Efficient Logspace Classes for Enumeration, Counting, and Uniform Generation

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

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

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

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

ABSTRACT

We study two simple yet general complexity classes, which provide a unifying framework for efficient query evaluation in areas like graph databases and information extraction, 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 logarithmic-space transducers (NL transducers). For the first class, we consider the case of unambiguous NL transducers, and we prove constant delay enumeration, and both counting and uniform generation of solutions in polynomial time. For the second class, we consider unrestricted NL transducers, and we obtain polynomial delay enumeration, approximate counting in polynomial time, and polynomial-time randomized algorithms for uniform generation. More specifically, we show that each problem in this second class admits a fully polynomial-time randomized approximation scheme (FPRAS) and a polynomial-time Las Vegas algorithm (with preprocessing) for uniform generation. Remarkably, the key idea to prove these results is to show that the fundamental problem #NFA admits an FPRAS, where #NFA is the problem of counting the number of strings of length n (given in unary) accepted by a non-deterministic finite automaton (NFA). While this problem is known to be #P-complete and, more precisely, SpanL-complete, it was open whether this problem admits an FPRAS. In this work, we solve this open problem, and obtain as a welcome corollary that every function in SpanL admits an FPRAS.

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 field. In the most classical view of this problem, one is interested in computing all the answers, or solutions, to a query. To present such a view, consider a running example from the area of graph databases [8].

This is a minor revision of the paper entitled “Efficient Logspace Classes for Enumeration, Counting, and Uniform Generation”, published in PODS’19, ISBN978-1-4503-6227-6/19/06, June 30-July 5, 2019, Amsterdam, Netherlands. DOI: http://dx.doi.org/10.1145/3294052.3319704

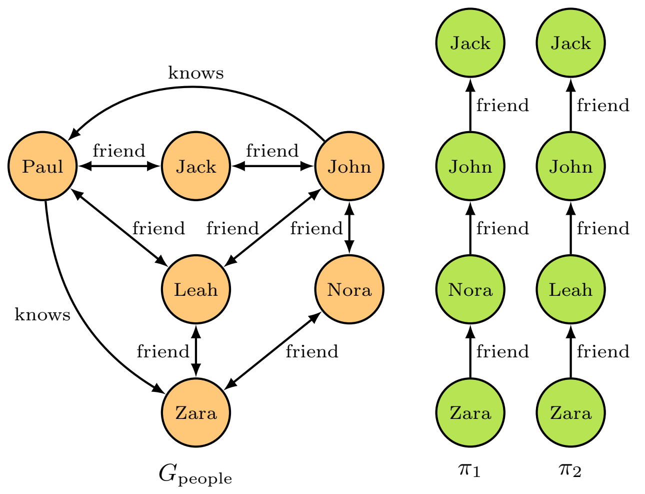


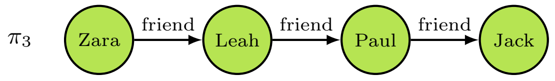
Figure 1: A graph database G_{people}, and two paths π_1, π_2 of friends in G_{people} from Zara to Jack.

Given a set Δ of labels, one can model a graph database G as a pair (V, E) where V is a set of vertices and $E \subseteq V \times \Delta \times V$ is a set of labeled edges. For example, G_{people} in Figure 1 is a graph database storing information about people and their relationships; in particular, the set of labels for G_{people} is $\{\text{friend}, \text{knows}\}$, so that a triple (a, friend, b) indicates that a and b are friends, while a triple (a, knows, b) indicates that a knows b. Path queries are a fundamental way to retrieve information from graph databases [8, 18]. In its simplest form, a path query Q over a graph database $G = (V, E)$ is a triple (a, r, b), where $a, b \in V$ and r is a regular expression over the set Δ of edge labels for G. An answer to Q over G is a path from a to b whose labels conform to r. Formally, such a path is a sequence $\pi = v_0, p_1, v_1, p_2, \ldots, p_n, v_n$ of vertices and labels such that $(v_i, p_{i+1}, v_{i+1}) \in E$, $a = v_0$ and $b = v_n$. Moreover, π is said to conform to r if the string $p_1 p_2 \cdots p_n$ is in the regular language defined by r. For example, $Q_1 = (\text{Zara}, \text{friend}^*, \text{Jack})$ is a path query over the graph database G_{people} in Figure 1, for which two answers are the paths π_1, π_2 shown in this figure. Thus, an answer to Q_1 over G_{people}

