On the Complexity of Enumeration

Florent Capelli[1] and Yann Strozecki[2]

1 **Birkbeck University, London**
2 **Université de Versailles Saint-Quentin-en-Yvelines, DAVID laboratory**

Abstract

We investigate the relationship between several enumeration complexity classes and focus in particular on the incremental polynomial time and the polynomial delay (IncP and DelayP). We prove, modulo the Exponential Time Hypothesis, that IncP contains a strict hierarchy of subclasses. Since DelayP is included in IncP_1, the first class of the hierarchy, it is separated from IncP. We prove for some algorithms that we can turn an average delay into a worst case delay, suggesting that $\mathrm{IncP}_1 = \mathrm{DelayP}$ even with a polynomially bounded memory. Finally we relate the uniform generation of solutions to probabilistic enumeration algorithms with polynomial delay.

1998 ACM Subject Classification F.2.2 Nonnumerical Algorithms and Problems

Keywords and phrases enumeration, incremental time, polynomial delay, classes separation, exponential time hypothesis

Digital Object Identifier 10.4230/LIPIcs...

1 Introduction

An enumeration problem is the task of listing a set of elements, often the solutions of some problem, such as the spanning trees of a graph or the satisfying assignments of a formula. One way to measure the complexity of an enumeration algorithm is the *total time* used to compute all solutions. The number of solutions may be exponential in the size of the input, therefore a problem is tractable and said to be *output polynomial* when it can be solved in polynomial time in the size of the *input and the output*. This measure is relevant when one wants to generate and store all elements, for instance to constitute a library of interesting objects, as it is often done in biology or chemistry [4].

Another application of enumeration algorithms is to compute an optimal solution by generating all solutions. It can also be used to compute statistics on the set of solutions or its cardinality. If the set of solutions is too large, it can be interesting to generate only a fraction of the solutions. A good algorithm must then guarantee that if it is given more time, it will find more solutions. *Polynomial incremental time* algorithms are defined as follows: the time between the enumeration of the k^{th} and $(k+1)^{\text{th}}$ solutions is polynomial in k and in the size of the input. Many problems which can be solved with an incremental delay are of the following form: given a set of elements and a polynomial time function acting on tuples of elements, produce the closure of the set by the function. For instance, the best algorithm to generate all circuits of a matroid is in polynomial incremental time because it uses some closure property of the circuits [17]. The fundamental problem of generating the minimal transversals of a hypergraph can also be solved in subexponential incremental time [12] and some of its restrictions in polynomial incremental time [9].

Most efficient enumeration algorithms have a *delay* between consecutive solutions bounded by a polynomial in the input. *Polynomial delay* algorithms produces solutions regularly and need a time linear in the number of solutions, which can still be overall exponential if the set of solutions is large. There are two main polynomial delay methods, the *backtrack search*

and the *reverse search*, which have been used to list the cycles of a graph [20], the satisfying assignments of variants of SAT [7], the spanning trees, the connected induced subgraphs of a graph [2] etc. These methods only use a *polynomial space*, which is required in practice. Another way to approach enumeration problems, while using only polynomial space, is to build random generators of solutions, a very active area of research [8]. We explore this direction in this paper by giving a deeper understanding of the relation between enumeration and random generators.

Enumeration algorithms have been studied for the last 40 years [20] and the notions of incremental polynomial time and polynomial delay already appear in [15], but very little is known on the structural complexity of enumeration, which seems harder to formalize than decision or counting complexity. Most previous results on structural complexity appear in the thesis of the second author [23], and definitions of classes, reductions and proofs of closure of classes by operations or reductions can also be found in [3, 5, 18]. One of the main difficulties is that complete problems are known only for EnumP, the equivalent of NP in enumeration, but not for the other natural classes. In this paper, we therefore focus on understanding these classes by using classical hypotheses in complexity theory. Observe that we do not impose the order on the generation of solutions, such a requirement makes enumeration complexity similar to decision complexity but it is also more artificial. We also ask the generated solutions to be checkable in polynomial time, a reasonable hypothesis which makes separation of classes much harder.

The paper is organized as follows: Sec. 2 is dedicated to the definition of the complexity classes either with polynomial time checkable solutions or not. We use classical complexity hypotheses to prove separation between most classes. These results are not completely new and can be found in the thesis of the second author [23], in the master's thesis of Johannes Schmidt [22] or were folklore in the community but were never published. In Sec. 3, we use the Exponential Time Hypothesis (ETH) to exhibit a strict natural hierarchy inside classes of problems having incremental polynomial time algorithms which implies a separation between polynomial delay and incremental polynomial time, the last classes not yet separated. This separation is the first in enumeration complexity to rely on ETH and we believe it can lead to new conditional lower bounds on real enumeration problems. In Sec. 4, we prove that algorithms in *linear incremental time* and polynomial space can often be turned into *polynomial delay* and polynomial space algorithms, paving the way for a proof that the two classes are equal. Finally, in Sec. 5, we consider enumeration problems whose solutions can be given by a polynomial time uniform random generator. We prove that this class of problems is included in the classes studied in this article if we allow randomization: a problem which admits a polynomial time uniform random generator can be solved by an exponential space randomized polynomial delay algorithm or a polynomial space randomized polynomial incremental time algorithm.

2 Complexity Classes

Let Σ be a finite alphabet and Σ^* be the set of finite words built on Σ. We assume that our alphabet is $\{0, 1, \sharp\}$. We denote by $|x|$ the size of a word $x \in \Sigma^*$ and by $|S|$ the cardinal of a set S. We recall here the definition of an enumeration problem:

▶ **Definition 1** (Enumeration Problem). Let $A \subseteq \Sigma^* \times \Sigma^*$ be a binary predicate, we write $A(x)$ for the set of y such that $A(x, y)$ holds. The enumeration problem Π_A is the function which associates $A(x)$ to x.

