Geometric Amortization of Enumeration Algorithms

Florent Capelli
Université de Lille, CNRS, Inria, Centrale Lille, UMR 9189 – CRIStAL, F-59000 Lille, France

Yann Strozecki
Université Paris Saclay, UVSQ, DAVID, France

Abstract

In this paper, we introduce a technique we call geometric amortization for enumeration algorithms, which can be used to make the delay of enumeration algorithms more regular with little overhead on the space it uses. More precisely, we consider enumeration algorithms having incremental linear delay, that is, algorithms enumerating, on input x, a set $A(x)$ such that for every $t \leq \sharp A(x)$, it outputs at least t solutions in time $O(t \cdot p(|x|))$, where p is a polynomial. We call p the incremental delay of the algorithm. While it is folklore that one can transform such an algorithm into an algorithm with maximal delay $O(p(|x|))$, the naive transformation may use exponential space. We show that, using geometric amortization, such an algorithm can be transformed into an algorithm with delay $O(p(|x|) \log(\sharp A(x)))$ and space $O(s \log(\sharp A(x)))$ where s is the space used by the original algorithm. In terms of complexity, we prove that classes DelayP and IncP$_1$ with polynomial space coincide.

We apply geometric amortization to show that one can trade the delay of flashlight search algorithms for their average delay up to a factor of $O(\log(\sharp A(x)))$. We illustrate how this tradeoff is advantageous for the enumeration of solutions of DNF formulas.

2012 ACM Subject Classification Theory of computation → Complexity classes

Keywords and phrases Enumeration, Polynomial Delay, Incremental Delay, Amortization

Digital Object Identifier 10.4230/LIPIcs.STACS.2023.18

Related Version *Full Version*: `https://arxiv.org/abs/2108.10208`

Supplementary Material
InteractiveResource (Visualization): `http://florent.capelli.me/coussinet/`

Funding This work was supported by project ANR KCODA, ANR-20-CE48-0004.

Acknowledgements We would like to thank anonymous reviewers for helpful comments on previous version of the paper and Nofar Carmeli and Stefan Mengel for proofreading earlier version of the paper.

1 Introduction

An enumeration problem is the task of listing a set of elements without redundancies. It is an important and old class of problems: the Baguenaudier game [28] from the 19th century can be seen as the problem of enumerating integers in Gray code order. Ruskey even reports [33] on thousand-year-old methods to list simple combinatorial structures such as the subsets or the partitions of a finite set. Modern enumeration algorithms date back to the 1970s with algorithms computing circuits or spanning trees of a graph [36, 32], while fundamental complexity notions for enumeration have been formalized 30 years ago by Johnson, Yannakakis and Papadimitriou [23]. The main specificity of enumeration problems is that the size of the enumerated set is typically exponential in the size of the input. Hence, a problem is considered tractable and said to be *output polynomial* when it can be solved in time polynomial in the size of the *input and the output.* This measure is relevant when one wants to generate and store all elements of a set, for instance to build a library of objects later to be analyzed by experts, as it is done in biology, chemistry, or network analytics [2, 6, 9].

For most problems, the set to enumerate is too large, or may not be needed in its entirety. It is then desirable to efficiently generate a part of the set for statistical analysis or on the fly processing. In this case, a more relevant measure of the complexity and hence of the quality of the enumeration algorithm is its *delay*, that is, the time spent between two consecutive outputs. One prominent focus has been to design algorithms whose delay is bounded by a polynomial in the size of the input. Problems admitting such algorithms constitute the class DelayP and many problems are in this class, for example the enumeration of the maximal independent sets of a graph [23], or answer tuples of restricted database queries [19] (see [39] for many more examples).

It also happens that new elements of the output set, also called solutions, become increasingly difficult to find. In this case, polynomial delay is usually out of reach but one may still design algorithms with *polynomial incremental time.* An algorithm is in polynomial incremental time if for every i, the delay between the output of the i^{th} and the $(i+1)^{\text{st}}$ solution is polynomial in i and in the size of the input. Such algorithms naturally exist for saturation problems: given a set of elements and a polynomial time function acting on tuples of elements, produce the closure of the set by the function. One can generate such a closure by iteratively applying the function until no new element is found. As the set grows bigger, finding new elements becomes harder. The best algorithm to generate circuits of a matroid uses a closure property of the circuits [24] and is thus in polynomial incremental time. The fundamental problem of generating the minimal transversals of a hypergraph can also be solved in quasi-polynomial incremental time [21, 8] and some of its restrictions in polynomial incremental time [20]. In this paper, the class of problems which can be solved with polynomial incremental time is denoted by UsualIncP.

While the delay is a natural way of measuring the quality of an enumeration algorithm, it might sometimes be too strong of a restriction. Indeed, if the enumeration algorithm is used to generate a subset of the solutions, it is often enough to have guarantees that the time needed to generate i solutions is reasonable for every i. For example, one could be satisfied with an algorithm that has the property that after a time $i \cdot p(n)$, it has output at least i solutions, where p is a polynomial and n the input size. In this paper, we refer to this kind of algorithm as IncP_1-enumerators[1] and call $p(n)$ *the incremental delay* of the algorithm.

While polynomial delay enumerators are IncP_1-enumerator, the converse is not true. Indeed, IncP_1-enumerators do not have to output their solutions regularly. Take for example an algorithm that, on an input of size n, outputs 2^n solutions in 2^n steps, then nothing for 2^n steps and finally outputs the last solution. It can be readily verified that this algorithm outputs at least i solutions after $2i$ steps for every $i \leq 2^n + 1$, and it is thus an IncP_1-enumerator. However, the delay of such an algorithm is not polynomial as the time spent between the output of the last two solutions is 2^n. Instead of executing the output instruction of this algorithm, one could store the solutions that are found in a queue. Then, every two steps of the original algorithm, one solution is removed from the queue and output. The IncP_1 property ensures that the queue is never empty when dequeued and we now have polynomial delay. Intuitively, the solutions being dense in the beginning, they are used to pave the gap until the last solution is found. While this strategy may always be applied to turn an IncP_1-enumerator into a polynomial delay algorithm, the size of the queue may

[1] The 1 in IncP_1 stands for the linear dependency of the incremental time in the number of solutions.

become exponential in the size of the input. In the above example, after the simulation of 2^n steps, 2^n solutions have been pushed into the queue but only 2^{n-1} of them are output, so the queue still contains 2^{n-1} solutions. Unfortunately, an algorithm using exponential space may not be feasible. Therefore, much effort has been devoted to ensure that polynomial delay methods run with polynomial space [27, 3, 15, 17, 10].

Contributions. The main result of this paper is a proof that the regularization of an IncP_1-enumerator may be done without exponentially increasing the space used. More formally, we show that the class $\mathrm{DelayP}^{\mathrm{poly}}$ of problems that can be solved by a polynomial delay and polynomial space algorithm is the same as the class $\mathrm{IncP}_1^{\mathrm{poly}}$ of problems that can be solved by a polynomial space IncP_1-enumerator. In other words, we prove $\mathrm{DelayP}^{\mathrm{poly}} = \mathrm{IncP}_1^{\mathrm{poly}}$, answering positively a question we raised in [11] and where only special cases were proven. Our result relies on a constructive method that we call *geometric amortization*. It turns any IncP_1-enumerator into a polynomial delay algorithm whose delay and space complexity are the incremental delay and space complexity of the original enumerator multiplied by a factor of $\log(S)$, where S is the number of solutions (Theorem 3). Interestingly, we also show that the total time can be asymptotically preserved.

