A.1 Proof of Proposition 5.1

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 (resp. polynomial) delay, so there is an algorithm $\mathcal{E}$ that enumerates $W_S(f(x))$ with constant (resp. 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 witnesses in $W_S(f(x))$, that is, it enumerates all witnesses 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 on $|x|$. The enumeration phase is the same as for $\mathcal{E}(f(x))$, so it has constant (resp. polynomial) delay. We conclude that ENUM(R) can be solved with constant (resp. polynomial) delay.

Now, suppose there exists a polynomial-time algorithm $\mathcal{A}$ for COUNT(S), and let q be the polynomial that characterizes its complexity. Now, 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 on $|x|$. 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 witness from $W_S(f(x))$, that is, a witness 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 on $|x|$. We conclude that GEN(R) has a polynomial-time randomized algorithm. The proof for the case of an PLVUG is completely analogous.

A.2 Proof of Proposition 5.2

We will prove the result for the case of RelationUL and MEM-UFA. The other case is completely analogous. The main content of the proof is given by the following Lemma.

Lemma A.1. *Let R be a relation in* RelationUL *defined on alphabet Σ and $x \in \Sigma^*$. Then there exists an unambiguous NFA N_x such that $W_R(x) = \mathcal{L}(N_x)$. Also, N_x can be constructed in poly$(|x|)$ time.*

Proof. 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 suppose that M has only one accepting state, so it can be written as a tuple $M = (Q, \Gamma, b, \Sigma, \delta, q_0, \{q_F\})$. If it has more than one accepting state, say a set F of accepting states, 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 $n = |x|$ and let $f(n) = O(\log(n))$ be the function that bounds the amount of work tape that can be used. 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 $c_I = (q_0, 1, 1, \varepsilon)$, 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. 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 suppose the accepting configuration to be unique, by changing M so that it runs for a little longer in order to reach it. If C_x is the set of possible configuration tuples then we have that

$$\begin{aligned} |C_x| &\leq |Q| \cdot n \cdot f(n) \cdot |\Gamma|^{f(n)} \\ &= |Q| \cdot n \cdot f(n) \cdot |\Gamma|^{O(\log(n))} \\ &= |Q| \cdot n \cdot f(n) \cdot O(n) \\ &= O(n^2 \log(n)), \end{aligned}$$

which is polynomial in $|x|$. We now define the NFA $N_x = (C_x, \Sigma, \Delta_x, c_I, \{c_F\})$ 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 is quick to check 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 N_x can be done in polynomial time. It only rests to show that $W_R(x) = \mathcal{L}(N_x)$ and that N_x is unambiguous.

Let $y \in W_R(x)$. That means there is an accepting execution 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 $\{w_k\}_{k=0}^{m}$ 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, $w_k = \varepsilon$.
- $y = w_0 \circ w_1 \circ \ldots \circ w_m$.

By definition, that means that y is accepted by N_x. That is, $y \in \mathcal{L}(N_x)$ and so we can conclude that $W_R(x) \subseteq \mathcal{L}(N_x)$. Since all the previous implications are clearly equivalencies, we can also conclude that $\mathcal{L}(N_x) \subseteq W_R(x)$. Hence $W_R(x) = L(N_x)$ as we wanted. 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 N_x on input y. That implies that N_x is unambiguous. Otherwise, there would be some $y \in L(N_x)$ such that two different runs of N_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 N_x is actually not an NFA (under the definition given in Section 2), 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. That ends the proof. □

So, let R be any relation in RelationUL and x some input. We know by Lemma A.1 that we can construct an unambiguous N_x in polynomial time such that $W_R(x) = \mathcal{L}(N_x)$. Now, since R is a p-relation, there exists a polynomial q such that $|y| = q(|x|)$ for all $y \in W_R(x) = \mathcal{L}(N_x)$. That means that all words accepted by N_x have the same length $p(|x|)$. We can conclude that $W_R(x) = W_{\text{MEM-UFA}}\left(\left(N_x, 0^{p(|x|)}\right)\right)$. Since this works for any $R \in$ RelationUL and any input x, we have that MEM-UFA is complete for RelationUL.

A.3 Proof of Proposition 5.3

A.3.1 Proof that ENUM(MEM-UFA) *can be solved with constant delay.* Our aim here is to prove the first of the three results included in Proposition 5.3. Let $(N, 0^k)$ be an input. Without loss of generality, we can assume that N has a unique final state. If it had more than one, say a set F of final states, we can create a new state q_F, set it as the unique final state, and add ε-transitions from all states in F to q_F. Afterwards, we can get rid of he ε-transitions in the standard way. All of this can be done in polynomial time and preserves $\mathcal{L}(N)$ so it causes no problems. In the end, N has a very well defined form, a simple example of which is presented in Figure 1.

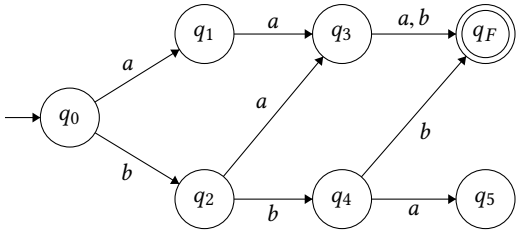

Figure 1: Unambiguous NFA N.

Notice that an NFA of this kind basically constitutes a directed acyclic graph (DAG). And we can indeed define a DAG from N and k, making explicit for each vertex the length of the words accepted until that vertex and where the edges of the graph are labeled. By doing that, for each w in $\mathcal{L}(N)$, there is a path from a "start node" of the graph to an "end node", such that the labels read along the way form w. This notions are encapsulated in Lemma A.2. For our example, the corresponding DAG is presented in Figure 2, where the start node is $(q_0, 0)$ and the end node is $(q_F, 3)$. Notice that we have omitted many nodes from it, like $(q_F, 0)$. We can do that since we only want this graph to enumerate the words accepted by N, that is, we need to enumerate all labeled paths from $(q_0, 0)$ to $(q_F, 3)$, so any vertex that does not fall in one of those paths, we do not need for the enumeration. Now, starting from $(N, 0^k)$, we can construct the DAG G in polynomial time. This is the pre-processing phase of the algorithm. Once we have G, we can start the enumeration phase.

Lemma A.2. *Let N be an unambiguous NFA with set of states Q and a unique final state, and k a natural number in unary. Then in poly$(|N|, k)$ time we can construct a directed acyclic graph $G = (V, E)$ such that all of its edges have non empty labels codified by a function $\lambda : E \to \Sigma$, and*

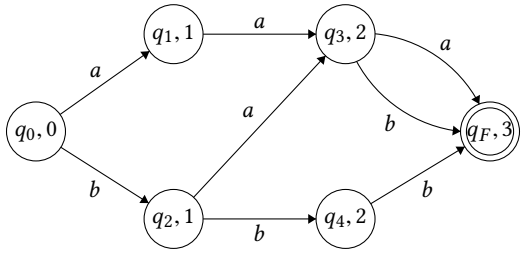

Figure 2: Graph G obtained from N.

such that G has the following properties. First, the set of vertices is $V = Q \times \{0, \ldots, n\}$. Second, there are two nodes $s_0, s_F \in V$ such that for all $w \in \mathcal{L}_k(N)$, there exists a unique path $(v_0, v_1, \ldots, v_{k-1}, v_k)$ in G where:

- $v_0 = s_0$.
- $v_k = s_F$.
- *For all $i \in \{1, \ldots, k\}$, we have that $w_i = \lambda((v_{i-1}, v_i))$.*

And finally, if $e \in E$, then e is part of a path from s_0 to s_F.