From now on, we only consider predicates A such that $A(x)$ is finite for all x. The computational model is the random access machine model (RAM) with addition, subtraction and multiplication as its basic arithmetic operations. We have additional output registers, and when a special OUTPUT instruction is executed, the content of the output registers is produced. A RAM machine solves Π_A if, for each $x \in \Sigma^*$, on input x it produces a sequence $y_1, \dots, y_n$ such that $A(x) = \{y_1, \dots, y_n\}$ and for all $i \neq j$, $y_i \neq y_j$.

To simplify the definition of the complexity classes, we ask the RAM machine to stop immediately after the last OUTPUT instruction is executed. The cost of every instruction is assumed to be in $O(1)$ except the arithmetic instructions which are of cost linear in the size of their inputs. The space used by the machine at a given step is the sum of the number of bits required to store the integers in its registers.

We denote by $T(M, x, i)$ the sum of the costs of the instructions executed before the i^{th} OUTPUT instruction. Usually the machine M will be clear from the context and we will write $T(x, i)$ instead of $T(M, x, i)$.

The class EnumP. We can naturally define complexity classes of enumeration problems by restricting the predicate $A(x, y)$ used to define enumeration problems.

▶ **Definition 2.** Let $\mathcal{C}$ be a set of binary predicates, Enum $\cdot\, \mathcal{C}$ is the set of problems Π_A such that $A \in \mathcal{C}$.

We are especially interested in the class of problems which are the enumeration of the solutions of an NP problem. Let PTPB (polynomial time polynomially balanced) be the set of predicates A such that $A(x, y)$ is decidable in polynomial time and such that the elements of $A(x)$ are of size polynomial in $|x|$. We will denote the class Enum $\cdot$ PTPB by EnumP for resemblance with NP. We will also consider Enum $\cdot$ F the class of all enumeration problems, where F is the set of all A such that, for all x, $A(x)$ is finite.

The class EnumP has complete problems for the *parsimonious reduction*. This reduction is taken from counting complexity and realizes polynomial time bijections between sets of solutions. The problem Π_{SAT}, the task of listing all solutions of a 3-CNF formula is EnumP-complete, since the reduction used in the proof that SAT is NP-complete [6] is parsimonious. Some problems seemingly as hard as Π_{SAT} are not EnumP-complete because the parsimonious reduction is too restrictive. Therefore many other reductions have been considered [18] but none has allowed to prove a completeness result for a class other than EnumP.

The class OutputP. To measure the complexity of an enumeration problem, we consider the total time taken to compute all solutions. Since the number of solutions can be exponential with regard to the *input*, it is more relevant to give the total time in function of the size of the input and the *the output* combined. In particular, we like it to be polynomial in the number of solutions; algorithms with this complexity are said to be in output polynomial time or sometimes in polynomial total time. We define two corresponding classes, one when the problem is in EnumP and one when it is not restricted.

▶ **Definition 3** (Output polynomial time)**.** A problem $\Pi_A \in$ EnumP (respectively, in Enum $\cdot$ F) is in OutputP (resp., OutputP^F) if there is a polynomial $p(x, y)$ and a machine M which solves Π_A and such that for all x, $T(x, |A(x)|) < p(|x|, |A(x)|)$.

For instance, if we see a polynomial as a set of monomials, then classical algorithms for interpolating multivariate polynomials from their values are output polynomial [25] as they produce the polynomial in a time proportional to the number of its monomials.

▶ **Proposition 4.** OutputP ≠ EnumP if and only if P ≠ NP.

Proof. Assume OutputP = EnumP, thus Π_{SAT} is in OutputP. Then on an instance x, it can be solved in time bounded by $p(|x|)q(|SAT(x)|)$ where p and q are two polynomials. Therefore if we run the enumeration algorithm for Π_{SAT} and it does not stop before a time $p(|x|)$, we know there must be a least an element in $SAT(x)$. If it stops before a time $p(|x|)$, it produces the set $SAT(x)$ therefore we can decide the problem SAT in polynomial time.

Assume now that P = NP. The problem SAT is autoreducible, that is given a formula ϕ and a partial assignment of its variables a, we can decide whether a can be extended to a satisfying assignment by deciding SAT on another instance. Therefore we can decide in polynomial time if there is an extension to a partial assignment and by using the classical method of the backtrack search we obtain an OutputP algorithm for Π_{SAT}. ◀

The classes EnumP and OutputP may be seen as analog of NP and P for the enumeration. Usually an enumeration problem is considered to be tractable if it is in OutputP, especially if its complexity is linear in the number of solutions. The problems in OutputP are easy to solve when there are few solutions and hard otherwise. We now introduce classes of complexity inside OutputP to capture the problems which could be considered as classes of tractable problems even when the number of solutions is high.

The class IncP. From now on, a polynomial time precomputation step is always allowed before the start of the enumeration. It makes the classes of complexity more meaningful, especially their fine grained version. It is usually used in practice to set up useful datastructures or to preprocess the instance.

Given an enumeration problem A, we say that a machine M enumerates A in *incremental time* $f(m)g(n)$ if on every input x, M enumerates m elements of $A(x)$ in time $f(m)g(|x|)$ for every $m \leq |A(x)|$ and runs in time $f(|A(x)|)g(|x|)$.

▶ **Definition 5** (Incremental polynomial time)**.** A problem $\Pi_A \in$ EnumP (respectively, in Enum · F) is in IncP_a (resp., IncP_a^F) if there is a machine M which solves it in incremental time $C \cdot m^a n^b$ for C and b constants. Moreover, we define $\mathrm{IncP} = \bigcup_{a\geq 1} \mathrm{IncP}_a$ and $\mathrm{IncP}^F = \bigcup_{a\geq 1} \mathrm{IncP}_a^F$.

The class IncP can be defined through a search problem associated to a predicate A: given x and a set of solutions $\mathcal{S} \subseteq A(x)$ find $y \in A(x) \setminus \mathcal{S}$ or answer that $\mathcal{S} = A(x)$. The problems in IncP are the ones with a polynomial search problem.

The class IncP can also be defined as the class of problems solvable by an algorithm with a delay polynomial in the number of already generated solutions and in the size of the input. This alternative definition is motivated by saturation algorithms, which generates solutions by applying some polynomial time rules to enrich the set of solutions until saturation. There are many saturation algorithms, for instance to enumerate circuits of matroids [17] or to compute saturation by sets closure operations [19].