We also apply geometric amortization to transform the average delay of many DelayP-enumerators into a guaranteed delay. Indeed, we show that some widely used algorithmic techniques to design DelayP algorithms also have an incremental delay that matches their average delay. Thus, using geometric amortization, we show that the delay of such an enumerator can be traded for their average delay multiplied by the logarithm of the number of solutions. We apply this result to an algorithm solving Π_{DNF} [12], the problem of listing the models of a DNF formula. This gives an algorithm with sublinear delay and polynomial memory, answering an open question of [12].

The main consequence of our result is that it makes proving that an enumeration problem is in $\mathrm{DelayP}^{\mathrm{poly}}$ easier as one does not have to design an algorithm with polynomial delay but only with polynomial incremental delay. One side-effect of our transformation however is that the order the solutions are output in is changed which may have some practical consequences when used. However, we do not see this as a downside. Actually, we do not believe our method should be used in practice as we cannot see any advantages of having an algorithm with polynomial delay over one with polynomial incremental delay, a notion that we find more natural. This opinion may not be shared by everyone and the main point of our result is to show that from a purely theoretical point of view, it actually does not matter as both notions are – and it came as a surprise to us – the same.

Related work. The notion of polynomial incremental delay is natural enough to have appeared before in the literature. In her PhD thesis [22], Goldberg introduced the notion of *polynomial cumulative delay*, which exactly corresponds to our notion of polynomial incremental delay. We however decided to stick to the terminology of [11]. Goldberg mentions on page 10 that one can turn a linear incremental algorithm into a polynomial delay algorithm but at the price of exponential space. She argues that one would probably prefer in practice incremental delay and polynomial space to polynomial delay and exponential space. Interestingly, she also designs for every constant k, an IncP_1-algorithm with polynomial space to enumerate on input n, every graph that is k-colorable (Theorem 15 on page 112). She leaves open the question of designing a polynomial delay and polynomial space algorithm for this problem, which now comes as a corollary of our theorem applied to her IncP_1-algorithm.

In [37], Tziavelis, Gatterbauer and Riedewald introduce the notion of *Time-To-k* to describe the time needed to output the k best answers of a database query for every k. They design algorithms having a Time-To-k complexity of the form $\mathrm{poly}(n)k$ where n is the size of the input, which hence corresponds to the notion of IncP_1. They argue that delay is sufficient but not necessary to get good Time-to-k complexity and they argue that in practice, having small Time-to-k complexity is better than having small delay. Observe however that in their case, our method does not apply well since they are interested in the *best* answers, meaning that the order is important in this context. Our method does not preserve order.

Organization of the paper. In Section 2, we introduce enumeration problems, the related computational model and complexity measures. Section 3 presents different techniques to turn an IncP_1-enumerator into a DelayP-enumerator using a technique called geometric amortization. Interactive visualization of how geometric amortization works can be found at `http://florent.capelli.me/coussinet/`. In Section 3.3 we apply geometric amortization to incremental polynomial algorithms, showing that our result generalizes to the IncP_i hierarchy. Section 4 gives a method to transform many DelayP-enumerators of average delay $a(n)$ into a DelayP-enumerator with maximal delay $a(n)\log(K)$ where K is the number of solutions. We use it to obtain an algorithm for the problem of enumerating the models of a DNF formula. To outline the main ideas of our algorithms, they are presented using pseudocode with instructions to simulate a given Random Access Machin (RAM). The details on the complexity of using such instructions with minimal overhead are given in the appendix.

2 Preliminaries

Enumeration problems. Let Σ be a finite alphabet and Σ^* be the set of finite words built on Σ. We denote by $|x|$ the length of $x \in \Sigma^*$. Let $A \subseteq \Sigma^* \times \Sigma^*$ be a binary predicat. 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. The element x is called the *instance* or the *input*, while an element of $A(x)$ is called a *solution*. We denote the cardinality of a set $A(x)$ by $\sharp A(x)$.

A predicate A is said to be *polynomially balanced* if for all $y \in A(x)$, $|y|$ is polynomial in $|x|$. It implies that $\sharp A(x)$ is bounded by $|\Sigma|^{\mathrm{poly}(|x|)}$. Let $\textsc{Check}{\cdot}A$ be the problem of deciding, given x and y, whether $y \in A(x)$. The class EnumP, a natural analogous to NP for enumeration, is defined to be the set of all problems Π_A where A is polynomially balanced and $\textsc{Check}{\cdot}A \in \mathrm{P}$. More details can be found in [11, 35].

Computational model. In this paper, we use the Random Access Machine (RAM) model introduced by Cook and Reckhow [18] with comparison, addition, subtraction and multiplication as its basic arithmetic operations augmented with an $\mathtt{OUTPUT}(i, j)$ operation which outputs the content of the values of registers $R_i, R_{i+1}, \ldots, R_j$ as in [4, 34] to capture enumeration problems. We use an hybrid between *uniform cost model* and *logarithmic cost model* (see [18, 1]): output, addition, multiplication and comparison are in constant time on values less than n, where n is the size of the input. In first-order query problems, it is justified by bounding the values in the registers by n times a constant [19, 4]. However, it is not practical for general enumeration algorithms which may store and access 2^n solutions and thus need to deal with large integers. Hence, rather than bounding the register size, we define the cost of an instruction to be the sum of the size of its arguments *divided by* $\log(n)$. Thus, any operation on a value polynomial in n can be done in constant time, but unlike in the usual uniform cost model, we take into account the cost of dealing with superpolynomial values.

A RAM M on input $x \in \Sigma^*$ produces a sequence of outputs $y_1, \ldots, y_S$. The set of outputs of M is denoted by $M(x)$ and its cardinality by $\sharp M(x)$. We say that M solves Π_A if, on every input $x \in \Sigma^*$, $A(x) = M(x)$ and for all $i \neq j$ we have $y_i \neq y_j$, that is *no solution is repeated*. All registers are initialized with zero. The space used by the M is the sum of the bitsize of the integers stored in its registers, up to the register of the largest index accessed.

We denote by $T_M(x, i)$ the time taken by the machine M on input x before the i^{th} `OUTPUT` instruction is executed. When the machine is clear from the context, we drop the subscript M and write $T(x, i)$. The *delay* of a RAM which outputs the sequence $y_1, \ldots, y_S$ is the maximum over all $i \leq s$ of the time spent between the generation of y_i and y_{i+1}, that is $\max_{1 \leq i \leq S} T(x, i+1) - T(x, i)$. The *preprocessing* is $T_M(x, 1)$, the time spent before the first solution is output. The *postprocessing* is the time spent between the output of the last solution and the end of the computation. To simplify the presentation, we assume that there is no postprocessing, that is, a RAM solving an enumeration problem stops right after having output the last solution. This assumption does not affect the complexity classes studied in this paper, as the output of the last solution can be delayed to the end of the algorithm.

