Notes on Efficient Enumeration, Counting, and Uniform Generation of Logspace Classes

Richard Willie

March 16, 2025

Preface

These notes are simply a summary of [Ar+21] for my own reference. They are *not* endorsed by the authors, and I strongly recommend reading the original work. Any errors here are almost certainly mine.

Contents

1 Preliminaries

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

§1.1 Relations and Problems

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

This is a very general framework, so we work with ***p*-relations** (see [JeVaVa86]).

Definition 1.1.1 (p-relation)
Formally, a relation $R \subseteq \{0,1\}^* \times \{0,1\}^*$ is a *p-relation* if

1. there exists a polynomial q such that $(x, y) \in R$ implies that $|y| \leq q(|x|)$ and
2. there exists a deterministic Turing Machine (TM) that receives as input $(x, y) \in \{0,1\}^* \times \{0,1\}^*$ runs in polynomial time, and accepts if, and only if, $(x, y) \in R$. In other words, the TM is a verifier that accepts as input an instance of a problem x and its witness y.

Without loss of generality, from now on we assume that for a p-relation in R, there exists a polynomial q such that $|y| = q(|x|)$ for every $(x, y) \in R$. This is not a strong requirement, since all solutions can be made to have the same length through padding.

§1.2 Enumeration, Counting, and Uniform Generation

Given a p-relation R, we are interested in the following problems:

Definition 1.2.1 (ENUM)

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

Definition 1.2.2 (COUNT)

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

Definition 1.2.3 (GEN)

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

Given that $|y| = q(|x|)$ for every $(x, y) \in R$, we have that $W_R(x)$ is finite and these three problems are well defined. Notice that in the case of ENUM(R), we do not assume a specific order on words, so the elements of $W_R(x)$ can be enumerated in any order (but without repetitions). Moreover, in the case 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 $W_R(x) = \varnothing$.

§1.3 Enumeration with Polynomial and Constant Delay

An enumeration algorithm for ENUM(R) is a procedure that receives an input $x \in \{0,1\}^*$ and, during the computation, it outputs each word in $W_R(x)$, one-by-one and without repetitions. The time between two consecutive outputs is called the delay of the enumeration. 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 (see [**JoYaPa88**]) and is defined as follows.

Definition 1.3.1 (Polynomial delay enumeration)

An enumeration algorithm is of *polynomial delay* if there exists a polynomial p such that for every input $x \in \{0,1\}^*$, the time between the beginning of the algorithm and the initial output, between any two consecutive outputs, and between the last output and end of the algorithm, is bounded by $p(|x|)$.

Constant delay enumeration is another notion of efficiency for enumeration algorithms that has attracted a lot attention in recent years (see [**Ba06**; **Co09**; **Se13**]). This notion has stronger guarantees compared to polynomial delay: The enumeration done in a second phase after the processing of the input and taking constant-time between two consecutive outputs in a very precise sense. Several notions of constant delay enumeration have been given, most of them in database theory where it is important to divide the analysis between query and data. Here, 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 [**Ba06**; **Co09**; **Se13**].

Constant delay enumeration cannot be achieved in general with a standard Turing Machine, because merely moving the head through the tape will take up more than constant time. So, as it is standard in the literature [Se13], for the notion of constant delay enumeration, we consider enumeration algorithms on *Random Access Machines (RAM)* with addition and uniform cost measure (see [AhHo74]).

Definition 1.3.2 (Constant delay enumeration)
Given a relation $R \subseteq \{0,1\}^* \times \{0,1\}^*$, an enumeration algorithm E for R has *constant delay* if E runs in two phases over the input x.

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

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

Our definition of constant delay algorithm differ from the definitions in [Se13] in two aspects.

1. First, in our definition the input is not divided into some components, so the preprocessing phase must take polynomial time in the size of the entire input.
2. Second, our definition of constant delay is what in [Ba06; Co09] 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 (see Definition 1.3.1) means polynomial in $|x|$ and, instead, the notion of linear delay from [Ba06; Co09] 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).