is a path of friends from Zara to Jack. The set of answers of a path query Q over a graph database G is denoted by $Q(G)$. Clearly, $Q(G)$ can be an infinite set, as paths can contain cycles, so there can be an infinite number of them in a graph. For this reason, the length of the paths to be retrieved must also be specified when posing a path query; the length of a path $\pi = v_0, p_1, \ldots, p_n, v_n$, denoted by $|\pi|$, is defined as n. Hence, in the most classical view of the query answering problem in graph databases, the input is a triple (G, Q, n) with G a graph database, Q a path query over G and n a natural number. The task then is to compute *all* paths π such that $\pi \in Q(G)$ and $|\pi| = n$. For example, assuming that Q_1 = (Zara, friend*, Jack), the paths π_1, π_2 in Figure 1 belong to $Q_1(G_{\text{people}})$ when $n = 3$.

As the quantity of data becomes enormously large, the number of answers to a query could also be enormous, so computing the complete set of solutions can be prohibitively expensive. In our running example, just think about computing all the paths from a source to a target node over a large graph, this set of answers can be huge and infeasible to produce in practice [9, 28]. To overcome this limitation, the idea of enumerating the answers to a query with a *small delay* has recently attracted a lot of attention [33, 29]. More specifically, the idea is to divide the computation of the answers 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. Considering again G_{people} and path query Q_1 = (Zara, friend*, Jack), such an algorithm has a preprocessing phase that allows it to return a first path in $Q_1(G_{\text{people}})$ in polynomial time, say π_1, and then to return one by one the answers in $Q_1(G_{\text{people}})$ taking polynomial time between any two consecutive outputs, say taking polynomial time to return π_2 after π_1. In the case of constant delay enumeration algorithms, the preprocessing phase should take polynomial time, while the time between consecutive answers should be constant. Such 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 polynomial number of answers.

Unfortunately, because of the data structures used in the preprocessing phase, these enumeration algorithms usually return answers that are similar to each other [12, 33, 17]. In our running example, an enumeration algorithm for the query Q_1 = (Zara, friend*, Jack) can return as the first two answers the paths π_1 and π_2 shown in Figure 1, which are similar to each other as they only differ on the second node: Nora and Leah. In this respect, other approaches can be used to return some solutions efficiently but improving the variety. For instance, if we are going to generate two answers to Q_1 over G_{people}, instead of producing paths π_1 and π_2 in Figure 1, it would be desirable to improve the variety by producing π_1 and the following path:



The possibility of generating an answer uniformly, at random, is a desirable condition to improve the variety, if it can be done efficiently. Notice that the uniform generation of answers is also important for other query evaluation tasks like approximate query answering, and estimating aggregates and parameters for query optimization [14, 3, 36]. Moreover, the possibility of returning varied solutions has been identified as an important feature not only in databases, but also for algorithms that retrieve information in a broader sense [2, 1].

Efficient algorithms for enumeration or uniform generation 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. For example, for the query Q_1 = (Zara, friend*, Jack) over the graph database G_{people} in Figure 1, we have that $Q_1(G_{\text{people}})$ contains three paths π such that $|\pi| = 3$. Hence, if we have an efficient algorithm to compute the number of paths in $Q_1(G_{\text{people}})$ of length 3, then we know that there are no more answers to produce after generating paths π_1 and π_2 in Figure 1, and the previous path π_3. Similar than for enumeration and uniform generation, counting the number of solutions has other applications in query evaluation like computing the size of intermediate results, computing histograms, among others [20, 23].

Taken together, enumeration, counting, and uniform generation techniques form a powerful attacking trident when confronting to the query answering problem in our running example and, more generally, in any database system. The goal of this work is to find efficient algorithms for these problems but following a principled approach, instead of focusing on them in isolation and for some specific domains. More precisely, we follow the guidance of [24], which urges the use of relations to formalize the notion of solution to a given input of a problem, so that enumeration, counting, and uniform generation appear as particular problems in this formalization. In Section 2, we present this framework together with the formal notions of efficiency that we pursue for these three problems. The next step then is to provide a simple way to identify relations that have good properties in terms of these three tasks. For this, we use the concept of non-deterministic logspace transducers to provide two classes of relations, called RelationNL and RelationUL, and show that the aforementioned three problems admit efficient algorithms when restricted to these classes of relations. RelationNL and RelationUL are formally defined in Section 3, where our main results are also formally stated. Interestingly, one can show that several problems in data management are in one of these two classes. More specifically, by establishing membership in one of these two classes of relations, we show in Section 4 that problems about information extraction and binary decision diagrams, as well as our running example, admit efficient algorithms for enumeration, counting, and uniform generation.