With our definition, we capture the fact that investing more time guarantees more solutions to be output, which is a bit more general at first sight than the delay because we do not impose regularity between two solutions. We will see in Sec. 4 that both definitions are actually equivalent but the price for regularity is to use exponential space.

▶ **Proposition 6.** If P ≠ NP ∩ coNP then IncP ⊊ OutputP.

Proof. Let $L \in \mathrm{coNP} \cap \mathrm{NP}$: there is a predicate A such that $L = \{x \mid A(x) \neq \emptyset\}$ and a predicate B such that $\bar{L} = \{x \mid B(x) \neq \emptyset\}$. Let q be a polynomial such that $y \in A(x)$ or $y \in B(x)$ implies that $|y| \leq q(|x|)$. For simplicity, we assume L to be over $\{0,1\}^*$.

Let $C(x, y\sharp w)$ be the predicate which is true if and only if $A(x, y) \vee B(x, y)$ and $|w| \leq q(|x|)$, that is C is the union of A and B plus some padding. Observe that the set $C(x)$ is never empty, since if $x \in L$ there exists y such that $A(x, y)$ holds and if not there exists y such that $B(x, y)$ holds.

Thanks to the padding, there are more than $2^{|w|} = 2^{q(|x|)}$ elements in $C(x)$ for each x satisfying either A or B. Therefore the bruteforce enumeration algorithm is polynomial in the number of solutions, which proves that Π_C is in OutputP.

If IncP = OutputP, we have an incremental algorithm for Π_C. In particular, it gives, on any instance x, the first solution $y\sharp w$ in polynomial time and we can decide whether y satisfies $A(x, y)$ or $B(x, y)$ in polynomial time. This procedure decides whether $x \in L$ in polynomial time therefore P = coNP ∩ NP. ◀

▶ **Open problem 1.** Can we prove an equivalence by proving either IncP $\subsetneq$ OutputP $\Rightarrow$ P $\neq$ NP ∩ coNP or P $\neq$ NP $\Rightarrow$ IncP $\subsetneq$ OutputP.

Observe that without the requirement to be in EnumP, IncP and OutputP are separated unconditionally.

▶ **Proposition 7.** $\text{IncP}^F \neq \text{OutputP}^F$.

Proof. Choose any EXP-complete decision problem L and let A be the predicate such that $A(x, y)$ holds if and only if $x = 0\sharp i$ if $x \in L$ or $1\sharp i$ if $x \notin L$ with $0 \leq i < 2^{|x|}$. Therefore Π_A is easy to solve in linear total time, but since EXP $\neq$ P we cannot produce the first solution in polynomial time and thus Π_A is not in incremental polynomial time. ◀

The class DelayP. We now define the polynomial delay which by definition is a subclass of IncP_1. In Sec.3 we prove its separation from IncP, while in Sec. 4 we study its relationship with IncP_1.

▶ **Definition 8** (Polynomial delay)**.** A problem $\Pi_A \in$ EnumP (respectively in Enum · F) is in DelayP (resp. in DelayP^F) if there is a machine M which solves it such that for all x and for all $0 < t \leq |A(x)|$, $|T(x, t) - T(x, t-1)| < C|x|^a$ for C and a constants.

3 Strict hierarchy in incremental time problems

We prove strict hierarchies for IncP_a^F unconditionally and for IncP_a modulo the *Exponential Time Hypothesis* (ETH). Since DelayP $\subseteq \text{IncP}_1$ it implies that DelayP $\neq$ IncP modulo ETH.

▶ **Proposition 9.** $\text{IncP}_a^F \subsetneq \text{IncP}_b^F$ when $1 \leq a < b$.

Proof. By the time hierarchy theorem [13], there exists a language L which can be decided in time $O(2^{nb})$ but not in time $O(2^{na})$. Let $n = |x|$. We build a predicate $A(x, y)$ which is true if and only if either y is a positive integer written in binary with $y < 2^n$ or $y = \sharp 0$ when $x \notin L$ or $y = \sharp 1$ when $x \in L$. We have an algorithm to solve Π_A: first enumerate the 2^n trivial solutions then run the $O(2^{nb})$ algorithm which solves A to compute the last solution. This algorithm is in IncP_b^F, since finding the 2^n first solutions can be done in IncP_1 and the last one can be found in time $O((2^n)^b)$. On the other hand, if there is an IncP_a^F algorithm, one finds the solution $\sharp 0$ or $\sharp 1$ in time $O(2^{na})$ which solves A in time $O(2^{na})$ therefore $\Pi_A \notin \text{IncP}_a^F$. ◀

To prove the existence of a strict hierarchy in IncP, we need to assume some complexity hypothesis since P = NP implies IncP = IncP_1 by the same argument as in the proof of

Prop. 4. Moreover, the hypothesis must be strong enough to replace the time hierarchy argument.

The Exponential Time Hypothesis states that there exists $\epsilon > 0$ such that there is no algorithm for 3-SAT in time $\tilde{O}(2^{\epsilon n})$ where n is the number of variables of the formula and $\tilde{O}$ means that we have a factor of $n^{O(1)}$. The *Strong Exponential Time Hypothesis* (SETH) states that for every $\epsilon < 1$, there is no algorithm solving SAT in time $\tilde{O}(2^{\epsilon n})$.

We show that if ETH holds, then $\mathrm{IncP}_a \subsetneq \mathrm{IncP}_b$ for all $a < b$. For $t \leq 1$, let R_t be the following predicate: given a CNF ϕ with n variables, $R_t(\phi)$ contains:

- the integers from 1 to $2^{nt} - 1$
- the satisfying assignments of ϕ duplicated 2^n times each, that is $SAT(\phi) \times [2^n]$.