Pseudocode. In this paper, we describe our algorithms using pseudocode that is then compiled to a RAM with the usual complexity guarantees. In our algorithms, we freely use variables and the usual control flow instructions, arithmetic operations and data structures. We also assume that we have access to an $\mathsf{output}(s)$ instruction which outputs the value of variable s. When compiled, this instruction calls the `OUTPUT` instruction of the RAM on the registers holding the value of s.

As this paper mostly deals with transforming a given enumeration algorithm into another one having better complexity guarantees, it is convenient to call an algorithm as an oracle to execute it step by step. Therefore, we use two other instructions in our pseudocode: load and move. The instruction $\mathsf{load}(I, x)$ takes two parameters: the first one is the code of a RAM and the second one is its input. It returns a data structure M that can later be used with the move instruction: $\mathsf{move}(M)$ simulates the next step of the computation of machine I on input x. We assume that $\mathsf{move}(M)$ returns false if the computation is finished. We also assume that we have access to the following functions on M:

- $\mathsf{sol}(M)$ returns the solution that M has just output. If the last simulated step of M was not an output instruction, it returns undef. We abuse notation by writing **if**$(\mathsf{sol}(M))$ **then** ... to express the fact that we explore the **then** branch if and only if $\mathsf{sol}(M)$ is not undef.
- $\mathsf{steps}(M)$ returns the number of steps of M that have been simulated.

If we have an upper bound $u(|x|)$ on the memory used by a machine of code I on input x, and if u is computable in time $t(|x|)$, we can implement load and move on a RAM with respective complexity $O(t(|x|))$ and $O(1)$ and space $O(u(|x|))$. Indeed, it is sufficient to reserve $u(|x|)$ contiguous registers in memory and shift all registers used by I so that it uses the reserved memory.

It is also possible to implement these instructions without knowing in advance the memory used by I but one has to use data structures able to dynamically adjust the memory used. In this case, move can be executed either in $O(1)$ with a small space overhead or in $O(\log(\log(|x|)))$ with no space overhead. We leave this improvement for a long version of this article (see [13]) and state the main results when a polynomial time computable upper bound $u(|x|)$ on the memory is known.

Complexity measures and classes. Complexity measures and the relevant complexity classes for enumeration have been formally introduced by Johnson, Yanakakis and Papadimitriou in [23] first. The *total time*, that is $T_M(x, \sharp A(x))$, is similar to what is used for the complexity of decision problems. Since the number of solutions can be exponential in the *input* size, it is more relevant to give the total time as a function of the combined size of the *input and output*. However, this notion does not capture the dynamic nature of an enumeration algorithm. When generating all solutions already takes too long, we want to be able to generate at least some solutions. Hence, we should measure (and bound) the total time used to produce a given number of solutions. We give here the notion of linear incremental time, central to the paper, while the more general notion of polynomial incremental time is given and studied in Section 3.3.

▶ **Definition 1** (Linear incremental time). *A problem $\Pi_A \in \mathrm{EnumP}$ is in IncP_1 if there is a polynomial d and a machine M which solves Π_A, such that for all x and for all $1 < i \leq \sharp A(x)$, $T(x,i) < i \cdot d(|x|)$ and $T(x,1) < d(|x|)$. Such a machine M is called an IncP_1-enumerator with incremental delay $d(n)$.*

▶ **Definition 2** (Polynomial delay). *A problem $\Pi_A \in \mathrm{EnumP}$ is in DelayP if there is a polynomial d and a machine M which solves Π_A, such that for all x and for all $1 < i \leq \sharp A(x)$, $T(x,i) - T(x,i-1) \leq d(|x|)$ and $T(x,1) \leq d(|x|)$. Such a machine M is called a DelayP-enumerator of delay $d(n)$.*

Observe that if M is a DelayP-enumerator then for all i we have $T(x,i) - T(x,1) \leq \sum_{1<j\leq i} d(|x|) = (i-1)d(|x|)$. Hence $\mathrm{DelayP} \subseteq \mathrm{IncP}_1$. Polynomial delay is the most common notion of tractability in enumeration, because it guarantees both regularity and linear total time and also because it is relatively easy to prove that an algorithm has a polynomial delay. Indeed, most methods used to design enumeration algorithms such as backtrack search with a polynomial time extension problem [29], or efficient traversal of a supergraph of solutions [27, 3, 16], yield polynomial delay algorithms on many enumeration problems.

To better capture the notion of tractability in enumeration, it is important to use polynomial space algorithms. We let $\mathrm{DelayP}^{\mathrm{poly}}$ be the class of problems solvable by a polynomial space DelayP-enumerator. We define $\mathrm{IncP}_1^{\mathrm{poly}}$, as the class of problems which can be solved by a polynomial space IncP_1-enumerator.

3 From IncP_1 to DelayP

3.1 Geometric Amortization

The folklore method (e.g., [23, 34, 14]) used to transform an IncP_1-enumerator into a DelayP-enumerator that was sketched in the introduction uses a queue to delay the output of solutions. This queue may however become of size exponential in the input size. To overcome this issue, we introduce a technique that we call *geometric amortization*, illustrated by Algorithm 1 which regularizes the delay of an IncP_1-enumerator with a space overhead of $O(\log(\sharp I(x)))$, which is polynomially bounded since I is in EnumP. To achieve this, we however have to compromise a bit on the delay which becomes $O((\log(\sharp I(x)) \cdot p(|x|))$. Moreover, with geometric amortization, the solutions are not output in the same order as the order they are output by I. Algorithm 1 relies on the knowledge of an upper bound K of $\sharp I(x)$, but this assumption is relaxed in Section 3.2. We now proceed to prove the correctness and complexity of Algorithm 1 that is summarized in the theorem below.

Algorithm 1 Using geometric amortization for regularizing the delay of an IncP_1-enumerator I having incremental delay $p(n)$ only using polynomial space. In the code, $Z_0 = [0; p(n)]$ and $Z_j = [2^{j-1}p(n) + 1; 2^j p(n)]$ for $j > 0$.

```
Input  : $x \in \Sigma^*$ of size $n$ and $K$ such that $K \geq \sharp I(x)$
Output : Enumerate $I(x)$ with delay $O(p(n) \cdot \log(K))$
begin
    $N \leftarrow \lceil \log(K) \rceil$;
    for $i = 0$ to $N$ do  $M[i] \leftarrow \mathsf{load}(I, x)$ ;
    $j \leftarrow N$;
    while $j \geq 0$ do
        for $b \leftarrow 2p(n)$ to $0$ do
            $\mathsf{move}(M[j])$;
            if $\mathsf{sol}(M[j])$ and $\mathsf{steps}(M[j]) \in Z_j$ then
                $\mathsf{output}(\mathsf{sol}(M[j]))$;
                $j \leftarrow N$;
                break;
        if $b = 0$ then $j \leftarrow j - 1$;
```

▶ **Theorem 3.** *Given an* IncP_1*-enumerator* I *with incremental delay* $p(n)$ *and space complexity* $s(n)$ *and given* $K \geq \sharp I(x)$*, one can construct a* DelayP*-enumerator* I' *which enumerates* $I(x)$ *on any input* $x \in \Sigma^*$ *with delay* $O(\log(K)p(n))$ *and space complexity* $O(s(n)\log(K))$*.*

Proof. The pseudo-code for I', accessing an oracle to I as a blackbox, is presented in Algorithm 1. Its correctness and complexity are proven in the rest of this section. ◀

Since $\text{IncP}_1 \subseteq \text{EnumP}$, we know that there is a polynomial $q(n)$ such that every element of $I(x)$ is of size at most $q(|x|)$ and by choosing $K = |\Sigma|^{q(n)}$, we have that $\log(K)$ is polynomially bounded and the following is a direct corollary of Theorem 3:

▶ **Corollary 4.** $\text{IncP}_1^{\text{poly}} = \text{DelayP}^{\text{poly}}$.