§1.4 Approximate Counting and Las Vegas Uniform Generation with Preprocessing

Given a relation $R \subseteq \{0,1\}^* \times \{0,1\}^*$, the problem $\text{COUNT}(R)$ can be solved efficiently if there exists a polynomial-time algorithm that, given $x \in \{0,1\}^*$, computes $|W_R(x)|$. In other words, if we think of COUNT(R) as a function that maps x to the value $|W_R(x)|$, then COUNT(R) can be computed efficiently if COUNT$(R) \in \mathbf{FP}$, 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).

Definition 1.4.1 (FPRAS)

More precisely, COUNT(R) is said to admit an **FPRAS** (see [**JeVaVa86**]) if there exists a randomized algorithm $\mathcal{A} : \{0,1\}^* \times (0,1) \to \mathbb{N}$ and a polynomial $q(u,v)$ such that for every $x \in \{0,1\}^*$ and $\delta \in (0,1)$, it holds that:

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

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

The problem GEN(R) can be solved efficiently if there exists a polynomial-time randomized algorithm such that, given $x \in \{0,1\}^*$, generates an element of $|W_R(x)|$ with uniform probability distribution (if $W_R(x) = \varnothing$, then it returns $\perp$). However, as in the case of COUNT(R), the existence of such a generator is not guaranteed for many fundamental problems, so we also consider a relaxed notion of generation that has a probability of failing in returning a solution.

Definition 1.4.2 (PPLVUG)

More precisely, GEN(R) is said to admit a **preprocessing polynomial-time Las Vegas uniform generator (PPLVUG)** if there exists a pair of randomized algorithms $\mathcal{P} : \{0,1\}^* \times (0,1) \to (\{0,1\}^* \cup \{\perp\})$, $\mathcal{G} : \{0,1\}^* \to (\{0,1\}^* \cup \{\textbf{fail}\})$ and a pair of polynomials $q(u,v)$, $r(u)$ such that for every $x \in \{0,1\}^*$ and $\delta \in (0,1)$:

1. The preprocessing algorithm $\mathcal{P}$ receives as inputs x and δ and runs in time bounded by $q(|x|, \log(1/\delta))$. If $W_R(x) \neq \varnothing$, $\mathcal{P}(x,\delta)$ returns a string $\mathcal{D}$ such that $\mathcal{D}$ is **good-for-generation** with probability $1-\delta$. If $W_R(x) = \varnothing$, then $\mathcal{P}(x,\delta)$ returns $\perp$.
2. The generator algorithm $\mathcal{G}$ receives as input $\mathcal{D}$ and runs in time bounded by $r(|\mathcal{D}|)$. Moreover, if $\mathcal{D}$ is good-for-generation, then:
   a) $\mathcal{G}(\mathcal{D})$ returns **fail** with a probability of at most $\frac{1}{2}$, and
   b) conditioned on not returning **fail**, $\mathcal{G}(\mathcal{D})$ returns a truly uniform sample $y \in W_R(x)$, i.e. with a probability $\frac{1}{|W_R(x)|}$ for each $y \in W_R(x)$.

   Otherwise, if $\mathcal{D}$ is not good-for-generation, then $\mathcal{G}(\mathcal{D})$ outputs a string without any guarantee.

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

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

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

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

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

It is important to notice that the notion of preprocessing polynomial-time Las Vegas uniform generator imposes stronger requirements than the notion of fully polynomial-time almost uniform generator introduced in [**JeVaVa86**]. 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 $[1/|W_R(x)| - \epsilon, 1/|W_R(x)| + \epsilon]$ for a given error $\epsilon \in (0, 1)$.

2 NLOGSPACE Transducers: Definitions and Main Results

Definition 2.0.1 (**NL**-transducer)

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

Definition 2.0.2 (**RelationNL**)

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

Cycles are forbidden in **NL**-transducers to ensure polynomial-size solutions for each input (see [ÀlJe93]). However, we do not need to impose this restriction here, as we only work with p-relations.