We let Pad_t be the enumeration problem associated to R_t, that is $\mathsf{Pad}_t = \Pi_{R_t}$. The intuition behind Pad_t is the following. Imagine that $t = b^{-1}$ for some $b \in \mathbb{N}$. By adding sufficiently many dummy solutions to the satisfying assignments of a CNF-formula ϕ, we can first enumerate them quickly and then have sufficient time to bruteforce $SAT(\phi)$ in IncP_b before outputting the next solution. This shows that $\mathsf{Pad}_{b^{-1}} \in \mathrm{IncP}_b$. Now, if there exists $a < b$ such that $\mathrm{IncP}_a = \mathrm{IncP}_b$, we would have a way to find a solution of ϕ in time $\tilde{O}(2^{\frac{a}{b}n})$ which already violate SETH. To show that we also violates ETH we repeat this trick but we do not bruteforce SAT(ϕ) anymore. We can do better by using this $\tilde{O}(2^{\frac{a}{b}n})$ algorithm for SAT and we can gain a bit more on the constant in the exponent. We show that by repeating this trick, we can make the constant as small as we want. We formalize this idea:

▶ **Lemma 10.** *Let $d < 1$. If we have an $\tilde{O}(2^{dn})$ algorithm for* SAT*, then for all $b \in \mathbb{N}$, $\mathsf{Pad}_{\frac{d}{b}}$ is in IncP_b.*

Proof. We enumerate the integers from 1 to $2^{\frac{dn}{b}} - 1$ and then call the algorithm to find a satisfying assignment of ϕ. We have enough time to run this algorithm since the time allowed before the next anser is $\tilde{O}\left(\left(2^{\frac{dn}{b}}\right)^b\right) = \tilde{O}(2^{dn})$. If the formula is not satisfiable, then we stop the enumeration. Otherwise, we enumerate all copies of the discovered solution. We have then enough time to bruteforce the other solutions. ◀

▶ **Lemma 11.** *If Pad_t is in IncP_a, then there exists an $\tilde{O}(2^{nta})$ algorithm for* SAT*.*

Proof. Since Pad_t is in IncP_a, we have an algorithm for Pad_t that outputs m elements of $R_t(\phi)$ in time $O(m^a|\phi|^c)$ for a constant c. We can then output 2^{nt} elements of $R_t(\phi)$ in time $O(2^{nta}|\phi|^c) = \tilde{O}(2^{nta})$. If the enumeration stops before having output 2^{nt} solutions, then the formula is not satisfiable. Otherwise, we have necessarily enumerated at least one satisfying assignment of ϕ which gives the algorithm. ◀

▶ **Lemma 12.** *If $\mathrm{IncP}_a = \mathrm{IncP}_b$, then for all $i \in \mathbb{N}$, $\mathsf{Pad}_{\frac{a^i}{b^{i+1}}}$ is in IncP_a.*

Proof. The proof is by induction on i. For $i = 0$, by Lemma 10, $\mathsf{Pad}_{\frac{1}{b}}$ is in IncP_b since we have an $\tilde{O}(2^n)$ bruteforce algorithm for SAT. Thus, if $\mathrm{IncP}_b = \mathrm{IncP}_a$, $\mathsf{Pad}_{\frac{1}{b}}$ is in IncP_a too.

Now assume that $\mathsf{Pad}_{\frac{a^i}{b^{i+1}}}$ is IncP_a. By Lemma 11, we have an $\tilde{O}(2^{dn})$ algorithm for $d = \frac{a^{i+1}}{b^{i+1}}$. Thus, by Lemma 10, $\mathsf{Pad}_{\frac{d}{b}} = \mathsf{Pad}_{\frac{a^{i+1}}{b^{i+2}}}$ is in $\mathrm{IncP}_b = \mathrm{IncP}_a$. ◀

▶ **Theorem 13.** *If* ETH *holds, then $\mathrm{IncP}_a \subsetneq \mathrm{IncP}_b$ for all $a < b$.*

Proof. If there exists $a < b$ such that $\mathrm{IncP}_a = \mathrm{IncP}_b$, then by Lemma 12, for all i, $\mathsf{Pad}_{\frac{a^i}{b^{i+1}}}$ is in IncP_a. Thus by Lemma 11, we have an $\tilde{O}(2^{d_i n})$ algorithm for SAT and then for 3-SAT in particular, where $d_i = \left(\frac{a}{b}\right)^i$. Since $\lim_{i\to\infty} d_i = 0$, this contradicts ETH. ◀

In the previous proofs, we did not really used SAT. We needed an NP problem, with a set of easy to enumerate potential solutions of size 2^n that cannot be solved in time $2^{o(n)}$. For instance we could use CIRCUIT-SAT which is the problem of finding a satisfying assignment to a Boolean circuit. We can thus prove our result by assuming a weaker version of ETH as it is done in [1]. It would be nice to further weaken the hypothesis, but it seems hard to rely only on a classical complexity hypothesis such as P $\neq$ NP. The other way we could improve this result, is to prove a lower bound for a natural enumeration problem instead of Pad_t.

▶ **Open problem 2.** Prove that enumerating the minimal transversals of an hypergraph cannot be done in IncP_1 if ETH hods.

4 Space and regularity of enumeration algorithms

The main difference between IncP_1 and DelayP is the regularity of the delay between two solutions. In several algorithms [15] an exponential queue is used to store results, which are then output regularly to guarantee a polynomial delay. This is in fact a general method which can be used to prove that $\mathrm{IncP}_1 = \mathrm{DelayP}$.

▶ **Proposition 14.** $\mathrm{IncP}_1 = \mathrm{DelayP}$.

Proof. Let $\Pi_A \in \mathrm{IncP}_1$, then there is an algorithm I which on an instance of size n, produces k solutions in time bounded by $kp(n)$ where p is a polynomial.

We construct an algorithm I' which solves Π_A with delay $O(p(n)+s)$ where s is a bound on the size of a solution. The algorithm I' simulates I and increments a counter c at each step of I. Each time I outputs a solution, we append it to a queue ℓ instead. Each time c reaches $p(n)$, it is reset to zero and the first solution of ℓ is output and removed.