Proof. Let $N = (Q, \Sigma, \Delta, q_0, \{q_F\})$. We start by defining a directed graph $G = (V, E)$ where $V = Q \times \{0, 1, \ldots, n\}$ and the set of labeled edges E is defined as

$$E = \{((q, i), a, (q', i+1)) \mid i \in \{0, \ldots, n-1\} \text{ and } (q, a, q') \in \Delta\}.$$

Which induces the definition $\lambda(((q, i), a, (q', i+1))) = a$ for each $((q, i), a, (q', i+1)) \in E$. We also denote the labeled edges in E like $(q, i) \xrightarrow{a} (q', i+1)$. Notice that G can be constructed in time $O(k^2|N|^2)$. Set now $s_0 = (q_0, 0)$ and $s_F = (q_F, k)$ and consider the set $\mathcal{P}$ of labeled paths from s_0 to s_F in G. We claim that $\mathcal{L}_k(N) = \mathcal{P}$. To see that, consider the function $g : \mathcal{L}_k(N) \to \mathcal{P}$ defined in the following way. Let $y = y_1 \ldots y_n \in \mathcal{L}_k(N)$. Then, since N is unambiguous, there exists a unique accepting run of N on input k given by

$$p_0 \xrightarrow{y_1} p_1 \xrightarrow{y_1} p_2 \xrightarrow{y_2} \ldots \xrightarrow{y_k} p_k$$

where $p_0 = q_0, p_k = q_F$ and $(p_{i-1}, y_i, p_i) \in \Delta$ for all $i \in \{1, \ldots, k\}$. So, we define $g(y)$ as the path $s_0 = (p_0, 0) \xrightarrow{y_1} (p_1, 1) \xrightarrow{y_2} \ldots \xrightarrow{y_k} (q_F, k) = s_F$. By definition of E, it is easy to see that g is a bijective function, since the accepting run of y is unique, which proves that $\mathcal{L}_k(N) = \mathcal{P}$. Now, notice that G is a directed acyclic graph. Indeed, suppose it had a cycle $v_0 \to v_1 \to \ldots \to v_m \to v_{m+1}$ with $v_{m+1} = v_0$. By definition of E, that would imply that for some number i and states p_k in Q we would have $v_k = (p_k, i+k)$ for all $k \in \{0, \ldots, m+1\}$. But that would mean that $v_0 = (p_0, 0)$ and also $v_0 = v_{m+1} = (p_{m+1}, m+1)$ which cannot be true.

All requirements of the lemma are fulfilled, save for the last one. The only thing left to do then, is to remove from G all edges that are not in some path from s_0 to s_F. We can do that by going through all edges, and for each one verifying whether it is part of some path from s_0 to s_F. It it is not, we remove it. That verification is easy to do in polynomial time, and the number of edges is polynomial, so the total time of the algorithm is still $\text{poly}(|N|, k)$. □

The idea for the constant delay algorithm is the following. Consider our example. If a vertex has two outgoing edges, then we can specify some order between them (for example, the order could be inherited from an order on the labels, so that the edge labeled by an a comes before the edge labeled by a b). Now, we start from $(q_0, 0)$ and choose the first of its outgoing edges, so we read an a and move to $(q_1, 1)$. Once there, we do the same (there is only one edge this time), so we read an a and move to $(q_3, 2)$. We once again do the same, and choose the first outgoing edge, so we read an a and move to $(q_F, 3)$. Since we are now at the end node, we output the concatenation of the labels read along the path, that is *aaa*.

That clearly took only linear time, but now comes the question of how to output the next word, since we want to keep the delay linear and we do not want any words repeated. In order to do this in an orderly fashion, we store all the points where we made a decision about which edge to take. In this case, we would store the transitions $(q_0, 0) \xrightarrow{a} (q_1, 1)$ and $(q_3, 2) \xrightarrow{a} (q_F, 3)$. It is not necessary to store the transition $(q_1, 1) \xrightarrow{a} (q_3, 2)$, since it was an only choice when we were in $(q_1, 1)$. Now, we can use this information. We start from $(q_0, 0)$ and use the stored transitions to recreate the same path as before, until we come to the last one, $(q_3, 2) \xrightarrow{a} (q_F, 3)$ in this case, where we change our choice to the other edge $(q_3, 2) \xrightarrow{b} (q_F, 3)$.

Since we have arrived at the end node, we output *aab* which is the concatenation of the labels read along the way. Notice that it again took linear time to output the word. And also, since we have now used all edges from $(q_3, 2)$, we remove that last transition, leaving only $(q_0, 0) \xrightarrow{a} (q_1, 1)$ stored. And now, to proceed with the next word, we once again start from $(q_0, 0)$ and use the stored transitions to recreate

our path, until we come to the last one. In this case, the last stored transition comes right away, so we change our choice to $(q_0, 0) \xrightarrow{b} (q_2, 1)$, and from that moment on we proceed by choosing at each step the edge that comes first, and storing our decisions for later. This process of storing decisions, outputting words, and using stored transitions to recreate the previous path up until some point, goes on until the moment when, after outputting some word, we remove the decision $(q_0, 0) \xrightarrow{b} (q_2, 1)$ from our storage. This idea is formalized below, but before, we introduce some notation.

Let $G = (V, E)$ be a DAG with the properties stated in Lemma A.2. For some $q \in V$ we define the set $V(q) = \{(a, q') \mid (q, a, q') \in E\}$ and we fix some total order on that set. Based on that, we define $\min(V(q))$ as the minimum element of $V(q)$ according to that order, and analogously for $\max(V(q))$. Also, we define a function succ that returns the successor of an element in $V(q)$, according to the total order defined. We also use a `list` structure where we store elements of the form $(q, (a, q'))$ where $(a, q') \in V(q)$. That structure supports the following operations.

- `append`$((q, (a, q')))$: it adds element $(q, (a, q'))$ at the end of the list.
- `pop`$()$: it removes the last element of the list.
- `next`(q): if there is an element $(q, (a, q'))$ on the list, it returns (a, q'). Otherwise, it returns $(\perp, \perp)$.
- `last`$()$: it returns the last element of the list, and $(\perp, \perp)$ if the list is empty.

Enumeration Algorithm : `Enumerate`$(N, 0^k)$:

(1) Construct $G = (V, E)$ using Lemma A.2.
(2) Initialize an empty list `list`.
(3) Initialize an empty word $w = \varepsilon$.
(4) Initialize a node variable $n = (q_0, 0)$.
(5) Initialize an integer $j = 0$.
(6) While $j < k$:
  (a) Get $(a, s') =$ `list.next`(n)
  (b) If $s' \neq \perp$:
    - Update $n = s'$.
    - Update $w = w \circ a$.
  (c) Else:
    - Get $(a, s') = \min(V(s))$.
    - Update $n = s'$.
    - Update $w = w \circ a$.
    - If $|V(n)| > 1$ do `list.append`$(s, (a, s'))$.
  (d) Update $j = j + 1$.
(7) **Output:** w
(8) Get $(s, (a, s')) =$ `list.last()`.
(9) If $s = \perp$, **STOP**.
(10) While $(a, s') = \max(V(s))$:
  (a) Do `list.pop()`
  (b) Get $(s, (a, s')) =$ `list.last()`.
  (c) If $s = \perp$, **STOP**.
(11) Do `list.append`$(s, \text{succ}((a, s')))$.
(12) Go to Step (3).

The previous algorithm works because, as we explained before, it makes sure to never repeat the same path while traversing G. And since each path in the DAG is associated to a different word (because the NFA was unambiguous), no word will be repeated. Also, the algorithm makes sure to go through all possible paths from s_0 to s_F. And since we removed from G all edges that are not part of such paths, we know the algorithm will not spend time traversing some part of the graph that will not yield an output (which would ruin the constant delay).

Notice that the precomputation phase (which basically amounts to constructing G and setting up the data structures) is polynomial in time. And notice that the enumeration phase works with constant delay. Between one output and the next, we have to go through Step (6). That means k iterations, which is no problem since the outputs are of size k. And each iteration can be done in constant time, since the functions min, max, succ and all of the `list` operations can be implemented in constant time using a RAM model. The same goes for Step (10). Since the list at most keeps k elements (it uses the elements to recreate the last path it made up until the point where it has to change it, so it does not need to store more than k elements), that Step takes at most $c \cdot k$ time where c is a constant. That means that the delay is constant, under the definition of Section 2. We conclude that ENUM(MEM-UFA) can be solved with constant delay.

A.3.2 Proof that there exists a polynomial-time algorithm for COUNT(MEM-UFA). Our aim here is to prove the second of the three results included in Proposition 5.3. Take an input $(N, 0^k)$ where N is an unambiguous NFA. Consider now a nondeterministic logspace Turing machine A that takes $(N, 0^k)$ as an input and executes the following procedure.