I don't get it.

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

Definition 2.0.3 (SAT-DNF)

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

Fix this cosmetic bug.

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.

Lemma 2.0.1

SAT-DNF $\in$ **RelationNL**.

Proof. 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 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 a set of truth assignments satisfying φ. On input φ, the **NL**-transducer M nondeterministically chooses a disjunct D_i. 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 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 $j \in \{0, \ldots, 1\}$, if x_j is a conjunct in D_i, then M writes the symbol 1 to the output tape, and if $\neg x_j$ is a conjunct in D_i, then M writes the symbol 0 in the output tape. Finally, if neither x_j nor $\neg x_j$ is a conjunct in D_i, then M nondeterministically writes the symbol 0 or 1 in the output tape. $\square$

Given that COUNT(SAT-DNF) is a **#P**-complete problem (see [**PrBa83**]), we cannot expect COUNT(R) to be solvable in polynomial time for every $R \in$ **RelationNL**. However, COUNT(SAT-DNF) admits an FPRAS (see [**KaLu83**]), 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 nondeterministic 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 (see [**ÀlJe93**]), but it is open whether it admits an FPRAS; in fact, the best randomized approximation scheme known for #NFA runs in time $n^{O(\log(n))}$ (see [**KaSwMa95**]). In our notation, this problem is represented by the following relation:

Definition 2.0.4 (MEM-NFA)

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

Fix this cosmetic bug.

that is, we have

Definition 2.0.5 (#NFA)
#NFA = COUNT(MEM-NFA).

It is easy to see that

Lemma 2.0.2
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 2.0.3

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

Proof. Refer to Chapter 5. □

It is worth mentioning a fundamental consequence of this result in computational complexity. The class of functions **SpanL** was introduced in [ÀlJe93] to provide a characterization of some functions that are hard to compute.

Definition 2.0.6 (SpanL)

More specifically, a function $f : \{0,1\}^* \to \mathbb{N}$ is in **SpanL** if there exists an **NL**-transducer M with input alphabet $\{0,1\}$ such that $f(x) = |M(x)|$ for every $x \in \{0,1\}^*$.

The complexity class **SpanL** is contained in **#P**, and it is a hard class in the sense that if $\mathbf{SpanL} \subseteq \mathbf{FP}$, then $\mathbf{P} = \mathbf{NP}$ (see [ÀlJe93]), 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 (see [ÀlJe93; ArCoPé12; HeVo95; LoMa13]).

Lemma 2.0.4

#NFA belongs to **SpanL**.

Proof. This is a corollary of Lemma 2.0.2 (see [ÀlJe93]). □

Definition 2.0.7 (Parsimonious reduction)

Given functions $f, g : \{0,1\}^* \to \mathbb{N}$, f is said to be **parsimoniously reducible** to g in polynomial time if there exists a polynomial-time computable function $h : \{0,1\}^* \to \{0,1\}^*$ such that, for every $x \in \{0,1\}^*$, it holds that $f(x) = g(h(x))$.

It is known that #NFA is **SpanL**-complete under polynomial-time parsimonious reductions, which in particular implies that if #NFA can be computed in polynomial time, then $\mathbf{P} = \mathbf{NP}$ (see [ÀlJe93]). Moreover, given that #NFA admits an FPRAS and parsimonious reductions preserve the existence of FPRAS, we obtain the following corollary from Theorem 2.0.3:

Corollary 2.0.5

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

A natural question at this point is whether a simple syntactic restriction on the definition of **RelationNL** gives rise to a class of relations with better properties in terms of enumeration, counting, and uniform generation. Fortunately, the answer to this question comes by imposing a natural and well-studied restriction on Turing Machines, which allows the definition of a class that contains many natural problems. More precisely, we consider the notion of **UL-transducer**, where the letter "U" stands for "unambiguous".