The construction of I' from I in Theorem 3 is presented in Algorithm 1, which uses a technique that we call *geometric amortization*. The idea of geometric amortization is to simulate several copies of I on input x at different speeds. Each process is responsible for enumerating solutions in different intervals of time to avoid repetitions in the enumeration. The name comes from the fact that the size of the intervals we use follows a geometric progression (the size of the $(i+1)^{\text{th}}$ interval is twice the size of the i^{th} one).

Explanation of Algorithm 1. Algorithm 1 maintains $N+1$ simulations $M[0], \ldots, M[N]$ of I on input x where $N = \lceil \log(K) \rceil$. When simulation $M[i]$ finds a solution, it outputs it if and only if the number of steps of $M[i]$ is in Z_i, where $Z_i := [1 + 2^{i-1}p(n), 2^i p(n)]$ for $i > 0$ and $Z_0 = [1, p(n)]$. These intervals are clearly disjoint and cover every possible step of the simulation since the total time of I is at most $\sharp I(x)p(n) \leq Kp(n) \leq 2^N p(n)$ (by convention, we assumed enumerators to stop on their last solution, see Section 2). Thus, every solution is enumerated as long as $M[i]$ has reached the end of Z_i when the algorithm stops.

Algorithm 1 starts by moving $M[N]$. It is given a budget of $2p(n)$ steps. If these $2p(n)$ steps are executed without finding a solution in Z_N, $M[N-1]$ is then moved similarly with a budget of $2p(n)$ steps. It continues until one machine $M[j]$ finds a solution in its zone Z_j. In this case, the solution is output and the algorithm proceeds back with $M[N]$. The algorithm stops when $M[0]$ has left Z_0, that is when $p(n)+1$ steps of $M[0]$ have been simulated[2].

Bounding the delay. From the above description of Algorithm 1, between two outputs, the variable j takes at most $N+1$ values (from N to 0) and at most $2p(n)$ move instructions are executed for each machine $M[j]$. A move instruction can be executed in $O(1)$ (see Appendix A). Moreover, the size of b being $O(\log(n))$, we can increment it in $O(1)$ in the RAM model we consider. Finally, we have to compare $\mathsf{steps}(M[i])$ with integers of values in $O(\log(K)p(n))$. Manipulating such integers in the RAM model would normally cost $O(\log(K)/\log(n))$. However, we give in Appendix A.2 a method using Gray Code encodings which allows us to increment $\mathsf{steps}(M[i])$ and to detect when it enters and exits Z_i in $O(1)$. Thus, the overall delay of Algorithm 1 is $O(\log(K)p(n))$.

Space complexity. We have seen in Section 2 that a RAM can be simulated without using more space than the original machine (see Appendix A for more details). Since Algorithm 1 uses $O(\log(K)$ simulations of I, its space complexity is $O(s(n)\log(K))$.

Correctness of Algorithm 1. It remains to show that Algorithm 1 correctly outputs $I(x)$ on input x. Recall that a solution of $I(x)$ is enumerated by $M[i]$ if it is produced by I at step $c \in Z_i = [1+2^{i-1}p(n), 2^i p(n)]$. Since, by definition, the total time of I on input x is at most $\sharp I(x)p(n)$, it is clear that $Z_0 \uplus \cdots \uplus Z_N \supseteq [1, Kp(n)] \supseteq [1, \sharp(I)p(n)]$ covers every solution and that each solution is produced at most once. Thus, it remains to show that when the algorithm stops, $M[i]$ has moved by at least $2^i p(n)$ steps, that is, it has reached the end of Z_i and output all solutions in this zone.

We study an execution of Algorithm 1 on input x. For the purpose of the proof, we only need to look at the values of $\mathsf{steps}(M[0]), \dots, \mathsf{steps}(M[N])$ during the execution of the algorithm. We thus say that the algorithm is in state $c = (c_0, \dots, c_N)$ if $\mathsf{steps}(M[i]) = c_i$ for all $0 \leq i \leq N$. We denote by S_i^c the set of solutions that have been output by $M[0], \dots, M[i]$ when state c is reached; that is, a solution is in S_i^c if and only if it is produced by I at step $k \in Z_j$ for $j \leq i$ and $k \leq c_j = \mathsf{steps}(M[j])$. We claim the following invariant:

▶ **Lemma 5.** *For every state c and $i < N$, we have $c_{i+1} \geq 2p(n)|S_i^c|$.*

Proof. The proof is by induction on c. For the state c just after initializing the variables, we have that for every $i \leq N$, $|S_i^c| = 0$ and $c_i = 0$. Hence, for $i < N$, $c_{i+1} \geq 0 = 2p(n)|S_i^c|$.

Now assume the statement holds at state c' and let c be the next state. Let $i < N$. If $|S_i^c| = |S_i^{c'}|$, then the inequality still holds since $c_{i+1} \geq c'_{i+1}$ and $c'_{i+1} \geq 2p(n)|S_i^{c'}| = 2p(n)|S_i^c|$ by induction. Otherwise, we have $|S_i^c| = |S_i^{c'}| + 1$, that is, some simulation $M[k]$ with $k \leq i$ has just output a solution. In particular, the variable j has value $k \leq i < N$. Let c'' be the first state before c' such that variable j has value $i+1$ and b has value $2p(n)$, that is, c'' is the state just before Algorithm 1 starts the for loop to move $M[i+1]$ by $2p(n)$ steps. No solution has been output between state c'' and c' since otherwise j would have been

[2] An illustration of Algorithm 1 can be found at `http://florent.capelli.me/coussinet/` where one can see the run of a machine represented as a list and the different simulations moving in this list and discovering solutions.

reset to N. Thus, $|S_i^{c''}| = |S_i^{c'}|$. Moreover, $c_{i+1} \geq c'_{i+1} \geq c''_{i+1} + 2p(n)$ since $M[i+1]$ has moved by $2p(n)$ steps in the for loop without finding a solution. By induction, we have $c''_{i+1} \geq 2p(n)|S_i^{c''}| = 2p(n)|S_i^{c'}|$. Thus $c_{i+1} \geq c''_{i+1} + 2p(n) = 2p(n)(|S_i^{c'}|+1) = 2p(n)|S_i^{c}|$ which concludes the induction. ◀

▶ **Corollary 6.** *Let $c = (c_0, \dots, c_N)$ be the state reached when Algorithm 1 stops. We have for every $i \leq N$, $c_i \geq 2^i p(n)$.*

Proof. The proof is by induction on i. If i is 0, then we necessarily have $c_0 \geq p(n)$ since Algorithm 1 stops only when $M[0]$ has moved outside Z_0, that is when it has been moved by at least $p(n)$ steps.