It is important to mention that the main technical result of this work is to prove that each problem in RelationNL admits a fully polynomial-time randomized approximation scheme (FPRAS) [24] and a polynomial-time Las Vegas algorithm (with preprocessing) for uniform generation. 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 [5], it was open whether it admits an FPRAS, and only quasi polynomial time randomized approximation schemes were known for it [26, 21]. 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, that contains #P-complete problems and where each problem admits an FPRAS.

2. A UNIFYING FRAMEWORK BASED ON RELATIONS

As mentioned in the introduction, we follow a principled approach to study the problems of enumerating, counting and uniformly generating the answers to a query. We begin by following the guidance of [24], which urges the use of relations to formalize the notion of solution to a given input of a problem. Formally, if Σ denotes a finite alphabet, then we represent a problem as a relation $R \subseteq \Sigma^* \times \Sigma^*$, where, as usual, Σ^* denotes the set of all strings over Σ. For every pair $(x, y) \in R$, we interpret x as being the encoding of an input to the 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. For instance, the query answering problem from our running example in Figure 1 can be represented as the following relation:

$$\begin{aligned}\text{EVAL-PQ} = \{\, ((G,Q,n),\, \pi) \mid\, & G \text{ is a graph database,}\\ & Q \text{ is a path query, } n \in \mathbb{N},\ \pi \in Q(G), \text{ and } |\pi| = n\,\}, \quad (\dagger)\end{aligned}$$

that is, the input to the query answering problem is the triple (G, Q, n), and a solution for (G, Q, n) is a path π such that $\pi \in Q(G)$ and the length of π is n. Thus, the following is the set of solutions for (G, Q, n):

$$W_{\text{EVAL-PQ}}((G,Q,n)) = \{\, \pi \mid ((G,Q,n),\, \pi) \in \text{EVAL-PQ}\,\}.$$

This is a very general framework where any relation between input and solutions can be encoded, so we restrict to p-relations [24]. Formally, a relation $R \subseteq \Sigma^* \times \Sigma^*$ is a p-relation if (1) there exists a polynomial q such that $(x, y) \in R$ implies that $|y| = q(|x|)$ 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, $(x, y) \in R$. One can easily check that EVAL-PQ is a p-relation, as many other query answering problems in data management. Thus, considering p-relations is a natural and reasonable restriction for our framework.

The problems studied in this work can be formalized as follows in the framework presented:

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

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

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

Given that $|y| = q(|x|)$ for every $(x, y) \in R$, for a polynomial q, 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 elements of $W_R(x)$ can be enumerated in any order (but without repetitions). Moreover, in the case of COUNT(R), we assume that $|W_R(x)|$ is encoded in binary and, therefore, the size of the output is logarithmic in the size of $W_R(x)$. Finally, in the case of GEN(R), we generate a word $y \in W_R(x)$ with probability $\frac{1}{|W_R(x)|}$ if the set $W_R(x)$ is not empty; otherwise, we return a special symbol $\perp$ to indicate that $W_R(x) = \varnothing$. Hence, in our running example, the problems of enumerating, counting and uniformly generating the answers to a path query correspond to ENUM(EVAL-PQ), COUNT(EVAL-PQ), and GEN(EVAL-PQ), respectively.

In what follows, we present the notions of efficiency that we pursue for the problems studied in this work.

2.1 Notions of efficiency for enumeration

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 [25] 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 [11, 15, 33]. This notion has stronger guarantees compared to polynomial delay: after the processing of the input, the enumeration is done in a second phase taking constant-time between two consecutive outputs. Several notions of constant-delay enumeration have been studied, 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 [35]) and, for this reason, we use a more general notion of constant-delay enumeration than the ones in [11, 15, 33].