Definition 2.0.8 (UL-transducer)

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** (see [Va76]) and **UL** (see [ReAl00])), adapted to our case, namely, adapted to problems defined as relations.

Definition 2.0.9 (RelationUL)

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

For the class **RelationUL**, we obtain the following result:

Theorem 2.0.6

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).

*On the relationship of **RelationNL** and **RelationUL** with known complexity classes.* It is well-known that a function f is in **#P** if and only if there exists a p-relation R such that $f =$ COUNT(R) (recall the definition of p-relation from Section 1.1). In the same way, there exists a tight relationship between **SpanL** and **RelationNL**, as it is easy to see that a function f is in **SpanL** if and only if there exists a relation $R \in$ **RelationNL** such that $f =$ COUNT(R). Hence, one may wonder why it is necessary to introduce **RelationNL** and **RelationUL**, considering further that such classes are defined in terms of well-known Turing Machine models. The key issue to consider here is that function complexity classes, such as **#P** and **SpanL**, are not appropriate to state results about the enumeration and uniform generation problems. For instance, it would not be correct to state that every function in **SpanL** admits a PPLVUG, as the definition of **SpanL** does not provide a unique notion of solution for an input, which is the object to be generated in this case. In this respect, we introduce **RelationNL** and **RelationUL** to have a unified framework to study the counting, enumeration, and uniform generation problems. The definition of such classes should only be seen as our way of following the guidance of [JeVaVa86], which, as mentioned before, urges the use of relations to formalize the notion of solution for an input of a problem.

3 Applications of the Main Results

4 Completeness, Self-Reducibility, and Their Implications for the Class RelationNL

The goal of this chapter is to establish the good algorithmic properties of **RelationUL**, that is, to prove Theorem 2.0.6. To this end, we start by introducing a simple notion of reduction for the classes **RelationNL** and **RelationUL**, which allow for much simpler proofs. A natural question to ask is which notions of "completeness" and "reduction" are appropriate for our framework. Let $\mathcal{C}$ be a complexity class of relations $R, S \in \mathcal{C}$, and recall that $W_R(x)$ is defined as the set of solutions for input x, that is, $W_R(x) = \{y \mid (x, y) \in R\}$.

Definition 4.0.1 (Reduction and completeness)
We say that R is *reducible* to S if there exists a function $f : \{0,1\}^* \to \{0,1\}^*$, computable in polynomial time, such that for every $x \in \{0,1\}^*$: $W_R(x) = W_S(f(x))$. Also, if T is reducible to S for every $T \in \mathcal{C}$, then we say S is *complete* for $\mathcal{C}$.

Notice that this definition is very restricted, since the notion of reduction requires the set of solutions to be exactly the same for both relations (it is not sufficient that they have the same size, for example). The benefit of this kind of reduction is that it preserves all the properties of efficient enumeration, counting, and uniform generation that we introduced in Chapter 1 and Chapter 2, as stated in the following proposition.

Proposition 4.0.1
If a relation R can be reduced to a relation S, then:

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

Therefore, by finding a complete relation S for a class $\mathcal{C}$ under the notion of reduction just defined, we can study the aforementioned problems for S knowing that the obtained results will extend to every relation in class $\mathcal{C}$. In what follows, we identify complete problems for the classes **RelationNL** and **RelationUL** and use them first to establish the good algorithmic properties of **RelationUL**. Moreover, we prove that the identified problems are **self-reducible** (see [JeVaVa86]), which will be useful in establishing some of the results of this section as well as for some of the results proved in Chapter 5 for the class **RelationNL**.

§4.1 Complete Problems for RelationNL and RelationUL

The notion of reduction just defined is useful for us, because **RelationNL** and **RelationUL** admit natural complete problems under this notion. These complete problems are defined in terms of NFAs and we call them MEM-NFA and MEM-UFA. We already introduced MEM-NFA in Chapter 2, and we now define MEM-UFA as

Definition 4.1.1 (MEM-UFA)

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

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

Fix this cosmetic bug.