Since I is guaranteed to produce k solutions in time bounded by $kp(n)$, there will always be at least one solution in ℓ when c reaches $p(n)$, otherwise it is the end of the execution of I. Moreover, the time to maintain the counter plus the time to write a solution in ℓ is bounded by $O(p(n)+s)$ which is polynomial in n since by definition of IncP_1, $\Pi_A \in \mathrm{EnumP}$ that is A is polynomially balanced and s is bounded by a polynomial in n. ◀

An inconvenience of Proposition 14 is that the method used to go from incremental polynomial time to polynomial delay may blow up the memory. In practice, the polynomial delay is relevant if we also use only polynomial space. This naturally raises the question of understanding the relationship between IncP_1 and DelayP when the space is polynomial. Concretely, we are interested in the following question: does every problem in IncP_1 with polynomial space is also in DelayP with polynomial space? Unfortunately, no classical assumptions in complexity theory seem to help for separating these classes nor were we able to prove the equality of both classes. The rest of this section is dedicated to particular IncP_1 algorithms where the enumeration is sufficiently regular to be transformed into DelayP algorithm without blowing up the memory.

An algorithm I is *incremental linear* if there exists a polynomial h such that on any instance of size n, it produces k solutions in time bounded by $kh(n)$. We call h the *average delay* of I. By definition, a problem $\Pi_A \in \mathrm{EnumP}$ is in IncP_1 if and only if there exists an incremental linear algorithm solving Π_A.

Let I be an incremental linear algorithm. Recall that $T(I, x, i)$ is number of steps made by I before outputting the i^{th} solution. To make notations lighter, we will write $T(i)$ since x and I will be clear from the context. Consider a run of I on the instance x, we will call m_i an encoding of i, the memory of I and its state at the time it outputs the i^{th} solution. We say that the index i is a *p-gap* of I if $T(i+1) - T(i) > p(|x|)$. If I has no p gaps for

some polynomial p, it has polynomial delay p. We now show that when the number of large gaps is small, we can turn an incremental linear algorithm into a polynomial delay one, by computing shortcuts in advance.

▶ **Proposition 15.** Let $\Pi_A \in \mathrm{IncP}_1$ and I be an incremental linear algorithm for Π_A using polynomial space. Assume there are two polynomials p and q such that for all instances x of size n, there are at most $q(n)$ p-gaps in the run of I, then $\Pi_A \in \mathrm{DelayP}$.

Proof. Since I is incremental linear it has a polynomial average delay that we denote by h. We run in parallel two copies of the algorithm I that we call I_1 and I_2. When I_1 simulates one computation step of I, I_2 simulates $2h(n)$ computation steps of I. Moreover I_2 counts the number of consecutive steps without finding a new solution so that it detects p-gaps. When it detects such a gap, a pair (i, m_{i+1}) is stored where i is the index of the last solution before the gap and m_{i+1} is the description of the machine when it outputs the $(i+1)^{\text{th}}$ solution. Since there are at most $q(n)$ p-gaps and because I uses polynomial space, the memory used by I_2 is polynomial. When I_1 outputs a solution of index i and that (i, m_{i+1}) was stored by I_2, its state and memory is changed to m_{i+1}. Assume I_2 finds a gap at index i, then because I is incremental linear, we have $T(i+1) - T(i) < (i+1)h(n)$. Therefore I_1 at the same time has done at most $\frac{i+1}{2}$ computation steps and thus has not yet seen the i^{th} solution, which proves that the algorithm works as described. In that way, I_1 will always generate solutions with delay less than $p(n)h(n)$ because I_1 has no p-gaps by construction, and each of its computation steps involves $h(n)$ computation steps of I_2. ◀

We can prove something more general, by requiring the existence of a large interval of solutions without p-gaps rather than bounding the number of gaps. It captures more cases, for instance an algorithm which outputs an exponential number of solutions at the beginning without gaps and then has a superpolynomial number of gaps. The idea is to compensate for the gaps by using the dense parts of the enumeration.

▶ **Proposition 16.** Let $\Pi_A \in \mathrm{IncP}_1$ and I be an incremental linear algorithm for Π_A using polynomial space. Assume there are two polynomials p and q such that for all x of size n, and for all $k \leq |A(x)|$ there exists $a < b \leq k$ such that $b - a > \frac{k}{q(n)}$ and there are no p-gaps between the a^{th} and the b^{th} solution. Then $\Pi_A \in \mathrm{DelayP}$.

Proof. We let h be the average delay of I. We fix x of length n and describe a process that enumerates A with delay at most $2q(n)h(n) \cdot (q(n)h(n) + p(n))$ and polynomial space on input x. Our algorithm runs two processes in parallel: En, the enumerator and Ex, the explorer. Both processes simulate I on input x but at a different speed that we will fix later in the proof. En is the only one outputting solutions. We call a solution *fresh* if it has not yet been enumerated by En.

Ex simulates I and discovers the boundaries of the largest interval without p-gaps containing only fresh solutions that we call the *stock*. More precisely, it stores two machine states: m_a and m_b where a and b correspond to indices of fresh solutions such that there are no p-gaps between a and b and it is the largest such interval. Intuitively, the stock contains the fresh solutions that will make up for p-gaps in the enumeration of I.

En can work in two different modes. If En is in simple mode, then it only simulates I on input x and outputs a solution whenever I outputs one and counts the number of steps between two solutions. When it detects a p-gap, En switches to filling mode. In filling mode, En starts by copying m_a into a new variable s and m_b into a new variable t. It then runs two simulations of I: the first one is the continuation of the simulation that was done in simple mode. The second one, which we call the *filling simulation* is a simulation of I starting in

state m_a. En simulates $h(n)q(n)$ steps of the first enumeration and then as many steps as necessary to find the next solution of the filling simulation. Since the stock does not contain p-gaps by definition, we know that En outputs solutions with delay at most $h(n)q(n)+p(n)$. To avoid enumerating the same solution twice, whenever the first simulation reaches the state stored in s, we stop the first simulation and En switches again in simple mode using the filling simulation as starting point.

We claim that the first simulation will always reach state s before the filling simulation reach the end of the stock. Indeed, assume that the filling simulation has reached the end of the stock and outputs the b^{th} solution. By definition, the stock is the largest interval without p-gaps before this solution and it is of size at least $b/q(n)$ by assumption. Thus, the first simulation has simulated at least $(b/q(n))h(n)q(n) = b \cdot h(n)$ steps of I in parallel. Thus, by definition of h, the b first solutions have been found by the first simulation. It must have reached state s before the filling simulation reaches the end of the interval.