As it is standard in the literature [33], for constant-delay enumeration we consider enumeration algorithms on Random Access Machines (RAM) with addition and uniform cost measure [4]. Given a relation $R \subseteq \Sigma^* \times \Sigma^*$, an enumeration algorithm $\mathcal{E}$ for R has constant-delay if $\mathcal{E}$ 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 $\mathcal{E}$ is a constant-delay algorithm for R with precomputation time f, if $\mathcal{E}$ has constant-delay and the precomputation phase takes time $O(f(|x|))$. Moreover, we say that $\text{ENUM}(R)$ can be solved with constant-delay if there exists a constant-delay algorithm for R with precomputation time p for some polynomial p.

Our notion of constant-delay algorithm differs from the definitions in [33] in two aspects. First, as was previously mentioned, we relax the distinction between query and data in the preprocessing phase, allowing our algorithm to take polynomial time in the input (that is, we consider the combined complexity of the problem [35]). Second, our definition of constant-delay is what in [15, 11] is called *linear delay in the size of the output*, namely, writing the next output is linear in its size and does not depend on the size of the input. This is a natural assumption, since each output must at least be written down to return it to the user. Notice that, given an input x and an output y, the notion of polynomial-delay above means polynomial in $|x|$ and, instead, the notion of linear delay from [15, 11] means linear in $|y|$, i.e., constant in the size of $|x|$. Thus, we have decided to call the two-phase enumeration from above "constant-delay", as it does not depend on the size of the input x, and the delay is just what is needed to write the output (which is the minimum requirement for such an enumeration algorithm).

2.2 Notions of efficiency for counting and uniform generation

Given a relation $R \subseteq \Sigma^* \times \Sigma^*$, the problem $\text{COUNT}(R)$ 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 $\text{COUNT}(R)$ as a function that maps x to the value $|W_R(x)|$, then $\text{COUNT}(R)$ can be computed efficiently if $\text{COUNT}(R) \in \text{FP}$, the class of functions that can be computed in polynomial time. As such a condition does not hold for many fundamental problems in data management, we also consider the possibility of efficiently approximating the value of the function $\text{COUNT}(R)$. More precisely, $\text{COUNT}(R)$ is said to admit a fully polynomial-time randomized approximation scheme (FPRAS) [24] 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:

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

and the number of steps needed to compute $\mathcal{A}(x,\delta)$ is at most $q(|x|, 1/\delta)$. Thus, algorithm $\mathcal{A}(x,\delta)$ 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 $1/\delta$.

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

1. The preprocessing algorithm $\mathcal{P}$ receives as inputs x and δ and performs at most $q(|x|, \log(1/\delta))$ steps. If $W_R(x) \neq \varnothing$, then $\mathcal{P}(x,\delta)$ returns a string $\mathbf{d}$ such that $\mathbf{d} \in \mathcal{V}$ with probability $1-\delta$. If $W_R(x) = \varnothing$, then $\mathcal{P}(x,\delta)$ returns $\bot$.

2. The generator algorithm $\mathcal{G}$ receives as input $\mathbf{d}$ and performs at most $r(|\mathbf{d}|)$ steps. Moreover, if $\mathbf{d} \in \mathcal{V}$, then:

   (a) $\mathcal{G}(\mathbf{d})$ returns **fail** with a probability of at most $\frac{1}{2}$, and

   (b) conditioned on not returning **fail**, $\mathcal{G}(\mathbf{d})$ returns a truly uniform sample $y \in W_R(x)$, i.e. with a probability $1/|W_R(x)|$ for each $y \in W_R(x)$.

   Otherwise, if $\mathbf{d} \notin \mathcal{V}$, then $\mathcal{G}(\mathbf{d})$ outputs a string without any guarantee.