Recall from Chapter 2 that MEM-NFA $\in$ **RelationNL**. Besides, it is easy to see that

Lemma 4.1.1
MEM-UFA $\in$ **RelationUL**.

We now state the main result of this section.

Proposition 4.1.2
MEM-NFA is *complete* for **RelationNL** and MEM-UFA is *complete* for **RelationUL**.

We will prove the result only for the case of **RelationUL** and MEM-UFA, as the other case is completely analogous. The following lemma is the key ingredient to our argument.

Lemma 4.1.3
Let R be a relation in **RelationUL**. Then there exists a polynomial-time algorithm that, given $x \in \{0,1\}^*$, produces an unambiguous NFA A_x such that $y \in W_R(x)$ if and only if y is accepted by A_x.

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

Let $n = |x|$, $f(n)$ be the function that bounds the number of cells in the work tape that can be used, and assume that $f(n)$ is $O(log(n))$. Consider now an execution of

M on input x. Since the input tape never changes (its content is always x), we can completely characterize the configuration of the machine at any given moment as a tuple $(q, i, j, w) \in Q \times \{1, \ldots, n\} \times \{1, \ldots, f(n)\} \times \Gamma^{f(n)}$ where

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

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

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

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

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

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

- $c_0 = c_I$.
- $c_m = c_F$.
- For each $k \in \{0, \ldots, m-1\}$, the transition from c_k to c_{k+1} is valid on input x given the transition relation δ on M.

- For each $k \in \{0, \dots, m-1\}$, we have that w_k is equal to the symbol output when going from configurations c_k to c_{k+1} if a symbol was output. Otherwise, $w_k = \epsilon$.
- $y = w_0 \cdot w_1 \cdot \ldots \cdot w_m$.

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

Finally, notice that A_x is actually not an NFA (under the definition given in Chapter 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 (see [Ho01]). This concludes the proof of the lemma. □

We now prove Proposition 4.1.2.

Proof. Let R be a relation in **RelationUL** and x be a string in $\{0,1\}^*$. We know by Lemma 4.1.3 that we can construct in polynomial time an unambiguous NFA A_x such that $y \in W_R(x)$ if and and only if y is accepted by A_x. Now, since R is a p-relation, there exists a polynomial q such that $|y| = q(|x|)$ for all $y \in W_R(x)$. Thus, we have that all words accepted by A_x have the same length $q(|x|)$. We conclude that $W_R(x) = W_{\text{MEM-UFA}}((A_x, 0^{q(|x|)}))$. Since this works for every $R \in$ **RelationUL** and every input x, by definition of completeness, we deduce that MEM-UFA is complete for **RelationUL**. □

§4.2 MEM-NFA and MEM-UFA are Self-Reducible

Self-reducibility is a property of many natural relations, and it plays a key role in proving some important results, like the tight relationship between counting and uniform generation established in [JeVaVa86]. There are different ways of formalizing this concept, and they can get rather technical, but the intuition is pretty straightforward.

Definition 4.2.1 (Self-reducibility, informal version)

We say that a (decision) problem is *self-reducible*, if it can be solved by referring to smaller instances of the same problem.

Example 4.2.1

SAT is self-reducible. Given a propositional formula φ, consider its satisfiability problem. We can easily reduce that problem to smaller instances of SAT as follows. Take the first variable of φ and replace it by 0 to set a new formula φ_0. Do the same with 1 to get a new formula φ_1. Notice that φ is satisfiable if and only if φ_0 or φ_1 is satisfiable. Moreover, both φ_0 and φ_1 have one less variable then φ, so they are smaller instances.

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

Definitions (and proofs) of self-reducibility can get very technical, partly because they have to formalize the notion of "smaller instance". Hence, they crucially depend on the way that the problems are encoded.[1] We now state the main result of this section.

Proposition 4.2.2

MEM-NFA and MEM-UFA are self-reducible.