Using this strategy, it is readily verified that if En has always a sufficiently large stock at hand, it enumerates $A(x)$ entirely with delay at most $h(n)q(n) + p(n)$.

We now choose the speed of Ex in order to guarantee that the stock is always sufficiently large: each time En simulates one step of I, Ex simulates $2h(n)q(n)$ steps of I.

There is only one situation that could go wrong: the enumerator can reach state m_b, which is followed by a p-gap while the explorer has not found a new stock yet. We claim that having chosen the speed as we did, we are guaranteed that it never happens. Indeed, if En reaches m_b, then it has already output b solutions. Thus, Ex has already simulated at least $b \cdot 2h(n)q(n)$ steps of I. By definition of h, Ex has already found $2b \cdot q(n)$ solutions and then, it has found an interval without p-gaps of size $(2b \cdot q(n))/q(n) = 2b$ which is necessarily ahead of the simulation of En.

The property of the last paragraph is only true if the simulation of I by Ex has not stopped before $b \cdot 2h(n)q(n)$ steps. To deal with this case, as soon as Ex has stopped, En enters in filling mode if it was not in this mode and does a third simulation in parallel of I beginning at state m_b. This takes care of the solutions after the last stock. ◀

Observe that in those two proofs, we do not use all properties of an algorithm in IncP_1 but only that the predicate is polynomially balanced. All known algorithms which are both incremental and in polynomial space are in fact polynomial delay algorithms with a bounded number of repetitions and a polynomial time algorithm to decide whether it is the first time a solution is produced [23]. It seems that if we can turn such an algorithm to one in polynomial delay, we would have solved the general problem.

▶ **Open problem 3.** Prove or disprove that IncP_1 with polynomial space is equal to DelayP with polynomial space.

5 From Uniform Generator to efficient randomized enumeration

In this section, we explore the relationship between efficient enumeration and random generation of combinatorial structures or sampling. The link between sampling and counting combinatorial structures has already been studied. For instance, Markov Chain Monte Carlo algorithms can be used to compute an approximate number of objects [14] or in the other direction, generating functions encoding the number of objects of each size can be used to obtain Boltzman samplers [8]. We extend this link to enumeration by proving that the class of problems whose solutions can be uniformly generated in polynomial time is included in randomized DelayP.

▶ **Definition 17.** Let $\Pi_A \in \text{EnumP}$. A *polytime uniform generator* for A is a randomized RAM machine M which outputs an element y of $A(x)$ in time polynomial in $|x|$ such that the probability over every possible run of M on input x that M outputs y is $|A(x)|^{-1}$.

We now define a randomized version of IncP, which has first been introduced in [23, 24].

▶ **Definition 18.** A problem Π_A is in randomized IncP_k if $\Pi_A \in \text{EnumP}$ and there exists constants $a, b \in \mathbb{N}$ and a randomized RAM machine M such that for every $x \in \{0,1\}^*$ and $\epsilon \in \mathbb{Q}_+$, the probability that M, on input x and ϵ, enumerates $A(x)$ in incremental time $m^k n^a \epsilon^{-b}$ is greater than $1 - \epsilon$.

Definition 18 can be understood as follows, on input $x \in \{0,1\}^*$ and $\epsilon \in \mathbb{Q}_+$, the probability of the following fact is at least $1 - \epsilon$: for every $t \leq |A(x)|$, M has enumerated t distinct elements of $A(x)$ after $t^k n^a \epsilon^{-b}$ steps and stops in time less than $|A(x)|^k n^a \epsilon^{-b}$.

▶ **Theorem 19.** *If $\Pi_A \in \text{EnumP}$ has a polytime uniform generator, then Π_A is in randomized IncP_1.*

Algorithm 1: An algorithm to enumerate Π_A in randomized IncP_1, where every element of $A(x)$ is of size at most $p(|x|)$.

Input: $x \in \{0,1\}, \epsilon \in \mathbb{Q}_+$
begin
  $E \leftarrow \emptyset$; $r \leftarrow 0$;
  $K \leftarrow 2 \cdot \big(p(|x|) - \log(\epsilon/2)\big)$;
  while $r \leq K \cdot |E|$ **do**
    Draw $e \in A(x)$ uniformly and $r \leftarrow r + 1$;
    if $e \notin E$ **then**
      Output e and $E \leftarrow E \cup \{e\}$;

Proof. Algorithm 1 shows how to use a generator for A to enumerate its solutions in randomized IncP_1. The idea is the most simple: we keep drawing elements of $A(x)$ uniformly by using the generator. If the drawn element has not already been enumerated, then we output it and remember it in a set E. We keep track of the total number of draws in the variable r. If this variable reaches a value that is much higher than the number of distinct elements found at this point, we stop the algorithm. We claim that Algorithm 1 is in randomized IncP_1, the analysis is similar to the classical coupon collector theorem [10].

We let p be a polynomial such that for every $x \in \{0,1\}^*$, the size of elements of $A(x)$ is a most $p(|x|)$. Such a polynomial exists since $\Pi_A \in \text{EnumP}$. Observe that all operations can be done in polynomial time in $|x|$ since we can encode E – the set of elements that have already been enumerated – by using a datastructure such as a balanced binary search tree for which adding and searching for an element may be done in time $O(\log |E|)$, which is polynomial in x since $\log |E| \leq \log |A(x)| \leq p(|x|)$.

Moreover, observe that if the algorithm is still running after tK executions of the while loop, then we have $|E| \geq t$, thus we have enumerated more than t elements of $A(x)$. Since each loop takes a time polynomial in $|x|$ and K is polynomial in $|x|$ and ϵ, we have that if the algorithm still runs after a time $t \cdot \text{poly}(|x|)$, then the run is similar to a run in IncP_1.