The set $\mathcal{V}$ of strings is called the set of *valid* strings. In line with the notion of constant-delay enumeration algorithm, we allow the previous concept of uniform generator to have a preprocessing phase. If there is no witness for the input x (that is, $W_R(x) = \varnothing$), then the preprocessing algorithm $\mathcal{P}$ returns the symbol $\bot$. Otherwise, the invocation $\mathcal{P}(x,\delta)$ returns a string $\mathbf{d}$ in Σ^*, namely, a data structure or "advice" for the generation procedure $\mathcal{G}$. The output of the invocation $\mathcal{P}(x,\delta)$ is used by the generator algorithm $\mathcal{G}$ to produce a witness of x with uniform distribution (that is, with probability $1/|W_R(x)|$). If the output of $\mathcal{P}(x,\delta)$ is not valid (which occurs with probability δ), then we have no guarantees on the output of the generator algorithm $\mathcal{G}$. Otherwise, we know that $\mathcal{G}(\mathbf{d})$ returns an element of $W_R(x)$ with uniform distribution, or it returns **fail**. Furthermore, we can repeat $\mathcal{G}(\mathbf{d})$ as many times as needed, generating each time a truly uniform sample y from $W_R(x)$ whenever $y \neq \textbf{fail}$. It is important to notice that the definition of PPLVUG does not guarantee that it can be distinguished in polynomial time whether $\mathbf{d}$ is valid (that is, whether $\mathbf{d} \in \mathcal{V}$), so $\mathcal{G}$ has to use $\mathbf{d}$ only knowing that with probability $1-\delta$ is a valid string, in which case $\mathbf{d}$ will be useful for generating an element of $W_R(x)$ with uniform distribution.

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

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

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

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

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

Notice that the notion of preprocessing polynomial-time Las Vegas uniform generator imposes stronger requirements than the notion of fully polynomial-time almost uniform generator introduced in [24]. In particular, the latter not only has a probability of failing, but also considers the possibility of generating a solution with a probability distribution that is *almost* uniform, that is, an algorithm that generates a string $y \in W_R(x)$ with a probability in an interval $[1/|W_R(x)| - \varepsilon, 1/|W_R(x)| + \varepsilon]$ for a given error $\varepsilon \in (0, 1)$.

3. OUR MAIN CONTRIBUTIONS

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 together.

3.1 The general class RelationNL

A key notion that is used in our definitions of classes of relations is that of a 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 *iff 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. This problem can be naturally represented as follows:

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

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 is in RelationNL. In fact, assume that we are given a propositional formula φ of the form $D_1 \vee \cdots \vee D_m$, where each D_i is a conjunction of literals, that is, a conjunction of propositional variables and negation of propositional variables. Moreover, assume that each propositional variable in φ is of the form x_k, where k is a binary number, and that $x\_1, \ldots, x\_n$ are the variables occurring in φ. Notice that with such a representation, we have that φ is a string over the alphabet $\{x, \_, 0, 1, \wedge, \vee, \neg\}$. We define as follows an NL-transducer M such that $M(\varphi)$ is the set of truth assignments satisfying φ. On input φ, the NL-transducer M non-deterministically chooses a disjunct D_i, which is represented by two indexes indicating the starting and ending symbols of D_i in the string φ. Then it checks whether D_i is satisfiable, that is, whether D_i does not contain complementary literals. Notice that this can be done in logarithmic space by checking for every $j \in \{1, \ldots, n\}$, whether x_j and $\neg x\_j$ are both literals in D_i. If D_i is not satisfiable, then M halts in a non-accepting state. Otherwise, M returns a satisfying truth assignment of D_i as follows. A truth assignment for φ is represented by a string of length n over the alphabet $\{0, 1\}$, where the j-th symbol of this string is the truth value assigned to variable x_j. Then for every $j \in \{1, \ldots, n\}$, if x_j is a conjunct in D_i, then M write the symbol 1 in the output tape, and if $\neg x\_j$ is a conjunct in D_i, then M write the symbol 0 in the output tape. Finally if neither x_j nor $\neg x\_j$ is a conjunct in D_i, then M non-deterministically chooses a symbol $b \in \{0, 1\}$, and it writes b in the output tape.

By using NL-transducers, one can easily show that some query answering problems in data management are also in RelationNL, in the same way as for the case of SAT-DNF. For instance, the relation EVAL-PQ defined in (†), which is used to encode our running example, can be shown to be in RelationNL. To see this, assume a reasonable encoding for an input (G, Q, n) (this time, we omit the string representation of the input and output for simplicity). In particular, assume that $Q = (a, r, b)$ is a path query with a, b vertices in G and r a regular expression. Then our NL-transducer constructs on-the-fly the product of G with an NFA A accepting the regular language defined by r, uses this product to traverse G through a path π such that π conforms to r and $|\pi| = n$, and outputs π. More precisely, our NL-transducer keeps a counter c and two indices, called g and q, pointing to a vertex in G and a state in A, respectively. The transducer starts with $c = 0$, $g = a$ and $q = q_0$, assuming that q_0 is the initial state of A. Then, at each step the machine non-deterministically chooses an edge (g, ℓ, g') on G and a transition (q, ℓ, q') on A for some edge-label ℓ, writes (g, ℓ, g') in the output tape, and updates c, g, and q to $c + 1$, g', and q', respectively. If this combination edge-transition does not exist or c becomes greater than n, then the machine stops and rejects. Instead, if it holds that g is equal to b, q is a final state of A, and $c = n$, then our NL-transducer stops and accepts. In other words, we have shown that EVAL-PQ $\in$ RelationNL.