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

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

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

§4.3 Establishing the Good Algorithmic Properties of RelationUL

Theorem 2.0.6 is a consequence of Proposition 4.0.1, Proposition 4.1.2, and the following result:

Proposition 4.3.1

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

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

To sum up all the results just mentioned: MEM-UFA is complete for **RelationUL**, it has good algorithmic properties, and our notion of reduction (and completeness) preserves all the algorithmic properties we have discussed. In what follows, we prove each of the three results stated in Proposition 4.3.1.

§4.3.1 ENUM(MEM-UFA) Can Be Solved with Constant Delay

We now provide a sketch for the constant delay algorithm. The idea is conceptually simple. Remember what we want: to output all strings of a certain length accepted by an unambiguous NFA, without repetition. We may use a preprocessing phase of polynomial time, but afterwards, there can be at most linear time between one string and the next.

§4.3.2 There Exists a Polynomial-Time Algorithm for COUNT(MEM-UFA)

§4.3.3 There Exists a Polynomial-Time Randomized Algorithm for GEN(MEM-UFA)

5 #NFA Admits a Fully Polynomial-Time Randomized Approximation Scheme and Its Implications to the Class RelationNL

The goal of this chapter is to prove Theorem 2.0.3, which considers the class **RelationNL** defined in terms of **NL**-transducers. Given that we showed in Proposition 4.1.2 that MEM-NFA is complete for **RelationNL**, we have by Proposition 4.0.1 that Theorem 2.0.3 is a consequence of the following result:

Theorem 5.0.1

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

The existence problem for MEM-NFA has as input an NFA A and a value k given in unary (as the string 0^k), the question to answer is whether $W_{\text{MEM-NFA}}((A, 0^k)) \neq \varnothing$ (that is, whether there are any solutions for $(A, 0^k)$ according to the relation MEM-NFA). It is easy to prove that such a task can be solved in polynomial time, as the nonemptiness problem for NFA can be solved in polynomial time (see [Ho01]). Moreover, we proved in Proposition 4.2.2 that MEM-NFA is a self-reducible relation. Given all the above, a polynomial delay algorithm for ENUM(MEM-NFA) can be derived from the folklore result that such an enumeration algorithm exists for a self-reducible relation if the associated existence problem for this relation can be solved in polynomial time (a precise statement of this result can be found in Lemma 4.10 in [Sc09]).

In this chapter, we focus on the remaining part of the proof of Theorem 5.0.1. More specifically, we provide an algorithm that approximately counts the number of words of a given length accepted by an NFA, where this length is given in unary. This constitutes an FPRAS for COUNT(MEM-NFA), as formally stated in the following theorem:

Theorem 5.0.2

#NFA (and, thus, COUNT(MEM-NFA)) admits an FPRAS.

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

Theorem 5.0.3

GEN(MEM-NFA) admits a PPLVUG.

In the rest of this chapter, we prove Theorem 5.0.2 and Theorem 5.0.3. More specifically, we start by providing in Section 5.1 an overview of the algorithmic techniques used in the proof of Theorem 5.0.2. Then, we present in Section 5.2 the template for the FPRAS for #NFA, whose main components are given in Section 5.3 and Section 5.4. A complete version of the FPRAS for #NFA is finally given in Section 5.6, where its correctness and polynomial-time complexity are established. Moreover, the proof of Theorem 5.0.3 is also given in Section 5.6.

§5.1 An Overview of the Algorithmic Techniques

§5.2 The Algorithm Template

§5.3 Computing an Estimate for a Set of Vertices

§5.4 Uniform Sampling from a Vertex

§5.5 Bounding the Probability of Breaking the Main Assumption

§5.6 The Main Algorithm, Its Correctness, and Its Complexity

Bibliography

[AhHo74] Alfred V Aho and John E Hopcroft. *The design and analysis of computer algorithms*. Pearson Education India, 1974 (cited p. 6)