Hence, to show that Π_A is in randomized IncP_1, it only remains to prove that the probability that Algorithm 1 stops before having enumerated $A(x)$ completely is smaller

than ϵ. The main difficulty is to decide when to stop. It cannot be done deterministically since we do not know $|A(x)|$ *a priori*. Algorithm 1 stops when the total number of draws r is larger than $K \cdot |E|$, where E is the set of already enumerated elements of $A(x)$. In the rest of the proof, we prove that with $K = 2 \cdot \big(p(|x|) - \log(\epsilon/2)\big)$, Algorithm 1 stops after having enumerated $A(x)$ completely with probability greater than $1 - \epsilon$.

In the following, we fix $x \in \{0,1\}^*$ and $\epsilon \in \mathbb{Q}_+$. We denote by $s = |A(x)|$ the size of $A(x)$. Remember that we have $s \leq 2^{p(|x|)}$. We denote by T the random variable whose value is the number of distinct elements of $A(x)$ that have been enumerated when the algorithm stops. Our goal is to show that $\mathbb{P}(T < s) \leq \epsilon$.

We start by showing that $\mathbb{P}(T \leq s/2) \leq \epsilon/2$. Let $t \leq s/2$. We bound the probability that $T = t$. If the algorithm stops after having found t solutions, we know that it has found t solutions in less than $1 + (t-1)K$ draws, otherwise, if the algorithm has found less than t solutions after $1+(t-1)K$ draws then the while would finish. After that, it keeps on drawing already enumerated solutions until it has done $1 + tK$ draws and stops. Thus, it does at least K draws without finding new solutions. Since $t \leq s/2$, the probability of drawing a solution that was already found is at most $1/2$. Thus for all $t \leq s/2$,

$$\mathbb{P}(T = t) \leq 2^{-K} \leq 2^{-p(|x|)}(\epsilon/2)$$

since $K \geq -\log(\epsilon/2)$. Now, applying the union bound yields:

$$\mathbb{P}(T \leq s/2) \leq \sum_{t=1}^{s/2} \mathbb{P}(T = t) \leq (s/2)2^{-p(|x|)}(\epsilon/2) \leq \epsilon/2$$

since $s \leq 2^{p(|x|)}$.

Now, we show that $\mathbb{P}(s/2 < T < s) \leq \epsilon/2$. Assume that $T > s/2$. Then, after $K \cdot (s/2)$ draws, the algorithm has not stopped. Thus the probability that $s/2 < T < s$ is smaller than the probability that, after $r = K \cdot (s/2)$ draws, we have not found every element of $A(x)$. Given an element $y \in A(x)$, the probability that y is not drawn after r draws is $(1 - 1/s)^r$. Thus, the probability that after $r = K \cdot (s/2)$ draws, we have not found every element of $A(x)$ is at most

$$\begin{aligned}
s \cdot (1 - 1/s)^r &\leq s \cdot 2^{-r/s} \\
&\leq s \cdot 2^{\log(\epsilon/2) - p(|x|)} && \text{since } r = K \cdot (s/2) \\
&\leq \epsilon/2 \cdot s 2^{-p(|x|)} \\
&\leq \epsilon/2 && \text{since } s \leq 2^{p(|x|)}
\end{aligned}$$

In the end, $\mathbb{P}(T < s) \leq \mathbb{P}(T \leq s/2) + \mathbb{P}(s/2 < T < s/2) \leq \epsilon$. We observe that the running time of Algorithm 1 is actually polynomial in $\log(\epsilon^{-1})$ which is a much better bound than the one of Definition 18 since $\log(\epsilon^{-1})$ is polynomial in the size of the encoding of ϵ for $\epsilon < 1$. ◀

Applying the same technique as Prop. 14, we can turn Algorithm 1 into a randomized DelayP algorithm by delaying the solutions:

▶ **Corollary 20.** *If* $\Pi_A \in$ EnumP *has a polytime uniform generator, then* Π_A *is in randomized* DelayP.

We observe that this algorithm works also when the distribution of the generator is not uniform. However, each solution must appear frequently enough, that is a polynomial fraction of the uniform.

Polynomial space algorithm. The main default of Algorithm 1 is that it stores all solutions enumerated and therefore needs a space which may be exponential. We will now see a way to reduce the space used by this algorithm to a polynomial amount if we allow repetitions in the output. Indeed, if we only have polynomial space, we cannot avoid repetitions without making further assumptions about the generator: there may be an exponential number of solutions, at any point of the algorithm we need to encode the subset of already generated solutions and these subsets are in doubly exponential number and thus cannot be encoded in polynomial space.

We consider enumeration algorithms that can output the same solution an unbounded number of times. The main difficulty is again to decide when to stop so that no solution is forgotten with high probability. The method used in Algorithm 1 does not work in this case since we cannot maintain the number of distinct solutions that have been output so far. However, there exists data structures which allow to approximate the cardinal of a dynamic set using only a logarithmic number of bits in the size of the set [11, 16]. Algorithm 2 shows how we can exploit such datastructures to design a randomized incremental algorithm from a uniform generator. Unlike Algorithm 1, Algorithm 2 cannot be turned into a polynomial delay algorithm since it would require exponential space and our improvement would be then useless.

Algorithm 2: An algorithm in randomized IncP_1 with polynomial space such that every element of $A(x)$ is of size at most $p(|x|)$.

Input: $x \in \{0,1\}, \epsilon \in \mathbb{Q}_+$
begin
  Initialize E; $r \leftarrow 0$;
  $K \leftarrow 4 \cdot \big(p(|x|) - \log(\epsilon/4)\big)$;
  while $r \leq K \cdot \mathsf{estimate}(E)$ **do**
    Draw $e \in A(x)$ uniformly and $r \leftarrow r + 1$;
    Output e and $\mathsf{update}(E, e)$;

▶ **Proposition 21.** If $\Pi_A \in \mathrm{EnumP}$ has a polytime uniform generator, then there is an enumeration algorithm in randomized IncP_1 *with repetitions* and *polynomial space*.