Now assume $c_j \geq 2^j p(n)$ for every $j < i$. This means that for every $j < i$, $M[j]$ has been moved at least to the end of Z_j. Thus, $M[j]$ has found every solution in Z_j. Since it holds for every $j < i$, it means that $M[0], \dots, M[i-1]$ have found every solution in the interval $K = [1, 2^{i-1}p(n)]$. Since I is an IncP_1-enumerator with delay $p(n)$ and since $2^{i-1} \leq \sharp I(x)$ by definition of N, K contains at least 2^{i-1} solutions, that is, $|S_{i-1}^c| \geq 2^{i-1}$. Applying Lemma 5 gives that $c_i \geq 2^{i-1} \cdot 2p(n) = 2^i p(n)$. ◀

The correctness of Algorithm 1 directly follows from Corollary 6. Indeed, it means that for every $i \leq N$, every solution of $Z_i = [1 + 2^{i-1}p(n), 2^i p(n)]$ have been output, that is, every solution of $[1, 2^N p(n)]$ and $2^N p(n)$ is an upper bound on the total run time of I on input x.

Observe that Algorithm 1 does not preserve the order of the solutions since it interleaves solutions later seen in the enumeration process in order to amortize large delay between two solutions. One possible workaround is to output pairs of the form (s, r) where s is a solution and r its rank in the original enumeration order. To do so, one just has to keep a counter of the number of solutions seen so far by each process and output it along a solution. It does not restore the order, but allows recovering it afterward. When we are interested in finding the first K solutions only (a likely scenario for solving top-k problems), the previous workaround is not useful. However, our algorithm can output them only with a $\log(K)$ overhead by just running $\log(K)$ processes and ignoring solutions having a rank higher than K; the solutions will however not be output in order.

3.2 Improving Algorithm 1

One drawback of Algorithm 1 is that it needs to know in advance an upper bound K on $\sharp I(x)$ since it uses it to determine how many simulations of I it has to maintain. In theory, such an upper bound exists because I is assumed to be in EnumP and it is often known, *e.g.*, $|\Sigma|^N$ where N is an upper bound on the size of the output. In practice, however, it might be cumbersome to compute it or it may hurt efficiency if the upper bound is overestimated. It turns out that one can remove this hypothesis by slightly modifying Algorithm 1. The key observation is that during the execution of the algorithm, if $M[i]$ has not entered Z_i, it is simulated in the same way as $M[i+1], \dots, M[N]$. Indeed, it is not hard to see that $M[j]$ is always ahead of $M[i]$ for $j > i$ and that if $M[i]$ is not in Z_i, it will not output any solution in the loop at Line 6, hence this iteration of the loop will move $M[i]$ by $2p(n)$ steps, just like $M[j]$ for $j > i$. Hence, Algorithm 1 can be improved in the following way: we start with only two simulations $M[0], M[1]$ of I. Whenever $M[1]$ is about to enter Z_1, we start $M[2]$ as an independent copy of $M[1]$. During the execution of the algorithm, we hence maintain a list M of simulations of I and each time the last simulation $M[N]$ is about to enter Z_N, we copy it into a new simulation $M[N+1]$. The hardest part of implementing this idea is to

Algorithm 2 Improvement of Algorithm 1 which works without upper bounds on the number of solutions and has a better total time. In the code, $a_0 = 0$ and $a_j = 2^{j-1}p(n) + 1$.

```
    Input  : $x \in \Sigma^*$ of size $n$
    Output : Enumerate $I(x)$ with delay $O(p(n) \cdot \log(\sharp I(x)))$
 1  begin
 2      $M \leftarrow$ list($\emptyset$);
 3      insert($M$, load($I, x$));
 4      $j \leftarrow$ length($M$) $- 1$;
 5      while $j \geq 0$ do
 6          for $b \leftarrow 2p(n)$ to $0$ do
 7              move($M[j]$);
 8              if $j =$ length($M$) $- 1$ and steps($M[j]$) $= a_j$ then
 9                  insert($M$, copy($M[j]$));
10                  $j \leftarrow$ length($M$) $- 1$;
11                  break;
12              if sol($M[j]$) and steps($M[j]$) $\in [a_j; a_{j+1} - 1]$ then
13                  output(sol($M[j]$));
14                  $j \leftarrow$ length($M$) $- 1$;
15                  break;
16          if $b = 0$ then $j \leftarrow j - 1$;
```

show that one can copy simulation $M[N]$ without affecting the overall delay of the algorithm. That can be achieved by lazily copying parts of $M[N]$ whenever we move $M[N+1]$. The details are given in Appendix A.4.

By implementing this idea, one does not need to know an upper bound on $\sharp I(x)$ anymore: new simulations will be created as long as it is necessary to discover new solutions ahead. The fact that one has found every solution is still witnessed by the fact that $M[0]$ reaches the end of Z_0. This improvement has yet another advantage compared to Algorithm 1: it has roughly the same total time as the original algorithm. Hence, if one is interested in generating every solution with a polynomial delay from an IncP$_1$-enumerator, our method may make the maximal delay worse but does not change much the time needed to generate all solutions.

Correctness of Algorithm 2. Correctness of Algorithm 2 can be proven in a similar way as for Algorithm 1. Lemma 5 still holds for every state, where N in the statement has to be replaced by $\mathsf{length}(M) - 1$. The proof is exactly the same but we have to verify that when a new simulation is inserted into M, the property still holds. Indeed, let c be a state that follows the insertion of a new simulation (Line 8). We have now $\mathsf{length}(M) - 1 = N + 1$ (thus the last index of M is $N + 1$). Moreover, we claim that $S^c_{N+1} = S^c_N$. Indeed, at this point, the simulation $M[N + 1]$ has not output any solution. Moreover, by construction, $c_N = \mathsf{steps}(M[N]) = \mathsf{steps}(M[N + 1]) = c_{N+1}$. Since $c_N \geq 2p(n)|S^c_N|$ by induction, we have that $c_{N+1} \geq 2p(n)|S^c_{N+1}|$. Moreover, the following adaptation of Corollary 6 holds for Algorithm 2.

▶ **Lemma 7.** *Let c be the state reached when Algorithm 1 stops. Then $N := \mathsf{length}(M) - 1 = 1 + \log(\sharp I(x))$ and for every $i \leq N$, $c_i \geq 2^i p(n)$.*

Proof. The lower bound $c_i \geq 2^i p(n)$ for $i \leq N$ is proven by induction exactly as in the proof of Lemma 5. The induction holds as long as $2^{i-1} \leq \sharp I(x)$, because we need this assumption to prove that there are at least 2^{i-1} solutions in the interval $[1, 2^{i-1}p(n)]$. Now, one can easily see that if $i \leq 1 + \log(\sharp I(x))$ and $c_i \geq 2^i p(n)$ then the simulation $M[i]$ has reached $2^{i-1}p(n)$ at some point and thus, has created a new simulation $M[i+1]$. Thus, by induction, the algorithms creates at least $1 + \log(\sharp I(x)) = N$ new simulations. Thus $\mathsf{length}(M) \geq N + 1$ (as M starts with one element).