[ÀlJe93] Carme Àlvarez and Birgit Jenner. A very hard log-space counting class. In: *Theoretical Computer Science*, **107**:1 (1993), pp. 3–30 (cited pp. 9–11)

[Ar+21] Marcelo Arenas et al. # NFA Admits an FPRAS: Efficient Enumeration, Counting, and Uniform Generation for Logspace Classes. In: *Journal of the ACM (JACM)*, **68**:6 (2021), pp. 1–40 (cited p. 2)

[ArCoPé12] Marcelo Arenas, Sebastián Conca, and Jorge Pérez. “Counting beyond a Yottabyte, or how SPARQL 1.1 property paths will prevent adoption of the standard”. In: *Proceedings of the 21st international conference on World Wide Web*. 2012, pp. 629–638 (cited p. 11)

[Ba06] Guillaume Bagan. “MSO queries on tree decomposable structures are computable with linear delay”. In: *International Workshop on Computer Science Logic*. Springer. 2006, pp. 167–181 (cited pp. 5, 6)

[Co09] Bruno Courcelle. Linear delay enumeration and monadic second-order logic. In: *Discrete Applied Mathematics*, **157**:12 (2009), pp. 2675–2700 (cited pp. 5, 6)

[HeVo95] Lane A Hemaspaandra and Heribert Vollmer. The satanic notations: counting classes beyond# P and other definitional adventures. In: *ACM SIGACT News*, **26**:1 (1995), pp. 2–13 (cited p. 11)

[Ho01] JE Hopcroft. *Introduction to Automata Theory, Languages, and Computation*. 2001 (cited pp. 17, 20)

[JeVaVa86] Mark R Jerrum, Leslie G Valiant, and Vijay V Vazirani. Random generation of combinatorial structures from a uniform distribution. In: *Theoretical computer science*, **43**: (1986), pp. 169–188 (cited pp. 4, 7, 8, 12, 14, 17)

[JoYaPa88] David S Johnson, Mihalis Yannakakis, and Christos H Papadimitriou. On generating all maximal independent sets. In: *Information Processing Letters*, **27**:3 (1988), pp. 119–123 (cited p. 5)

[KaLu83] Richard M Karp and Michael Luby. “Monte-Carlo algorithms for enumeration and reliability problems”. In: *24th Annual Symposium on Foundations of Computer Science (sfcs 1983)*. IEEE Computer Society. 1983, pp. 56–64 (cited p. 10)

[KaSwMa95] Sampath Kannan, Z Sweedyk, and Steve Mahaney. “Counting and random generation of strings in regular languages”. In: *Proceedings of the sixth annual ACM-SIAM symposium on Discrete algorithms*. 1995, pp. 551–557 (cited p. 10)

[LoMa13] Katja Losemann and Wim Martens. The complexity of regular expressions and property paths in SPARQL. In: *ACM Transactions on Database Systems (TODS)*, **38**:4 (2013), pp. 1–39 (cited p. 11)

[**PrBa83**] J Scott Provan and Michael O Ball. The complexity of counting cuts and of computing the probability that a graph is connected. In: *SIAM Journal on Computing*, **12**:4 (1983), pp. 777–788 (cited p. 10)

[**ReAl00**] Klaus Reinhardt and Eric Allender. Making nondeterminism unambiguous. In: *SIAM Journal on Computing*, **29**:4 (2000), pp. 1118–1131 (cited p. 12)

[**Sc09**] Johannes Schmidt. Enumeration: Algorithms and complexity. In: *Preprint, available at https://www. thi. uni-hannover. de/fileadmin/forschung/arbeiten/schmidt-da. pdf*, (2009) (cited p. 20)

[**Se13**] Luc Segoufin. “Enumerating with constant delay the answers to a query”. In: *Proceedings of the 16th International Conference on Database Theory*. 2013, pp. 10–20 (cited pp. 5, 6)

[**Va76**] Leslie G Valiant. Relative complexity of checking and evaluating. In: *Information processing letters*, **5**:1 (1976), pp. 20–23 (cited p. 12)