(1) On input $(N, 0^k)$.
(2) Nondeterministically, generate a word $w \in \Sigma^*$ such that $|w| = k$.
(3) Verify whether N accepts w.
(4) If it does, stop in an accepting state.
(5) Otherwise, stop in a non accepting state.

Written like that, it cannot be implemented by A, since generating and storing the whole of w in the work tape would require linear space. But the same idea can be implemented using only logarithmic space, by generating w and simulating the execution of A on the fly. That is, at each moment we store just one symbol from w, and the current state of N. Using the stored information, nondeterministically we simulate a valid transition of N, and store the new state in place of the old one. We also replace the old symbol from w by nondeterministically choosing the next one. In addition to this, we need to store a counter, that is increased by one each time we simulate a transition of the NFA. When the counter reaches k, if the current stored state is a final state of N, then A stops in an accepting state. Otherwise, it stops in a non accepting state. Notice that since k is given in unary, the counter only uses logarithmic space on the input (the counter uses a binary representation to store numbers).

On input $(N, 0^k)$, how many accepting runs does A have? There are a total of $|\Sigma|^k$ possible w that can be nondeterministically generated by A. But when we simulate N, only the ones in $\mathcal{L}_k(N)$ will be accepted. That means that there are only $|\mathcal{L}_k(N)|$ possible w that A can generate so that the execution will end in an accepting state. Now, when A generates a word w, it then simulates a run of N on input w. But since N is unambiguous, there is only one accepting run for each $w \in \mathcal{L}_k(N)$. We conclude that all different accepting runs of A correspond to different w generated. In other words, when running A on input $(N, 0^k)$ there will be exactly $|\mathcal{L}_k(N)|$ accepting runs. Since A is a nondeterministic logspace Turing machine, and this works for any input $(N, 0^k)$, this means that the function $f : (N, 0^k) \mapsto |\mathcal{L}_k(N)|$ belongs to the class #L, defined in [3]. As shown in [3], that means that f can be computed in polynomial time. That is, the quantity $\left|W_{\text{MEM-UFA}}((N, 0^k))\right|$ can be computed in polynomial time. We conclude that there exists a polynomial-time algorithm for COUNT(MEM-UFA).

A.3.3 Proof that there exists a polynomial-time randomized algorithm for GEN(MEM-UFA). Our aim here is to prove the third of the three results included in Proposition 5.3. Take an input $(N, 0^k)$ where N is an unambiguous NFA. The idea is basically the same as the one in [21], that is, we use the fact that the relation is self-reducible and its counting problem can be solved efficiently. Let A be the polynomial-time algorithm for COUNT(MEM-UFA), and let ψ be the function from the definition of self-reducibility (recall that MEM-UFA is a self-reducible problem, see Appendix B for more details). Now, consider the following procedure for generating an witness of $(N, 0^k)$ uniformly at random. For simplicity's sake, we will make the assumption that $\Sigma = \{0, 1\}$, but it is easy to generalize it to the case of bigger alphabets. Also, to simplify, we represent $A((N, 0^k)$ by $A(N, k)$.

(1) On input $(N, 0^k)$.
(2) Assign variables $N' \leftarrow N, k' \leftarrow k, w \leftarrow \varepsilon$.
(3) While $k' > 0$, do:
   (a) Construct $(N_0, 0^{k-1}) = \psi((N', 0^{k'}), 0)$ and $(N_1, 0^{k-1}) = \psi((N', 0^{k'}), 1)$.
   (b) Calculate $p_0 = \frac{A(N_0, k-1)}{A(N_0, k-1)+A(N_1, k-1)}$ where $A(N_i, k-1) = |W_{\text{MEM-UFA}}((N_i, 0^{k-1}))|$.
   (c) With probability p_0, assign $N' \leftarrow N_0, w \leftarrow w \circ 0$. Otherwise, assign $N' \leftarrow N_1, w \leftarrow w \circ 1$.
   (d) Assign $k' \leftarrow k' - 1$.
(4) Output w.

First, notice the running time of the previous algorithm. It will execute exactly k iterations. Also, A is a polynomial-time algorithm and ψ can be computed in polynomial time (a fact that comes from the definition of self-reducibility). So the procedure above, as a whole, runs in polynomial time on the input $(N, 0^k)$. Now, let $w = w_1 w_2 \dots w_k$ be the output of the algorithm, and let $a = a_1 a_2 \dots a_k$ be any element in $W_{\text{MEM-UFA}}((N, 0^k))$. In order to show that the procedure above is a uniform generator, we will now calculate the probability that the output

is equal to a. We use the following notation. For $b \in \{0,1\}$, $\overline{b}$ denotes $1-b$. Also, consider step (3) (c), where both N' and w are updated. If w is updated to some word y, then we denote by N_y the value of N' assigned at that step. Now, using that notation, we have that

$$\begin{aligned}
\mathbf{Pr}(w = a) &= \mathbf{Pr}(w_1 = a_1 \wedge \cdots \wedge w_k = a_k) \\
&= \mathbf{Pr}(w_k = a_k \mid w_1 = a_1 \wedge \cdots \wedge w_{k-1} = a_{k-1}) \cdot \mathbf{Pr}(w_1 = a_1 \wedge \cdots \wedge w_{k-1} = a_{k-1}) \\
&= \frac{A(N_{a_1 \ldots a_k}, 0)}{A(N_{a_1 \ldots a_k}, 0) + A(N_{a_1 \ldots \overline{a_k}}, 0)} \cdot \mathbf{Pr}(w_1 = a_1 \wedge \cdots \wedge w_{k-1} = a_{k-1}) \\
&= \frac{A(N_{a_1 \ldots a_k}, 0)}{A(N_{a_1 \ldots a_{k-1}}, 1)} \cdot \mathbf{Pr}(w_1 = a_1 \wedge \cdots \wedge w_{k-1} = a_{k-1}) \\
&= \frac{A(N_{a_1 \ldots a_k}, 0)}{A(N_{a_1 \ldots a_{k-1}}, 1)} \cdot \mathbf{Pr}(w_{k-1} = a_{k-1} \mid w_1 = a_1 \wedge \cdots \wedge w_{k-2} = a_{k-2}) \cdot \mathbf{Pr}(w_1 = a_1 \wedge \cdots \wedge w_{k-2} = a_{k-2}) \\
&= \frac{A(N_{a_1 \ldots a_k}, 0)}{A(N_{a_1 \ldots a_{k-1}}, 1)} \cdot \frac{A(N_{a_1 \ldots a_{k-1}}, 1)}{A(N_{a_1 \ldots a_{k-2}}, 2)} \cdot \mathbf{Pr}(w_1 = a_1 \wedge \cdots \wedge w_{k-2} = a_{k-2}) \\
&\vdots \\
&= \frac{A(N_{a_1 \ldots a_k}, 0)}{A(N_{a_1 \ldots a_{k-1}}, 1)} \cdot \frac{A(N_{a_1 \ldots a_{k-1}}, 1)}{A(N_{a_1 \ldots a_{k-2}}, 2)} \cdot \ldots \cdot \frac{A(N_{a_1 a_2}, k-2)}{A(N_{a_1}, k-1)} \cdot \frac{A(N_{a_1}, k-1)}{A(N, k)} \\
&= \frac{A(N_{a_1 \ldots a_k}, 0)}{A(N, k)}.
\end{aligned}$$

Since a is in $W_{\text{MEM-UFA}}((N, 0^k))$, by self-reducibility we know that $A(N_{a_1 \ldots a_k}, 0) = 1$. Otherwise, we would have $A(N_{a_1 \ldots a_k}, 0) = 0$. So in the end, we get

$$\mathbf{Pr}(w = a) = \begin{cases} \dfrac{1}{|W_{\text{MEM-UFA}}((N, 0^k))|} & \text{if } a \in W_{\text{MEM-UFA}}((N, 0^k)), \\ 0 & \text{otherwise.} \end{cases}$$

That is, the procedure is a uniform generator. We conclude that there exists a polynomial-time randomized algorithm for GEN(MEM-UFA).