Finally, observe that $M[N]$ outputs solutions in the zone $Z_N = [2^{N-1}p(n) + 1, 2^N p(n)]$ and that $2^{N-1}p(n) = \sharp I(x)p(n)$ which is an upper bound on the total time of I on input x. Thus, the simulation $M[N]$ will end without creating a new simulation. In other words, $\mathsf{length}(M) - 1 = N$. ◀

Delay of Algorithm 2. While establishing the correctness of Algorithm 2 is similar to the one of Algorithm 1, proving a bound on the delay of Algorithm 2 is not as straightforward. By Lemma 7, the size of M remains bounded by $2 + \log(\sharp I(x))$ through the algorithm, so there are at most $2p(n)(2 + \log(\sharp I(x)))$ executions of move between two solutions, for the same reasons as before. However, we also have to account for the execution of copy. When implemented naively, this operation requires a time $O(s(n))$ to copy the entire configuration of the simulation in some fresh part of the memory. It would add $O(s(n))$ to the delay of Algorithm 2 compared to Algorithm 1. However, one can amortize this copy operation by lazily copying the memory while running the original simulation and by adapting the sizes of the zones so that we can still guarantee a delay of $O(\log(\sharp I(x))p(n))$ in Algorithm 2. The method is formally described in Appendix A.4.

Total time of Algorithm 2. A minor modification of Algorithm 2 improves its efficiency in terms of total time. By definition, when simulation $M[i]$ exits Z_j, it does not output solutions anymore. Thus, it can be removed from the list of simulations. It does not change anything concerning the correctness of the algorithm. One just has to be careful to adapt the bounds in Algorithm 2. Indeed, $2^j p(n)$ is not the right bound anymore as removing elements from M may shift the positions of the others. It can be easily circumvented by also maintaining a list Z such that $Z[i]$ always contains the zone that $M[i]$ has to enumerate.

By doing it, it can be seen that each step of I having a position in Z_i will only be visited by two simulations: the one responsible for enumerating Z_i and the one responsible for enumerating Z_{i+1}. Indeed, the other simulations would either be removed before entering Z_i or will be created after the last element of M has entered Z_{i+1}. Thus, the move operation is executed at most $2T(|x|)$ times where $T(|x|)$ is the total time taken by I on input x and the total time of this modification of Algorithm 2 is $O(T(n))$ where $T(n)$ is the total time of I.

All previous comment on Algorithm 2 allows us to state the following improvement of Theorem 3, where no upper bound on $\sharp I(x)$ is necessary but $s(n)$ and $p(n)$ are known.

▶ **Theorem 8.** *Given an* IncP$_1$*-enumerator I with incremental delay $p(n)$, space complexity $s(n)$ and total time $T(n)$, one can construct a* DelayP*-enumerator I' which enumerates $I(x)$ on any input $x \in \Sigma^*$ with space complexity $O(s(n)\log(\sharp I(x)))$, delay $O(\log(\sharp I(x))p(n))$ and total time $O(T(n))$.*

We observe that Algorithm 2 can be modified so that it can work with IncP$_1$-enumerators having a preprocessing. Indeed one only needs, as a preprocessing step of Algorithm 2, to run the first simulation created by the algorithm until it outputs its first solution to be in the same state as the case where there is no preprocessing.

We sill need to know two parameters (or an upper bound on them) to run Algorithm 2: the space of the amortized algorithm and its incremental delay. By using dynamic data structures, one could adapt our algorithm when the space used by the enumerator is not known for a very small overhead. Moreover, it is possible to give a lower bound showing that one cannot get a $O(p(n))$ polynomial delay when the incremental delay $p(n)$ is unknown (if I is a blackbox). We leave this improvement for a long version of this article (see [13]).

3.3 Geometric Amortization for IncP_i with $i > 1$

The dynamic version of the total time is called *incremental time*: Given an enumeration problem A, we say that a machine M solves Π_A in incremental time $f(i)g(n)$ if on every input x, and for all $i \leq \sharp A(x)$, $T_M(x,i) \leq f(i)g(|x|)$. The linear incremental time corresponds to the case $f(i) = i$. We generalize IncP_1, by polynomially bounding the incremental time.

▶ **Definition 9** (Polynomial incremental time). *A problem* $\Pi_A \in \mathrm{EnumP}$ *is in* IncP_a *if there is a constant b and a machine M which solves it with incremental time $O(i^a n^b)$. Such a machine is called an* IncP_a*-enumerator. Moreover, we define* $\mathrm{IncP} = \bigcup_{a \geq 1} \mathrm{IncP}_a$.

Allowing arbitrary polynomial preprocessing does not modify the class IncP_a since this preprocessing can be interpreted as the polynomial time before outputting the first solution. The class IncP is believed to be strictly included in OutputP, the class of problems solvable in total polynomial time, since this is equivalent to $\mathrm{TFNP} \neq \mathrm{FP}$ [11]. Moreover, the classes IncP_a form a strict hierarchy assuming the exponential time hypothesis [11].

▶ **Definition 10** (Usual definition of incremental time.). *A problem* $\Pi_A \in \mathrm{EnumP}$ *is in* $\mathrm{UsualIncP}_a$ *if there are b and c integers and a machine M which solves* Π_A *such that for all x and for all* $0 < t \leq \sharp A(x)$, $T(x,t) - T(x,t-1) < ct^a|x|^b$.

Our definition of IncP captures the fact that we can generate t solutions in time polynomial in t and in the size of the input, which seems more general than bounding the delay because the time between two solutions is not necessarily regular. Using geometric amortization, we can show that both definitions are equivalent even when the space is required to be polynomial.

For $a \geq 0$, we denote by $\mathrm{IncP}_a^{\mathrm{poly}}$ (respectively $\mathrm{UsualIncP}_a^{\mathrm{poly}}$), the class of problems that can be solved by an IncP_a (respectively $\mathrm{UsualIncP}_a$) algorithm and polynomial space. The following generalises Corollary 4 since $\mathrm{DelayP} = \mathrm{UsualIncP}_0$.

▶ **Theorem 11.** *For all* $a \geq 0$, $\mathrm{IncP}_{a+1}^{\mathrm{poly}} = \mathrm{UsualIncP}_a^{\mathrm{poly}}$.

Proof. The inclusion $\mathrm{UsualIncP}_a^{\mathrm{poly}} \subseteq \mathrm{IncP}_{a+1}^{\mathrm{poly}}$ is straightforward and follows by a simple computation of the time to generate i solutions, see [11].

The inclusion $\mathrm{IncP}_{a+1}^{\mathrm{poly}} \subseteq \mathrm{UsualIncP}_a^{\mathrm{poly}}$ is done by geometric amortization, by adapting Algorithm 1. Let I be an algorithm solving a problem in IncP_{a+1}. We assume we know $t^{a+1}p(n)$, a bound on its incremental time.

Then, the only modification we do in Algorithm 1 is to maintain a counter S of the number of output solutions and modify the initialization of b in the for loop at line 6 to $S^a(a+1)2p(n)$. By construction of the amortization algorithm, the delay between two solutions before the algorithm ends is bounded by $S^a(a+1)2p(n)\log(s)$, where S is the number of solutions output up to this point of the algorithm and s the total number of solutions. Thus, the algorithm is in $\mathrm{UsualIncP}_a$.