The problem COUNT(SAT-DNF) is a paramount example of a #P-complete problem. Moreover, it is known that COUNT(EVAL-PQ) is #P-complete as well [9]. Hence, we cannot expect COUNT(R) to be solvable in polynomial time for every $R \in$ RelationNL. However, the problem COUNT(SAT-DNF) admits an FPRAS [27], so we can still hope for COUNT(R) to admit an FPRAS for every $R \in$ RelationNL. In this work, we give a positive answer to the question of the existence of such an approximation algorithm for every relation in 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 PPLVUG.*

In particular, given that EVAL-PQ is in RelationNL, the three problems for graph databases mentioned in Sections 1 and 2 have good algorithmic properties: ENUM(EVAL-PQ) can be solved with polynomial delay, COUNT(EVAL-PQ) admits an FPRAS, and GEN(EVAL-PQ) admits a PPLVUG. Notice that deriving a polynomial-delay enumeration algorithm for EVAL-PQ is straightforward, but the existence of an FPRAS for COUNT(EVAL-PQ), as well as of a PPLVUG for GEN(EVAL-PQ), was not known before. This is one of the main advantages of our approach: by proving membership in RelationNL, we can easily identify query answering problems that have good algorithmic properties in terms of enumeration, counting, and uniform generation.

3.1.1 A fundamental consequence of our result

It turns out that proving our main result (Theorem 3.2) involves providing an FPRAS for a natural problem associated to path queries and graph databases. More specifically, #NFA is the problem of counting the number of words of length k accepted by a non-deterministic finite automaton (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 [5], but it is open whether it admits an FPRAS; in fact, the best randomized approximation scheme known for #NFA runs in time $n^{O(\log(n))}$ [26]. In our notation, this problem is represented by the following relation:

$$\begin{aligned}\text{MEM-NFA} \ = \ & \{((A, 0^k), w) \mid A \text{ is an NFA and} \\ & \quad w \text{ is a word of length } k \text{ accepted by } A\},\end{aligned}$$

that is, #NFA is the same problem as COUNT(MEM-NFA). To prove Theorem 3.2, we have to provide an FPRAS for COUNT(MEM-NFA), thus giving a positive answer to the open question of whether #NFA admits an FPRAS.

It is important to mention a fundamental consequence of this result in computational complexity. The class of function SpanL was introduced in [5] 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 complexity class SpanL is contained in #P, and it is a hard class in the sense that if SpanL $\subseteq$ FP, then P = NP [5], where FP is the class of functions that can be computed in polynomial time. In fact, SpanL has been instrumental in proving that some functions are difficult to compute [5, 22, 9, 28].

It is easy to see that #NFA belongs to SpanL. In fact, it was shown in [5] that #NFA is SpanL-complete under the notion of parsimonious reduction. Therefore, given that a parsimonious reduction preserves the existence of an FPRAS, we obtain the following corollary from Theorem 3.2 and our characterization of #NFA as COUNT(MEM-NFA):

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

Although some classes $\mathcal{C}$ containing #P-complete functions and for which every $f \in \mathcal{C}$ admits an FPRAS have been identified before [32, 10], to the best of our knowledge this is the first such a class with a simple and robust definition based on Turing Machines.

3.2 The more restricted class RelationUL

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 a UL-transducer if M is an NL-transducer such that for every input x and $y \in M(x)$, there exists exactly one run of M on input x that halts in an accepting state with y as the string in the output tape. Notice that this notion of transducer is based on well-known classes of decision problems (e.g. UP [34] and UL [31]) adapted to our case, namely, adapted to problems defined as relations.

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

For the class RelationUL, we obtain the following result.

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

Hence, given a relation 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 [6], 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 such a class with a simple and robust definition based on Turing Machines.