B A PROOF THAT MEM-NFA AND MEM-UFA ARE SELF-REDUCIBLE

We will 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 fringe cases. First, of all, 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, 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 (for example, when it has more than one final state). In those cases, we consider that $(N, 0^k)$, for any k, does not have any witnesses. Also, and this gets more technical, we consider that any input that has an invalid encoding does not have any witnesses 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 correction of the encoding can be checked in polynomial time (this is not a strong requirement as any reasonable encoding will allow for it). And it is important to have in mind that for some technical concepts like self-reduciblity, the encoding of the problem is critical.

We use the notion of self-reducibility stated in [29], because we want to utilize a result from that article which is proved under that specific notion of self-reducibility. We include the definition here, adapted to our situation, since [29] uses a slightly different framework to define an enumeration problem. We say a relation $R \subseteq \Sigma^* \times \Sigma^*$ is self reducible if there exist polynomial-time computable functions $\psi : \Sigma^* \times \Sigma^* \to \Sigma^*$, $\sigma : \Sigma^* \to \mathbb{N}$ and $\ell : \Sigma^* \to \mathbb{N}$ such that for every $x, y, w \in \Sigma^*$:

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

(2) if $\ell(x) = 0$, it can be tested in polynomial time in $|x|$, whether the empty string is a witness 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) $W_R(x) = \bigcup_{w \in \Sigma^{\sigma(x)}} \{w \circ y \mid y \in W_R(\psi(x, w))\}$.

The last condition can be equivalently stated in the following way, which is how we will use it:

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

As we already stated, the empty string is a witness only when the input is correctly encoded and the initial and final states of the automaton coincide. So condition (2) from the previous definition is satisfied regardless of our definition of ℓ. We will focus from now on on the other six conditions. Let $\mathcal{N} = \{N \mid N \text{ is an NFA with a unique final state and no } \varepsilon\text{-transitions}\}$. 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 others, the input is not correct, so the witness set 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 witness 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 witnesses have positive length, we can create another input that has the same witnesses, 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 (7). We now proceed in that direction.

Let $N = (Q, \Sigma, \delta, q_0, \{q_F\})$ 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 $x = (N, 0^k)$ which is incorrectly encoded or where N is not in $\mathcal{N}$. Then, it has no witnesses and it is enough to set $\psi(x, w) = x$ for all $w \in \Sigma^*$ (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 (7) amounts to $\forall y \in \Sigma^* : (x, y) \in \text{MEM-NFA} \iff (x, y) \in \text{MEM-NFA}$, 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 $w \in \Sigma^*$ (which is computable in polynomial time and means that condition (5) is trivially true), and since $\ell(x) = \sigma(x) = 0$, same as before, condition (6) is true and condition (7) again amounts to $\forall y \in \Sigma^* : (x, y) \in \text{MEM-NFA} \iff (x, y) \in \text{MEM-NFA}$, which is obviously met. 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 any $w \in \Sigma$ we set $\psi((N, 0^k), w) = (N', 0^{k-1})$ where N' is defined as follows. Let Q_w be the set

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

Basically, 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 $N' = (Q', \Sigma, \delta, q_0', \{q_F'\})$ where q_0' is a new state not contained in Q, and:

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

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 so 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.

Notice something 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. That means 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, that involved at least one state from Q_w. So, all in all, we have not added any new transitions, just simulated the ones were 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 (very 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 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:

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

Finally, we turn to condition (7). Let $y = y_1 y_2 \dots y_m \in \Sigma^*$. Since $\sigma(x) = 1$, condition (7) amounts to

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

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

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

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

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

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 $i < m$, and consider the case for $i + 1$. By the induction hypothesis, we know there is a valid run ρ_i such that

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

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 $s_{i+1} = p_{i+1}$, 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 $s_{i+1} = q'_0$, 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 $s_{i+1} = p_{i+1}$, 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 $s_{i+1} = q'_0$, 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

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

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

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

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 $p_k \notin Q_{y_1}$, 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 $p_k \in Q_{y_1}$, 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 $((N, 0^k), y) \in \text{MEM-NFA} \implies ((N', 0^{k-1}), y_2 \dots y_m) \in \text{MEM-NFA}$. The proof for the other direction is analogous.

The case for MEM-UFA is actually the same proof. That is, the same definitions of ℓ, σ and ψ work for that MEM-UFA. 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.

C PROOFS FROM SECTION 6

C.1 Proof of Proposition 6.2

First note that $U(s) = \varnothing$ for a state s if and only if there is no path from the start vertex $s_{start} \in N_{unroll}$ to s. Note that we remove all such unreachable states in Algorithm 4 prior to runnin our sampler, which has no effect on the claim of the proposition since for such a s we have $U(s) = \varnothing$ anyway. Now for the base case $t = \alpha$, this states that $U(s_j^\alpha) = W^\alpha$ since $T^\alpha = \{s_i^\alpha\}$, which is true since $w^\alpha = \varepsilon$. In addition, we use the induction invariant that for every t, any path from s_{start} to s_i^α labeled by a string $x \in U(s_i^\alpha)$ with suffix w^t must go through a state in T^t. For $t = \alpha$, the invariant holds because every path from s_{start} to s_i^α must clearly go through $T^\alpha = \{s_i^\alpha\}$.

Now assume for $t \leq \alpha$ that $(\bigcup_{s \in T^t} U(s)) = W^t$, that is, that $(\bigcup_{s \in T^t} U(s))$ is the set of strings x such that $x \circ w^t \in U(s_j^\alpha)$, and assume the invariant holds for this t. We show the result for $t - 1$. Let $x \in U(s_i^\alpha)$ be a string with suffix $w^{t-1} = w^{t-1}[1] \circ w^t$, and let π be any path from s_{start} to s_i^α labeled by x. Since x has the suffix w^t, by induction hypothesis, π must go through some $s \in T^t$. In order for π to get to s, it must go through some state s' in layer $(t-1)$ via the transition $s' \xrightarrow{w^{t-1}[1]} s$. Then by construction, s' in included in T^{t-1}, which proves the induction invariant for $t - 1$.

Now let $x \in W^{t-1}$. By definition of W^{t-1}, we know that $x \circ w^{t-1} \in U(s_j^\alpha)$. Then by the invariant, any path labelled by $x \circ w^{t-1}$ from s_{start} to s_i^α must go through some state s of T^{t-1}. Therefore, we conclude that $x \in (\bigcup_{s \in T^{t-1}} U(s))$. Conversely, assume that $x \in (\bigcup_{s \in T^{t-1}} U(s))$, and let $s \in T^{t-1}$ be such that $x \in U(s)$. Then let π be any path from s_{start} to s labeled by x. By definition, there is a transition $s \xrightarrow{w^{t-1}[1]} s'$ for some $s' \in T^t$, so $x \circ w^{t-1}[1] \in (\bigcup_{s \in T^t} U(s))$. Then by induction hypothesis, it holds that $(x \circ w^{t-1}[1]) \circ w^t \in U(s_j^\alpha)$. Hence, given that $w^{t-1} = w^{t-1}[1] \circ w^t$, we conclude that $x \circ w^{t-1} \in U(s_j^\alpha)$ and, therefore, $x \in W^{t-1}$, which completes the proof.

C.2 Proof of Proposition 6.3

First we show that every recursive call to **Sample** is such that $\varphi \in (0, 1)$. Since $\varphi_0 = \frac{e^{-4}}{R(s_i^\alpha)} > 0$ and with each call φ increases (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 is increasing after each recursive call, it suffices to show this for the final value of φ. Notice that at the $(\alpha - j)$-th call, for all $1 \leq j \leq \alpha$, we have that φ is divided by a factor of $\frac{\widetilde{W}^{\alpha-j}_{w[\alpha-j]}}{\widetilde{W}_0^{\alpha-j} + \widetilde{W}_1^{\alpha-j}}$ (recall that $w[i]$ is the i-th bit of w). So in the final call, φ has the value:

$$\begin{aligned} \varphi &= \left(\prod_{j=0}^{\alpha-1} \frac{\widetilde{W}^{\alpha-j}_{w[\alpha-j]}}{\widetilde{W}_0^{\alpha-j} + \widetilde{W}_1^{\alpha-j}} \right)^{-1} \cdot \varphi_0 \\ &= \left(\prod_{j=0}^{\alpha-1} \frac{\widetilde{W}^{\alpha-j}_{w[\alpha-j]}}{\widetilde{W}_0^{\alpha-j} + \widetilde{W}_1^{\alpha-j}} \right)^{-1} \cdot \left(\frac{e^{-4}}{R(s_i^\alpha)} \right) \end{aligned}$$