We still have to prove that all solutions are enumerated by the algorithm. Assume that the first $i+1$ machines $M[0], \dots, M[i]$ have output all the solutions in their zones, then we prove as in Corollary 6, that the the machine $M[i+1]$ has also output all its solutions. The number of solutions output by $M[0], \dots, M[i]$ is the number of solutions output by I up to time step $2^i p(n)$. Let s_i be this number, then $s_i^{a+1} p(n) \geq 2^i p(n)$ since I is in incremental time $t^{a+1} p(n)$. Hence, $s_i \geq 2^{i/(a+1)}$.

When a solution is output by a machine $M[j]$ with $j \leq i$, then j is set to N and all machines $M[k]$ with $k > i$ move by at least $S^a(a+1)2p(n)$ steps where S is the current number of output solutions before $M[i]$ moves again. Hence, we can lower bound the number of moves of the machine $M[i+1]$ by $\sum_{S=0}^{s_i} S^a(a+1)2p(n) \geq \sum_{S=0}^{2^{i/(a+1)}} S^a(a+1)2p(n)$. Since $\sum_{S=0}^{n} S^a \geq \int_0^n S^a \, dS \geq n^{a+1}/(a+1)$, the number of moves of $M[i+1]$ is larger than $2^{i+1}p(n)$ which is the upper bound of its zone. ◀

4 Other Applications of Geometric Amortization

4.1 Amortizing Self-Reducible Problems

Given an enumeration problem Π_A, we assume from now on, to lighten the exposition, that the solutions in $A(x)$ are sets over some universe $U(x)$. From A, we define the predicate $\tilde{A}$ which contains the pairs $((x, a, b), y)$ such that $y \in A(x)$ and $a \subseteq y \subseteq b$. From this predicate, we define a self-reducible[3] variant of Π_A and the extension problem ExtSol·A defined as the set of triples (x, a, b) such that there is a y in $\tilde{A}(x, a, b)$.

Solving Π_A on input x is equivalent to solving $\Pi_{\tilde{A}}$ on $(x, \emptyset, U(x))$. Let us now formalize a recursive method to solve $\Pi_{\tilde{A}}$, sometimes called *binary partition*, because it partitions the solutions to enumerate in two disjoint sets. Alternatively, it is called *flashlight search*, because we peek at subproblems to solve them only if they yield solutions. To our knowledge, all uses of flashlight search in the literature can be captured by this formalization, except for the partition of the set of solutions which can be in more than two subsets. We only present the binary partition for the sake of clarity, but our analysis can be adapted to finer partitions.

Given an instance (x, a, b) of $\Pi_{\tilde{A}}$ and some global auxiliary data D, a flashlight search consists in the following (subroutines are not specified, and yield different flashlight searches):

- if $a = b$, a is ouput and the algorithm stops
- choose $u \in b \setminus a$;
- if $(x, a \cup \{u\}, b) \in$ ExtSol·A, compute some auxiliary data D_1 from D and make a recursive call on $(x, a \cup \{u\}, b)$;
- if $(x, a, b \setminus \{u\}) \in$ ExtSol·A, compute some auxiliary data D_2 from D_1 and make a recursive call on $(x, a, b \setminus \{u\})$, then compute D from D_2.

Flashlight search can be seen as a depth-first traversal of a *partial solutions tree*. A node of this tree is a pair (a, b) such that $(x, a, b) \in$ ExtSol·A. Node (a, b) has children $(a \cup \{u\}, b)$ and $(a, b \setminus \{u\})$ if they are nodes. A leaf is a pair (a, a) and the root is $(\emptyset, U(x))$. The *cost* of a node (a, b) is the time to execute the flashlight search on (x, a, b) *except the time spent in recursive calls*. Usually, the cost of a node comes from deciding ExtSol·A and modifying the global data structure D used to solve ExtSol·A faster.

[3] For a classical definition of self-reducible problems, see e.g. [25, 5].

The cost of a path in a partial solution tree is the sum of the costs of the nodes in the path. We define the *path time* of a flashlight search algorithm as the maximum over the cost of all paths from the root. Twice the path time bounds the delay since, between two output solutions, a flashlight search traverses at most two paths in the tree of partial solutions. To our knowledge, all bounds on the delay of flashlight search are proved by bounding the path time. The path time is bounded by $\sharp U(x)$ times the complexity of solving $\textsc{ExtSol}{\cdot}A$. Auxiliary data can be used to amortize the cost of evaluating $\textsc{ExtSol}{\cdot}A$ repeatedly, generally to prove that the path time is equal to the complexity of solving $\textsc{ExtSol}{\cdot}A$ once, e.g., when generating minimal models of monotone CNF [31].

Using flashlight search, we obtain that $\Pi_A \in$ DelayP if $\textsc{ExtSol}{\cdot}A \in$ P and indeed many enumeration problems are in DelayP because their extension problem are in P, see e.g., [38, 29]. However, there are NP-hard extension problems whose enumeration problem is in DelayP, e.g., the extension of a maximal clique, whose hardness can be derived from the fact that finding the largest maximal clique in lexicographic order is NP-hard [23].

The *average delay* (also amortized delay or amortized time) of a machine M solving Π_A on input x is $T(x, \sharp A(x))/\sharp A(x)$. The average delay of an enumerator is bounded by its delay but it can be much smaller. This happens in flashlight search when the internal nodes of the tree of partial solutions are guaranteed to have many leaves. Uno describes the pushout method [38] harnessing this property to obtain constant average delay algorithms for many problems such as generating spanning trees.

To make sense of very low complexity enumeration algorithms, we may separate the preprocessing $T(x, 1)$ from the rest of the computation. We say that a machine with preprocessing has incremental delay $d(n)$ if, for all x and i, $T(x, i) - T(x, 1) \leq i \cdot d(|x|)$. The preprocessing is not taken into account in the incremental delay. When the preprocessing time is not zero, it is explicitly specified and we use preprocessing only in this section. We now prove, using Theorem 3, that the average delay of a flashlight search can be turned into a delay up to a small multiplicative factor. It relies on a small queue for amortization, so that its incremental delay is equal to its average delay, and on geometric amortization to turn the incremental delay into a delay.

▶ **Theorem 12.** *Let Π_A be an enumeration problem solved by a flashlight search algorithm, with space $s(n)$, path time $p(n)$ and average delay $d(n)$. Let $b(n)$ be the size of a single solution. There is an algorithm solving Π_A on any input x, with preprocessing $O(p(n)b(n))$, delay $O(d(n) \log(\sharp I(x)))$ and space $O(s(n) \log(\sharp I(x)) + p(n)b(n))$.*

Proof. Let I be the flashlight search algorithm solving Π_A. Let us first describe an algorithm I' in incremental linear time, which produces the same solutions as I on any input x of size n. The preprocessing of I' is to run I for $p(n)$ steps and to store each solution output in a queue. It takes a time at most $O(p(n)b(n))$ since there are at most $p(n)$ solutions of size $b(n)$ to store in the queue. The queue requires an additional space of $O(p(n)b(n))$. After the preprocessing, we first output all solutions in the queue and then I is simulated for the rest of its run and the solutions output by I are output by I' right away.