4. OTHER APPLICATIONS OF THE MAIN RESULTS

By using our machinery, we have already proved that query evaluation in graph databases has good properties in terms of enumeration, approximate counting, and uniform generation. In this section, we show further applications of our main results in information extraction and binary decision diagrams.

4.1 Information extraction

In [16], 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 a pair $s = [i, j\rangle$ with $1 \le i \le j \le n+1$. 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 [17] to state our main results. We only recall the main definitions, and we refer the reader to [17, 16] for more intuition and further details. An eVA is a tuple $\mathcal{A} = (Q, q_0, F, \delta)$ such that Q is a finite set of states, q_0 is the initial state, and F is the final set of states. Further, δ is the transition relation consisting of letter transitions (q, a, q'), or variable-set transitions (q, S, q'), where $S \subseteq \{x\vdash, \dashv x \mid x \in \mathbf{X}\}$ and $S \neq \varnothing$. 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 $X_{i+1} \neq \varnothing$, and $q_i = p_i$ otherwise (that is, when $X_{i+1} = \varnothing$). 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 [19, 30], it was shown that the decision problem related to query evaluation, namely, given an eVA $\mathcal{A}$ and a document d deciding whether $[\![\mathcal{A}]\!](d) \neq \varnothing$, is NP-hard. For this reason, in [17] a subclass of eVA is considered in order to recover polynomial-time evaluation. An eVA $\mathcal{A}$ is called functional if every accepting run is valid. Intuitively, a functional eVA does not need to check validity of the run given that it is already known that every run that reaches a final state will be valid. For the query evaluation problem of functional eVA (i.e. to compute $[\![\mathcal{A}]\!](d)$), one can naturally associate the following relation:

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

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

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

In [17], it was shown that every functional RGX or functional VA (not necessarily extended) can be converted in polynomial time into a functional eVA. Therefore, Corollary 4.1 also holds for these more general classes.

Regarding efficient enumeration and exact counting, an algorithm for constant-delay enumeration was given in [17] for the class of deterministic functional eVA. Here, we can 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 [17] for the unambiguous case.

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

4.2 Binary decision diagrams

Binary decision diagrams are an abstract representation of boolean functions which are widely used in computer science and have found many applications in areas like formal verification [13]. 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 (OBDDs), we also have a linear order $<$ over the variables in D such that, for every $v_1, v_2 \in V$ with v_2 a child of v_1, it holds that $\text{var}(v_1) < \text{var}(v_2)$. Note that not all variables need to 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 use of 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.

Corollary 4.3. *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. However, they show how easy and direct is to use UL-transducers to obtain some of the good algorithmic properties of OBDDs.

Some non-deterministic variants of BDDs have been studied [7]. In particular, an nOBDD extends an OBDD with vertices u without variables (i.e. $\text{var}(u) = \bot$) and without labels on its children. Thus, an nOBDD is non-deterministic in the sense that given an assignment σ, there can be several paths that bring σ from the initial 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 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 is in RELATIONNL, from which the following results follow.

Corollary 4.4. *ENUM(EVAL-nOBDD) can be solved with polynomial delay, COUNT(EVAL-nOBDD) admits an FPRAS, and GEN(EVAL-nOBDD) admits a PPLVUG.*

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

5. CONCLUDING REMARKS

We consider this work as a first step towards the definition of classes of problems in data management with good properties in terms of enumeration, counting, and uniform generation of solutions. Given the relevance of these problems for query answering, identifying good complexity classes, like RELATIONNL and RELATIONUL, should be the cornerstone to better understand the complexity of query evaluation. In this sense, there is plenty of room for extensions and improvements. In particular, one could be more ambitious and ask for more conditions to these relations, like having good properties in terms of ranked enumeration [33] (i.e. enumeration of the solutions following some specific order) or random generation with respect to a user-defined distribution. Moreover, we believe that other classes with good algorithmic properties can be identified, which could serve to unify enumeration, counting, and uniform generation in data management.

6. REFERENCES

[1] S. Abiteboul and G. Dowek. *The Age of Algorithms.* Cambridge University Press, 2020.

[2] S. Abiteboul, G. Miklau, J. Stoyanovich, and G. Weikum. Data, responsibly. *Dagstuhl Reports*, 6(7):42–71, 2016.