Thus, to show that $\varphi < 1$ at the end of the algorithm, it suffices to show that on any run of the algorithm, we have that

$$\left(\prod_{j=0}^{\alpha-1} \frac{\widetilde{W}^{\alpha-j}_{w[\alpha-j]}}{\widetilde{W}_0^{\alpha-j} + \widetilde{W}_1^{\alpha-j}} \right)^{-1} \cdot \left(\frac{e^{-4}}{R(s_i^\alpha)} \right) \quad < \quad 1.$$

Now since $\widetilde{W}_q^t = (1 \pm k^{-1/4})^t |W_q^t|$ for all $q \in \{0,1\}$ and $t \in \{1,\ldots,\alpha\}$, it follows that at the final recursive call to **Sample**, we have that:

$$\begin{aligned}
&\varphi \;=\; \\
&\left(\prod_{j=0}^{\alpha-1} \frac{\widetilde{W}^{\alpha-j}_{w[\alpha-j]}}{\widetilde{W}_0^{\alpha-j} + \widetilde{W}_1^{\alpha-j}}\right)^{-1} \cdot \left(\frac{e^{-4}}{R(s_i^\alpha)}\right) \leq \\
&\left(\prod_{j=0}^{\alpha-1} \frac{(1-k^{-1/4})^{\alpha-j}|W^{\alpha-j}_{w[\alpha-j]}|}{(1+k^{-1/4})^{\alpha-j}(|W_0^{\alpha-j}| + |W_1^{\alpha-j}|)}\right)^{-1} \cdot \left(\frac{e^{-4}}{R(s_i^\alpha)}\right) = \\
&\left(\prod_{j=0}^{\alpha-1} \frac{(1-k^{-1/4})^{\alpha-j}}{(1+k^{-1/4})^{\alpha-j}}\right)^{-1} \cdot \left(\prod_{j=0}^{\alpha-1} \frac{|W^{\alpha-j}_{w[\alpha-j]}|}{|W_0^{\alpha-j}| + |W_1^{\alpha-j}|}\right)^{-1} \cdot \left(\frac{e^{-4}}{R(s_i^\alpha)}\right)
\end{aligned}$$

Recall that by definition, we have that $|W^{\alpha-j}_{w[\alpha-j]}| = |W_1^{\alpha-(j+1)}| + |W_0^{\alpha-(j+1)}|$ for every $j = 0, 1, 2, \ldots, \alpha-1$. Also note that $|W^1_{w[1]}| = 1$, since W^1 is the set of strings $x \in \{0,1\}$ such that $x \circ w^1 \in U(s_i^\alpha)$, and $W^1_{w[1]}$ is the subset of W^1 with the last bit equal to $w[1]$ (of which there is just one). Thus, given that $|W^\alpha| = |W_0^\alpha| + |W_1^\alpha|$, we have that:

$$\left(\prod_{j=0}^{\alpha-1} \frac{|W^{\alpha-j}_{w[\alpha-j]}|}{|W_0^{\alpha-j}| + |W_1^{\alpha-j}|}\right)^{-1} \;=\; |W^\alpha| \;=\; |U(s_i^\alpha)|,$$

and so

$$\begin{aligned}
\varphi &\leq \left(\prod_{j=0}^{\alpha-1} \frac{(1-k^{-1/4})^{\alpha-j}}{(1+k^{-1/4})^{\alpha-j}}\right)^{-1} \cdot |U(s_i^\alpha)| \cdot \left(\frac{e^{-4}}{R(s_i^\alpha)}\right) \\
&= \left(\prod_{j=0}^{\alpha-1} \frac{(1+k^{-1/4})^{\alpha-j}}{(1-k^{-1/4})^{\alpha-j}}\right) \cdot |U(s_i^\alpha)| \cdot \left(\frac{e^{-4}}{R(s_i^\alpha)}\right) \\
&= \left(\frac{1+k^{-1/4}}{1-k^{-1/4}}\right)^{\frac{\alpha(\alpha+1)}{2}} \cdot |U(s_i^\alpha)| \cdot \left(\frac{e^{-4}}{R(s_i^\alpha)}\right) \\
&\leq \left(\frac{1+k^{-1/4}}{1-k^{-1/4}}\right)^{\alpha^2} \cdot |U(s_i^\alpha)| \cdot \left(\frac{e^{-4}}{R(s_i^\alpha)}\right) \\
&\leq \left(\frac{1+k^{-1/8}}{(1+k^{-1/8})(1-k^{-1/8})}\right)^{\alpha^2} \cdot |U(s_i^\alpha)| \cdot \left(\frac{e^{-4}}{R(s_i^\alpha)}\right) \\
&= \left(\frac{1}{1-k^{-1/8}}\right)^{\alpha^2} \cdot |U(s_i^\alpha)| \cdot \left(\frac{e^{-4}}{R(s_i^\alpha)}\right) \\
&\leq \left(\frac{1}{1-1/n^8}\right)^{n^2} \cdot |U(s_i^\alpha)| \cdot \left(\frac{e^{-4}}{R(s_i^\alpha)}\right) \\
&\leq \left(1+1/n^4\right)^{n^2} \cdot |U(s_i^\alpha)| \cdot \left(\frac{e^{-4}}{R(s_i^\alpha)}\right) \\
&\leq e^{1/n^2} \cdot |U(s_i^\alpha)| \cdot \left(\frac{e^{-4}}{R(s_i^\alpha)}\right) \\
&\leq \frac{1}{1-1/n^2} \cdot |U(s_i^\alpha)| \cdot \left(\frac{e^{-4}}{R(s_i^\alpha)}\right) \\
&\leq (1+1/n) \cdot |U(s_i^\alpha)| \cdot \left(\frac{e^{-4}}{R(s_i^\alpha)}\right).
\end{aligned}$$

Where the inequality $\left(\frac{1}{1-k^{-1/8}}\right)^{\alpha^2} \leq \left(\frac{1}{1-1/n^8}\right)^{n^2}$ holds because $\alpha \leq n$, $n \geq 2$ and $k > (nm)^{64}$, so $k^{-1/8} < (\frac{1}{nm})^8 \leq \frac{1}{n^8}$. Furthermore, since we have $R(s_i^\alpha) = (1 \pm k^{-1/4})|U(s_i^\alpha)|$ by assumption, and we know that $k^{-1/4} < (\frac{1}{nm})^{16} \leq \frac{1}{n^{16}}$, we conclude that $(1 - \frac{1}{n^{16}})|U(s_i^\alpha)| <$

$(1-k^{-1/4})|U(s_i^\alpha)| \leq |R(s_i^\alpha)|$. Therefore,

$$\begin{aligned}
\varphi &\leq (1+1/n)\cdot|U(s_i^\alpha)|\cdot\left(\frac{e^{-4}}{R(s_i^\alpha)}\right)\\
&\leq (1+1/n)\cdot\frac{|R(s_i^\alpha)|}{(1-1/n^{16})}\cdot\left(\frac{e^{-4}}{R(s_i^\alpha)}\right)\\
&= \left(\frac{n^{15}\cdot(n+1)}{(n^{16}-1)}\right)\cdot e^{-4}\\
&< 2\cdot e^{-4} < e^{-3} < 1,
\end{aligned}$$

since $n \geq 2$. Hence, we know that under the assumptions stated for this Proposition, for each call, and in particular on the last call we have $\varphi \in (0,1)$. The probability that the procedure outputs **fail** is then only due to Step 2 in Algorithm 3. That is, the probability we output fail is $(1-\varphi)$ where φ is as computed above. So to show that the failure probability is at most $1-e^{-5}$, we compute a lower bound for φ similarly as we computed an upper bound for it:

$$\begin{aligned}
\varphi &\geq \left(\prod_{j=0}^{\alpha-1}\frac{(1+k^{-1/4})^{\alpha-j}}{(1-k^{-1/4})^{\alpha-j}}\right)^{-1}\cdot|U(s_i^\alpha)|\cdot\left(\frac{e^{-4}}{R(s_i^\alpha)}\right)\\
&= \left(\frac{1-k^{-1/4}}{1+k^{-1/4}}\right)^{\frac{\alpha(\alpha+1)}{2}}\cdot|U(s_i^\alpha)|\cdot\left(\frac{e^{-4}}{R(s_i^\alpha)}\right)\\
&\geq \left(\frac{(1-k^{-1/8})(1+k^{-1/8})}{1+k^{-1/8}}\right)^{\frac{\alpha(\alpha+1)}{2}}\cdot|U(s_i^\alpha)|\cdot\left(\frac{e^{-4}}{R(s_i^\alpha)}\right)\\
&= (1-k^{-1/8})^{\frac{\alpha(\alpha+1)}{2}}\cdot|U(s_i^\alpha)|\cdot\left(\frac{e^{-4}}{R(s_i^\alpha)}\right)\\
&\geq (1-k^{-1/8})^{\alpha^2}\cdot|U(s_i^\alpha)|\cdot\left(\frac{e^{-4}}{R(s_i^\alpha)}\right)\\
&\geq (1-1/n^8)^{\alpha^2}\cdot|U(s_i^\alpha)|\cdot\left(\frac{e^{-4}}{R(s_i^\alpha)}\right)\\
&\geq (1-1/n^8)^{n^2}\cdot|U(s_i^\alpha)|\cdot\left(\frac{e^{-4}}{R(s_i^\alpha)}\right)\\
&\geq e^{-2/n^6}\cdot|U(s_i^\alpha)|\cdot\left(\frac{e^{-4}}{R(s_i^\alpha)}\right)\\
&\geq (1-2/n^6)\cdot|U(s_i^\alpha)|\cdot\left(\frac{e^{-4}}{R(s_i^\alpha)}\right)\\
&\geq (1-2/n^6)\cdot\frac{R(s_i^\alpha)}{(1+1/n^{16})}\cdot\left(\frac{e^{-4}}{R(s_i^\alpha)}\right)\\
&= \left(\frac{n^{10}(n^6-2)}{n^{16}+1}\right)\cdot e^{-4}\\
&\geq e^{-1}\cdot e^{-4} = e^{-5}
\end{aligned}$$

Notice that to infer that $(1-1/n^8)^{n^2} \geq e^{-2/n^6}$, we use the fact that $(1-x) \geq e^{-2x}$ for $x \in [0, \frac{1}{2}]$ and the hypothesis $n \geq 2$. Now, the probability of the output w being a particular $x \in U(s_i^\alpha)$ is given by the following expression:

$$\begin{aligned}
&\mathbf{Pr}(w = x)\\
&\quad = \mathbf{Pr}(w^0 = x \wedge \text{ the last call to } \textbf{Sample} \text{ does not fail})\\
&\quad = \mathbf{Pr}(\text{last call to } \textbf{Sample} \text{ does not fail} \mid w^0 = x)\cdot\mathbf{Pr}(w^0 = x)\\
&\quad = \left(\prod_{j=0}^{\alpha-1}\frac{\widetilde{W}^{\alpha-j}_{x[\alpha-j]}}{\widetilde{W}_0^{\alpha-j}+\widetilde{W}_1^{\alpha-j}}\right)^{-1}\cdot\left(\frac{e^{-4}}{R(s_i^\alpha)}\right)\cdot\left(\prod_{j=0}^{\alpha-1}\frac{\widetilde{W}^{\alpha-j}_{x[\alpha-j]}}{\widetilde{W}_0^{\alpha-j}+\widetilde{W}_1^{\alpha-j}}\right)
\end{aligned}$$

$$= \frac{e^{-4}}{R(s_i^\alpha)},$$

as desired. Therefore, conditioned on not outputting **fail**, our sampler returns $w \in U(S_i^\alpha)$ *uniformly at random*, which is the desired result.

C.3 Proof of Proposition 6.4

Within a call to $\textbf{Sample}(\{s_i^\alpha\}, \varepsilon, \frac{e^{-4}}{R(s_i^\alpha)})$, note that each $\widetilde{W}_q^t$ is of the form

$$\widetilde{W}_q^t \quad = \quad \sum_{s \in T_q^t} R(s) \cdot \frac{\left|X(s) \setminus \left(\bigcup_{s' \in T_q^t : s' \prec s} U(s')\right)\right|}{|X(s)|}.$$

Now by the assumption that Property 2 holds, we know that for each $s \in T_q^t$:

$$\frac{\left|U(s) \setminus \left(\bigcup_{s' \in T_q^t : s' \prec s} U(s')\right)\right|}{|U(s)|} - k^{-1/3} \; < \; \frac{\left|X(s) \setminus \left(\bigcup_{s' \in T_q^t : s' \prec s} U(s')\right)\right|}{|X(s)|} \; < \; \frac{\left|U(s) \setminus \left(\bigcup_{s' \in T_q^t : s' \prec s} U(s')\right)\right|}{|U(s)|} + k^{-1/3}$$

Moreover, given that T_q^t is a subset of states in layer $t-1$, by assumption that Property 1 holds for the nodes in these layers, we have that:

$$(1-k^{-1/4})^{t-1}|U(s)| \; \leq \; R(s) \; \leq \; (1+k^{-1/4})^{t-1}|U(s)|.$$

Putting these two bounds together, it follows that

$$(1-k^{-1/4})^{t-1} \sum_{s \in T_q^t} \left|U(s) \setminus \Big(\bigcup_{s' \in T_q^t : s' \prec s} U(s') \Big)\right| - k^{-1/3}|U(s)| \; < \; \widetilde{W}_q^t \; < \; (1+k^{-1/4})^{t-1} \sum_{s \in T_q^t} \left|U(s) \setminus \Big(\bigcup_{s' \in T_q^t : s' \prec s} U(s') \Big)\right| + k^{-1/3}|U(s)|.$$

Notice that

$$\sum_{s \in T_q^t} \left|U(s) \setminus \Big(\bigcup_{s' \in T_q^t : s' \prec s} U(s') \Big)\right| \; = \; \Big| \bigcup_{s \in T_q^t} U(s) \Big| \; = \; |W_q^t|$$

is the true value that we are trying to estimate. Let $s^* \in T_q^t$ be such that $|U(s^*)| \geq |U(s)|$ for all other $s \in T_q^t$. To show $\widetilde{W}_q^t = (1 \pm k^{-1/4})^t |W_q^t|$, we first show the upper bound. We have that:

$$\begin{aligned}
\widetilde{W}_q^t \; < \; & \left((1+k^{-1/4})^{t-1} \sum_{s \in T_q^t} \left|U(s) \setminus \Big(\bigcup_{s' \in T_q^t : s' \prec s} U(s') \Big)\right| \right) + mk^{-1/3}(1+k^{-1/4})^{t-1}|U(s^*)| \; < \\
& \left((1+k^{-1/4})^{t-1} \sum_{s \in T_q^t} \left|U(s) \setminus \Big(\bigcup_{s' \in T_q^t : s' \prec s} U(s') \Big)\right| \right) + \\
& k^{-1/4}(1+k^{-1/4})^{t-1} \left(\sum_{s \in T_q^t} \left|U(s) \setminus \Big(\bigcup_{s' \in T_q^t : s' \prec s} U(s') \Big)\right| \right) = \\
& (1+k^{-1/4})^{t} \sum_{s \in T_q^t} \left|U(s) \setminus \Big(\bigcup_{s' \in T_q^t,\; s' \prec s} U(s') \Big)\right|.
\end{aligned}$$