Proof. The procedure update in Algorithm 2 maintains a data structure which allows estimate to output an approximation of $|E|$. If we use the results of [16], we can get a 2-approximation of $|E|$ during all the algorithm with probability $1 - \frac{\epsilon}{2}$. The data structure uses a space $\log(|E|)\log(\frac{1}{\epsilon})$. The process $\mathsf{update}(E, e)$ does $O(\log(\frac{1}{\epsilon}))$ arithmetic operations and estimate does $O(1)$ arithmetic operations. The arithmetic operations are over solutions seen as integers which are of size polynomial in n. The analysis of the delay is the same as before, but to the cost of generating a solution, we add the cost of computing $\mathsf{update}(E, e)$ and estimate which are also polynomial in n.

The analysis of the correctness of the algorithm is the same, except that now v is a 2-approximation of $|E|$ with probability $1 - \frac{\epsilon}{2}$. We have adapted the value of K such that with probability $\frac{\epsilon}{2}$ the algorithm will not stop before generating all solutions even if $|E|$ is approximated by $\frac{|E|}{2}$. Therefore the probability to wrongly evaluate $|E|$ plus the probability that the algorithm stops too early is less than ϵ.

◀

The method we just described here can be relevant, when we have a polynomial delay algorithm using the supergraph method: a connected graph whose vertices are all the solutions is defined in such a way that the edges incident to a vertex can be enumerated in polynomial time. The algorithm does a traversal of this graph which requires to store all found solutions. By doing a random walk and using Algorithm 2 we get a slightly slower algorithm but we do not need exponential memory. One could also traverse the graph using only a logarithmic space in the numbers of solutions using a universal sequence [21] but we would have no guarantee on the delay and the polynomial slowdown would be huge in practice.

References

1 Amir Abboud, Thomas Dueholm Hansen, Virginia Vassilevska Williams, and Ryan Williams. Simulating branching programs with edit distance and friends: or: a polylog shaved is a lower bound made. In *Proceedings of the 48th Annual ACM SIGACT Symposium on Theory of Computing, STOC 2016, Cambridge, MA, USA, June 18-21, 2016*, pages 375–388, 2016.

2 D. Avis and K. Fukuda. Reverse search for enumeration. *Discrete Applied Mathematics*, 65(1):21–46, 1996.

3 Guillaume Bagan. *Algorithms and complexity of enumeration problems for the evaluation of logical queries*. PhD thesis, University of Caen Normandy, France, 2009.

4 Dominique Barth, Olivier David, Franck Quessette, Vincent Reinhard, Yann Strozecki, and Sandrine Vial. Efficient generation of stable planar cages for chemistry. In *International Symposium on Experimental Algorithms*, pages 235–246. Springer, 2015.

5 Johann Brault-Baron. *De la pertinence de l'énumération: complexité en logiques propositionnelle et du premier ordre*. PhD thesis, Université de Caen, 2013.

6 Stephen A Cook. The complexity of theorem-proving procedures. In *Proceedings of the third annual ACM symposium on Theory of computing*, pages 151–158. ACM, 1971.

7 Nadia Creignou and Jean-Jacques Hébrard. On generating all solutions of generalized satisfiability problems. *Informatique théorique et applications*, 31(6):499–511, 1997.

8 Philippe Duchon, Philippe Flajolet, Guy Louchard, and Gilles Schaeffer. Boltzmann samplers for the random generation of combinatorial structures. *Combinatorics, Probability and Computing*, 13(4-5):577–625, 2004.

9 Thomas Eiter, Georg Gottlob, and Kazuhisa Makino. New results on monotone dualization and generating hypergraph transversals. *SIAM Journal on Computing*, 32(2):514–537, 2003.

10 Paul Erdős. On a classical problem of probability theory. 1961.

11 Philippe Flajolet and G Nigel Martin. Probabilistic counting algorithms for data base applications. *Journal of computer and system sciences*, 31(2):182–209, 1985.

12 Michael L Fredman and Leonid Khachiyan. On the complexity of dualization of monotone disjunctive normal forms. *Journal of Algorithms*, 21(3):618–628, 1996.

13 Juris Hartmanis and Richard E Stearns. On the computational complexity of algorithms. *Transactions of the American Mathematical Society*, 117:285–306, 1965.

14 Mark Jerrum. *Counting, sampling and integrating: algorithms and complexity*. Springer Science & Business Media, 2003.

15 David S Johnson, Mihalis Yannakakis, and Christos H Papadimitriou. On generating all maximal independent sets. *Information Processing Letters*, 27(3):119–123, 1988.

16 Daniel M Kane, Jelani Nelson, and David P Woodruff. An optimal algorithm for the distinct elements problem. In *Proceedings of the twenty-ninth ACM SIGMOD-SIGACT-SIGART symposium on Principles of database systems*, pages 41–52. ACM, 2010.

17 Leonid Khachiyan, Endre Boros, Khaled Elbassioni, Vladimir Gurvich, and Kazuhisa Makino. On the complexity of some enumeration problems for matroids. *SIAM Journal on Discrete Mathematics*, 19(4):966–984, 2005.

18 Arnaud Mary. *Énumération des Dominants Minimaux d'un graphe*. PhD thesis, Université Blaise Pascal, 2013.

19 Arnaud Mary and Yann Strozecki. Efficient enumeration of solutions produced by closure operations. In *33rd Symposium on Theoretical Aspects of Computer Science*, 2016.

20 RC Read and RE Tarjan. Bounds on backtrack algorithms for listing cycles, paths, and spanning trees. *Networks*, 5(3):237–252, 1975.

21 Omer Reingold. Undirected connectivity in log-space. *Journal of the ACM (JACM)*, 55(4):17, 2008.

22 Johannes Schmidt. Complexity and enumeration. Master's thesis, Leibniz Universität Hannover, 2009.

23 Yann Strozecki. *Enumeration complexity and matroid decomposition*. PhD thesis, Université Paris Diderot - Paris 7, 2010.

24 Yann Strozecki. On enumerating monomials and other combinatorial structures by polynomial interpolation. *Theory of Computing Systems*, 53(4):532–568, 2013.

25 R. Zippel. Interpolating polynomials from their values. *Journal of Symbolic Computation*, 9(3):375–403, 1990.