[3] S. Acharya, P. B. Gibbons, V. Poosala, and S. Ramaswamy. Join synopses for approximate query answering. In *SIGMOD*, pages 275–286, 1999.

[4] A. V. Aho and J. E. Hopcroft. *The design and analysis of computer algorithms.* Pearson Education India, 1974.

[5] C. Álvarez and B. Jenner. A very hard log-space counting class. *Theoretical Computer Science*, 107(1):3–30, 1993.

[6] 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.

[7] A. Amarilli, F. Capelli, M. Monet, and P. Senellart. Connecting knowledge compilation classes and width parameters. *CoRR*, abs/1811.02944, 2018.

[8] R. Angles, M. Arenas, P. Barceló, A. Hogan, J. Reutter, and D. Vrgoč. Foundations of modern query languages for graph databases. *ACM Computing Surveys*, 50(5):68, 2017.

[9] 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 *WWW*, pages 629–638, 2012.

[10] M. Arenas, M. Muñoz, and C. Riveros. Descriptive complexity for counting complexity classes. In *LICS*, pages 1–12, 2017.

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

[12] G. Bagan, A. Durand, and E. Grandjean. On acyclic conjunctive queries and constant delay enumeration. In *Proceedings of CSL*, pages 208–222, 2007.

[13] R. E. Bryant. Symbolic boolean manipulation with ordered binary-decision diagrams. *ACM Computing Surveys (CSUR)*, 24(3):293–318, 1992.

[14] S. Chaudhuri, R. Motwani, and V. R. Narasayya. On random sampling over joins. In *SIGMOD*, pages 263–274, 1999.

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

[16] 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.

[17] F. Florenzano, C. Riveros, M. Ugarte, S. Vansummeren, and D. Vrgoc. Efficient enumeration algorithms for regular document spanners. *ACM Trans. Database Syst.*, 45(1):3:1–3:42, 2020.

[18] N. Francis, A. Green, P. Guagliardo, L. Libkin, T. Lindaaker, V. Marsault, S. Plantikow, M. Rydberg, P. Selmer, and A. Taylor. Cypher: An evolving query language for property graphs. In *Proceedings of SIGMOD*, pages 1433–1445, 2018.

[19] D. D. Freydenberger. A logic for document spanners. In *Proceedings of ICDT*, pages 13:1–13:18, 2017.

[20] H. Garcia-Molina, J. D. Ullman, and J. Widom. *Database systems - the complete book.* Pearson Education, 2009.

[21] 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.

[22] L. A. Hemaspaandra and H. Vollmer. The satanic notations: counting classes beyond #P and other definitional adventures. *SIGACT News*, 26(1):2–13, 1995.

[23] Y. E. Ioannidis. The history of histograms (abridged). In *VLDB*, pages 19–30. Morgan Kaufmann, 2003.

[24] 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.

[25] D. S. Johnson, M. Yannakakis, and C. H. Papadimitriou. On generating all maximal independent sets. *Information Processing Letters*, 27(3):119–123, 1988.

[26] 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.

[27] R. M. Karp and M. Luby. Monte-carlo algorithms for enumeration and reliability problems. In *Proceedings of FOCS*, pages 56–64, 1983.

[28] 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.

[29] W. Martens and T. Trautner. Dichotomies for evaluating simple regular path queries. *ACM Trans. Database Syst.*, 44(4):16:1–16:46, 2019.

[30] 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.

[31] K. Reinhardt and E. Allender. Making nondeterminism unambiguous. *SIAM J. Comput.*, 29(4):1118–1131, 2000.

[32] S. Saluja, K. Subrahmanyam, and M. N. Thakur. Descriptive complexity of #P functions. *Journal of Computer and System Sciences*, 50(3):493–505, 1995.

[33] L. Segoufin. Enumerating with constant delay the answers to a query. In *Proceedings of ICDT*, pages 10–20, 2013.

[34] L. G. Valiant. Relative complexity of checking and evaluating. *Inf. Process. Lett.*, 5(1):20–23, 1976.

[35] M. Y. Vardi. The complexity of relational query languages (extended abstract). In *STOC*, pages 137–146, 1982.

[36] Z. Zhao, R. Christensen, F. Li, X. Hu, and K. Yi. Random sampling over joins revisited. In *SIGMOD*, pages 1525–1539. ACM, 2018.