Notice that the second inequality holds because $k^{-1/12} < (nm)^{-4} < 1/m$ and

$$|U(s^*)| \quad \leq \quad \sum_{s \in T_q^t} \left|U(s) \setminus \Big(\bigcup_{s' \in T_q^t : s' \prec s} U(s') \Big)\right|,$$

since $U(s^*) \subseteq \bigcup_{s \in T_q^t} U(s)$. For the lower bound, we consider again s^* and obtain the following:

$$\begin{aligned}
\widetilde{W}_q^t \;>\; & \\
& \left((1-k^{-1/4})^{t-1} \sum_{s \in T_q^t} \left|U(s) \setminus \Big(\bigcup_{s' \in T_q^t : s' \prec s} U(s')\Big)\right|\right) - \\
& \qquad\qquad mk^{-1/3}(1-k^{-1/4})^{t-1}|U(s^*)| \;> \\
& \left((1-k^{-1/4})^{t-1} \sum_{s \in T_q^t} \left|U(s) \setminus \Big(\bigcup_{s' \in T_q^t : s' \prec s} U(s')\Big)\right|\right) - \\
& \quad k^{-1/4}(1-k^{-1/4})^{t-1}\left(\sum_{s \in T_q^t} \left|U(s) \setminus \Big(\bigcup_{s' \in T_q^t : s' \prec s} U(s')\Big)\right|\right) = \\
& (1-k^{-1/4})^{t} \sum_{s \in T_q^t} \left|U(s) \setminus \Big(\bigcup_{s' \in T_q^t,\ s' \prec s} U(s')\Big)\right|.
\end{aligned}$$

Therefore, we obtain that:

$$\begin{aligned}
\widetilde{W}_q^t \;&=\; (1 \pm k^{-1/4})^{t} \sum_{s \in T_q^t} \left|U(s) \setminus \Big(\bigcup_{s' \in T_q^t,\ s' \prec s} U(s')\Big)\right| \\
&=\; (1 \pm k^{-1/4})^{t} |W_q^t|.
\end{aligned}$$

Note since $R(s_i^\alpha)$ is the sum of two such quantities of the above form, we also obtain $R(s_i^\alpha) = \widetilde{W}_0^\alpha + \widetilde{W}_1^\alpha = (1 \pm k^{-1/4})^\alpha(|W_0^t| + |W_1^t|) = (1 \pm k^{-1/4})^\alpha |U(s_i^\alpha)|$, which concludes the proof of the proposition.

C.4 Proof of Lemma 6.6

Fix a state s_i^α. If s_i^α is exactly handled, then we have $X(s_i^\alpha) = U(s_i^\alpha)$, so Properties 1 and 2 hold trivially for state s_i^α (with probability 1). So we can assume that s_i^α is not exactly handled, which implies $|X(s_i^\alpha)| = k$. Now Property 1 for s_i^α follows directly from Proposition 6.4 (with probability 1). Thus, $R(s_i^\alpha) = (1 \pm k^{-1/4})^\alpha |U(s_i^\alpha)|$. Now given that $R(s_i^\alpha) = (1 \pm k^{-1/4})^\alpha |U(s_i^\alpha)|$ and that the event $\mathcal{E}^1 \wedge \cdots \wedge \mathcal{E}^{\alpha-1}$ holds, the conditions for Proposition 6.3 are satisfied by Proposition 6.4 when the queries to $\mathbf{Sample}(\{s_i^\alpha\}, \varepsilon, \frac{e^{-4}}{R(s_i^\alpha)})$ are made for the vertex s_i^α. Thus each sample stored in the set $X(s_i^\alpha)$ which is returned from $\mathbf{Sample}(\{s_i^\alpha\}, \varepsilon, \frac{e^{-4}}{R_{s_i^\alpha}})$ is a truly uniform independent and identically distributed sample of $U(s_i^\alpha)$. So fix any set $L \subseteq \{1, \ldots, m\}$. By Hoeffding's inequality:

$$\mathbf{Pr}\left(\left| \frac{\left|X(s_i^\alpha) \setminus \left(\bigcup_{j \in L} U(s_j^\alpha)\right)\right|}{|X(s_i^\alpha)|} \;-\; \frac{|U(s_i^\alpha) \setminus \left(\bigcup_{j \in L} U(s_j^\alpha)\right)|}{|U(s_i^\alpha)|} \right| \;\geq\; k^{-1/3} \right) \;\leq\; 2e^{-2k(k^{-1/3})^2} \;=\; 2e^{-2k^{1/3}}.$$

Here, we have defined the ℓ-th independent random variables used in Hoeffding's inequality to be the indicator variable of the event that the ℓ-th sample in $X(s_i^\alpha)$ is contained in $X(s_i^\alpha) \setminus \left(\bigcup_{j \in L} U(s_j^\alpha)\right)$. Note that because the samples in $X(s_i^\alpha)$ are uniform, the expectation of this random variable is precisely

$$\frac{\left|U(s_i^\alpha) \setminus \left(\bigcup_{j \in L} U(s_j^\alpha)\right)\right|}{|U(s_i^\alpha)|}$$

for every $\ell \in \{1, \ldots, k\}$. Therefore, we can use Hoeffding's inequality considering that $|X(s_i^\alpha)| = k$. Now there are at most $m2^m$ of such indexes $i \in \{1, \ldots, m\}$ and subsets L of $\{1, \ldots, m\}$. Given that $(nm)^{64} < k$, we have that $\log_2(m) + m \leq m^{21} \leq (nm)^{21} < k^{1/3}$, from which we conclude that $m2^m < e^{k^{1/3}}$. Hence, we conclude that:

$$\begin{aligned}
\mathbf{Pr}\Bigg(\bigvee_{i \in \{1,\ldots,m\}} \bigvee_{L \subseteq \{1,\ldots,m\}} \Bigg(& \left| \frac{\left|X(s_i^\alpha) \setminus \left(\bigcup_{j \in L} U(s_j^\alpha)\right)\right|}{|X(s_i^\alpha)|} - \right. \\
& \left. \frac{|U(s_i^\alpha) \setminus \left(\bigcup_{j \in L} U(s_j^\alpha)\right)|}{|U(s_i^\alpha)|} \right| \;\geq\; k^{-1/3} \Bigg) \Bigg) \;\leq \\
\sum_{i \in \{1,\ldots,m\}} \sum_{L \subseteq \{1,\ldots,m\}} \mathbf{Pr}\Bigg(& \left| \frac{\left|X(s_i^\alpha) \setminus \left(\bigcup_{j \in L} U(s_j^\alpha)\right)\right|}{|X(s_i^\alpha)|} - \right. \\
& \left. \frac{|U(s_i^\alpha) \setminus \left(\bigcup_{j \in L} U(s_j^\alpha)\right)|}{|U(s_i^\alpha)|} \right| \;\geq\; k^{-1/3} \Bigg) \;\leq
\end{aligned}$$

$$\sum_{i\in\{1,\ldots,m\}} \sum_{L\subseteq\{1,\ldots,m\}} 2e^{-2k^{1/3}} \leq$$
$$m2^m 2e^{-2k^{1/3}} < e^{k^{1/3}} 2e^{-2k^{1/3}} = 2e^{-k^{1/3}}.$$

Thus, by definition of Property 2, we deduce that $\mathbf{Pr}(\mathcal{E}^\alpha \mid \mathcal{E}^1 \wedge \cdots \wedge \mathcal{E}^{\alpha-1}) \geq 1 - 2e^{-k^{1/3}}$, which was to be shown.

C.5 Proof of Theorem 6.7

The full algorithm is given in Algorithm 4. We first consider its correctness. By definition of step (1) of this algorithm, we know that it returns the exact value $|\mathcal{L}_n(N)|$ when $n \leq 12$. Thus, we assume in the rest of the proof that $n \geq 13$. By Lemma 6.6, we have that:

$$\begin{aligned}\mathbf{Pr}(\mathcal{E}^1 \wedge \cdots \wedge \mathcal{E}^n) &= \prod_{i=1}^{n} \mathbf{Pr}(\mathcal{E}^i \mid \mathcal{E}^1 \wedge \cdots \wedge \mathcal{E}^{i-1}) \\ &\geq \prod_{i=1}^{n} (1 - 2e^{-k^{1/3}}) = (1 - 2e^{-k^{1/3}})^n.\end{aligned}$$

Moreover, we have that:

$$\begin{aligned}(1 - 2e^{-k^{1/3}})^n &= 1 + \sum_{j=1}^{n} \binom{n}{j} (-1)^j 2^j e^{-jk^{1/3}} \\ &\geq 1 + \sum_{j=1}^{n} \binom{n}{j} (-1) 2^j e^{-jk^{1/3}} \\ &= 1 - \sum_{j=1}^{n} \binom{n}{j} 2^j e^{-jk^{1/3}} \\ &\geq 1 - 2^n e^{-k^{1/3}} \sum_{j=1}^{n} \binom{n}{j} \\ &\geq 1 - 2^n e^{-k^{1/3}} 2^n \\ &\geq 1 - e^n e^{-k^{1/3}} e^n \\ &= 1 - e^{2n - k^{1/3}} \\ &\geq 1 - e^{2n - (nm)^{21}} \\ &\geq 1 - e^{-nm}\end{aligned}$$

Therefore, we conclude that $\mathbf{Pr}(\mathcal{E}^1 \wedge \cdots \wedge \mathcal{E}^n) \geq 1 - e^{-nm}$. If $(\mathcal{E}^1 \wedge \cdots \wedge \mathcal{E}^n)$ holds, then we know by Proposition 6.4 that the conditions of Proposition 6.3 hold. Thus, we conclude that by running each **Sample** procedure at most $\lceil (\frac{nm}{\delta})^4 \rceil$ times, we obtain at least one sample with probability at least $1-(1-e^{-5})^{\lceil (\frac{nm}{\delta})^4 \rceil} \geq 1-e^{-c_1 \cdot (\frac{nm}{\delta})^4}$, where $c_1 = |\ln(1-e^{-5})| > 0$. In other words, the algorithm does not fail in step 5 (c) (iii) with probability at least $1 - e^{-c_1 \cdot (\frac{nm}{\delta})^4}$. Since we require at most nmk samples over the course of the entire algorithm, where $k = \lceil (\frac{mn}{\delta})^{64} \rceil$, by a union bound we obtain all desired samples with probability at least $1 - nm\lceil (\frac{mn}{\delta})^{64} \rceil e^{-c_1 (\frac{nm}{\delta})^4} \geq 1 - e^{-c_2 \cdot (\frac{nm}{\delta})^4}$, where $c_2 > 0$ is a fixed constant (notice that such a constant exists since $n \geq 13$). Moreover, given that $\mathcal{E}^n$ holds, we know that $R(s_{final}) = (1 \pm k^{-1/4})^n |\mathcal{L}_n(N)|$, that is,

$$(1 - k^{-1/4})^n |\mathcal{L}_n(N)| \leq R(s_{final}) \leq (1 + k^{-1/4})^n |\mathcal{L}_n(N)|. \tag{1}$$

But we have that:

$$\begin{aligned}(1 + k^{-1/4})^n &\leq \left(1 + \left(\frac{\delta}{mn}\right)^{16}\right)^n \\ &= \left[\left(1 + \left(\frac{1}{(\frac{nm}{\delta})^{16}}\right)\right)^{(\frac{nm}{\delta})^{16}}\right]^{\frac{\delta^{16}}{n^{15}m^{16}}} \\ &\leq e^{\frac{\delta^{16}}{n^{15}m^{16}}} \\ &\leq 1 + 2\frac{\delta^{16}}{n^{15}m^{16}} \quad \text{since } e^x \leq (1+2x) \text{ for } x \in [0,1] \\ &= 1 + \delta \cdot \frac{2\delta^{15}}{n^{15}m^{16}}\end{aligned}$$

$$\begin{aligned} &\leq 1+\delta\cdot\frac{1}{2^{14}m^{16}}\\ &\leq 1+\delta, \end{aligned}$$

and we also have that:

$$\begin{aligned} (1k^{-1/4})^n &\geq \left(1-\left(\frac{\delta}{mn}\right)^{16}\right)^n\\ &= \left[\left(1-\left(\frac{1}{(\frac{nm}{\delta})^{16}}\right)\right)^{(\frac{nm}{\delta})^{16}}\right]^{\frac{\delta^{16}}{n^{15}m^{16}}}\\ &\geq (e^{-2})^{\frac{\delta^{16}}{n^{15}m^{16}}} \quad \text{since } \left(1-\frac{1}{x}\right)^x \geq e^{-2} \text{ for } x\geq 2\\ &= e^{-\frac{2\delta^{16}}{n^{15}m^{16}}}\\ &\geq 1-\frac{2\delta^{16}}{n^{15}m^{16}}\\ &= 1-\delta\cdot\frac{2\delta^{15}}{n^{15}m^{16}}\\ &\geq 1-\delta\cdot\frac{1}{2^{14}m^{16}}\\ &\geq 1-\delta. \end{aligned}$$

Thus, we conclude from (1) that:

$$(1-\delta)|\mathcal{L}_n(N)| \leq R(s_{final}) \leq (1+\delta)|\mathcal{L}_n(N)|. \tag{2}$$

Summing up, our algorithm returns a value $R = R(s_{final})$ such that $\left|R - |\mathcal{L}_n(N)|\right| \leq \delta|\mathcal{L}_n(N)|$ with probability at least

$$\begin{aligned} (1-e^{-nm})(1-e^{-c_2\cdot(\frac{mn}{\delta})^4}) &> 1-e^{-nm}-e^{-c_2\cdot(\frac{mn}{\delta})^4}\\ &\geq 1-e^{-nm}-e^{-c_2nm}\\ &\geq 1-2e^{-\min\{1,c_2\}nm}\\ &= 1-2e^{-\tau nm}, \end{aligned}$$

where $\tau = \min\{1, c_2\} > 0$ is the fixed constant mention in the statement of the theorem.

Now notice that in the previous analysis, we neglected the potential for failure based on the event that $R(s_i^\alpha) = 0$ at some point in step 5 (b) of Algorithm 4. We now address this issue. Observe that conditioned on $(\mathcal{E}^1 \wedge \cdots \wedge \mathcal{E}^n)$, we have for every state s_i^α that $R(s_i^\alpha) = 0$ if and only if $U(s_i^\alpha) = 0$. Moreover, since $U(s_i^\alpha) = 0$ if and only if there is no path from the start state s_{start} to s_i^α in the graph N_{unroll}, and since we remove all such unreachable states in the third step of Algorithm 4, it follows that conditioned on $(\mathcal{E}^1 \wedge \cdots \wedge \mathcal{E}^n)$, we will never have in the algorithm that $R(s_i^\alpha) = 0$ for a state s_i^α. Thus, the event that $R(s_i^\alpha) = 0$ and we fail (outputting 0) in step 5 (b) in Algorithm 4 cannot occur conditioned on $(\mathcal{E}^1 \wedge \cdots \wedge \mathcal{E}^n)$. Since there are no other opportunities for failure of the algorithm, this completes the proof of correctness.

It remains now to analyze the runtime. By a simple inspection of the steps of the algorithm, it is easy to see that it runs in polynomial time in $\frac{mn}{\delta}$. Thus, in what follows we just made some comments about the complexity of the key steps of the algorithms.

We first consider the portion of Algorithm 4 which computes $X(s)$, $R(s)$ for the exactly handled states s. For each such state, one can determine if it is exactly handled by looking at the values of $X(s')$ for all exactly handled states s' with an edge into s. By looking at the union of all such sets, we can determine the exact value of $U(s)$, and in particular determine if s is exactly handled. Since each $X(s')$ has at most k elements, the total time is $O(mk)$ to take the union (one can add a factor of mk to this complexity for inserting into a linked list to construct this union if one desired, or use another data-structure such as Union-Find). Thus the total time to compute $X(s)$, $R(s)$ for all exactly handled states, and determine which states are exactly handled, is polynomial in nmk and, thus, polynomial in $\frac{nm}{\delta}$ as needed.

Second, we notice that the procedure **Sample** runs in polynomial time, and it is called at most $nmk\lceil(\frac{nm}{\delta})^4\rceil$ times, which is a polynomial number of times in $\frac{mn}{\delta}$. Finally, we also notice that each value $\widetilde{W}_q$ with $q \in \{0,1\}$ can be computed in polynomial time, since for each query to compute

$$\frac{\left|X(s) \setminus \left(\bigcup_{s'\in T_b(s_i^\alpha):\, s'\prec s} U(s')\right)\right|}{|X(s)|}$$

for some pair of states s_i^α and $s \in T_b(s_i^\alpha)$, we check for each of the samples $x \in X(s)$ whether there is a path from s_{start} to s' for some $s' \in T_b(s_i^\alpha)$ such that $s' \prec s$, which can be done in time $O(nm)$ by a breadth first search. Thus, the total time for this step is polynomial as $|X(s)| \leq k$ and $|T_b(s_i^\alpha)| \leq m$. This concludes the proof of the theorem.