Checking the queue for emptiness and outputting a solution can be done in constant time. Hence, we can guarantee that there is a constant C, such that after C computation steps of I', one step of I is executed. Let us evaluate the number of solutions output when I' has run for a time Ct after the preprocessing. If at this time the queue is not empty, then a solution has been output at each time step, hence there are at least t output solutions.

If the queue is empty, the number of solutions output by I' is the same as the number of solutions output by I after running for a time $p(n) + t$. At this point in time, the flashlight search is considering some node (a, b) of the partial solutions tree and we denote

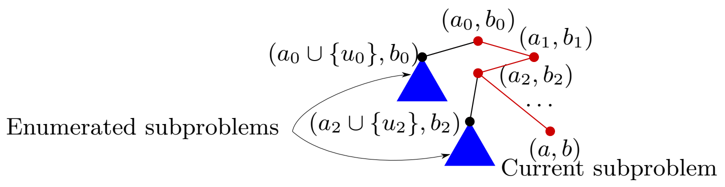

Figure 1 A traversal of the tree of partial solutions by the flashlight search. The subproblems completely solved recursively in blue, the path to the current solution in red.

by $(\emptyset, U(x)) = (a_0, b_0), \ldots, (a_i, b_i) = (a, b)$ the path from the root to (a, b). The time spent on the nodes of this path is bounded by $p(n)$, the path time of I. Hence, I spends at least a time t in the subtrees whose root is a child of some (a_i, b_i).

Also, observe that a subtree rooted at a child (c, d) of (a_i, b_i) with $(c, d) \neq (a_{i+1}, b_{i+1})$ has been either completely explored by the flashlight search or not at all, as shown in Figure 1. Since I is a flashlight search, it works recursively on subtrees, corresponding to subproblems. If a subtree rooted at (c, d) has been completely explored, then the flashlight search has recursively solved the problem $\tilde{A}(x, c, d)$. By definition of the average delay, the solutions in $\tilde{A}(x, c, d)$ have been produced by flashlight search in total time less than $d(n)\sharp\tilde{A}(x, c, d)$. Hence, the subproblems entirely solved by I contribute at least $t/d(n)$ solutions. Therefore, in time Ct, I' outputs at least $t/d(n)$ solutions.

Therefore, we have proven that I' is in incremental delay $O(d(n))$, space $O(s(n)+p(n)b(n))$ and preprocessing $O(p(n)b(n))$. Applying Theorem 3 to I' yields an algorithm with the stated complexity. ◀

4.2 Enumeration of the Models of DNF Formulas

In this section, we explore consequences of Theorem 12 on the problem of generating models of a DNF formula, which has been extensively studied in [12]. Let us denote by n the number of variables of a DNF formula, by m its number of terms and by Π_{DNF} the problem of generating the models of a DNF formula. The size of a DNF formula is at least m and at most $O(mn)$ (depending on the representation and the size of the terms), which can be exponential in n. Hence, we want to understand whether Π_{DNF} can be solved with a delay polynomial in n only, that is depending on the size of a model of the DNF formula but not on the size of the formula itself. A problem that admits an algorithm with a delay polynomial in the size of a single solution is said to be *strongly polynomial* and is in the class SDelayP. One typical obstacle to being in SDelayP is dealing with large non-disjoint unions of solutions. The problem Π_{DNF} is an example of such difficulty: its models are the union of the models of its terms, which are easy to generate with constant delay.

The paper [12] defines the *strong DNF enumeration conjecture* as follows: there is no algorithm solving Π_{DNF} in delay $o(m)$. It also describes an algorithm solving Π_{DNF} in *average* sublinear delay. It is based on flashlight search, with appropriate data structures and choice of variables to branch on (Theorem 10 in [12]). Thanks to Theorem 12, we can trade the average delay for a guaranteed delay and falsify the strong DNF enumeration conjecture.

▶ **Corollary 13.** *There is an algorithm solving Π_{DNF} with linear preprocessing, delay $O(n^2 m^{1-\log_3(2)})$ and space $O(n^2 m)$.*

Proof. The algorithm of [12] enumerates all models with average delay $O(nm^{1-\log_3(2)})$ and the space used is the representation of the DNF formula by a trie, that is $O(mn)$. We apply Theorem 12 to this algorithm. We have a bound on the incremental delay, the space used and the number of solutions, hence we can use Theorem 3 to do the geometric amortization without overhead in the method of Theorem 12. The auxiliary queue used in Theorem 12 is of size n^2m, since the path time is nm. The number of models is bounded by 2^n, hence the delay obtained by amortization is $O(n^2m^{1-\log_3(2)})$ and the space $O(n^2m)$, which proves the theorem. ◀

For monotone DNF formulas, Theorem 14 of [12] gives a flashlight search with an average delay of $O(\log(mn))$. Hence, we obtain an algorithm with delay $O(n\log(mn))$ listing the models of monotone DNF formulas with strong polynomial delay by Theorem 12. It gives an algorithm having a better delay, preprocessing and space usage than the algorithm given by Theorem 12 of [12].

▶ **Corollary 14.** *There is an algorithm solving Π_{DNF} on monotone formulas with polynomial space, linear preprocessing and strong polynomial delay.*

We have not proven that $\Pi_{DNF} \in$ SDelayP, and the DNF Enumeration Conjecture, which states that $\Pi_{DNF} \notin$ SDelayP still seems credible. Theorem 3 shows that this conjecture can be restated in terms of incremental delay, suggesting that the conjectured hardness should rely on the incremental delay and not on the delay.

▶ **Conjecture 15.** *There is no polynomial p such that Π_{DNF} can be solved with polynomial space and incremental delay $p(n)$.*

References

1 Alfred V Aho and John E Hopcroft. *The design and analysis of computer algorithms.* Pearson Education India, 1974.

2 Ricardo Andrade, Martin Wannagat, Cecilia C Klein, Vicente Acuña, Alberto Marchetti-Spaccamela, Paulo V Milreu, Leen Stougie, and Marie-France Sagot. Enumeration of minimal stoichiometric precursor sets in metabolic networks. *Algorithms for Molecular Biology*, 11(1):25, 2016.

3 David Avis and Komei Fukuda. Reverse search for enumeration. *Discrete Applied Mathematics*, 65(1-3):21–46, 1996.

4 Guillaume Bagan. *Algorithms and complexity of enumeration problems for the evaluation of logical queries.* PhD thesis, Université de Caen, France, 2009.

5 JoséL Balcázar. Self-reducibility. *Journal of Computer and System Sciences*, 41(3):367–388, 1990.

6 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.

7 James R Bitner, Gideon Ehrlich, and Edward M Reingold. Efficient generation of the binary reflected gray code and its applications. *Communications of the ACM*, 19(9):517–521, 1976.

8 Thomas Bläsius, Tobias Friedrich, Julius Lischeid, Kitty Meeks, and Martin Schirneck. Efficiently enumerating hitting sets of hypergraphs arising in data profiling. In *2019 Proceedings of the Twenty-First Workshop on Algorithm Engineering and Experiments (ALENEX)*, pages 130–143. SIAM, 2019.

9 Kateřina Böhmová, Luca Häfliger, Matúš Mihalák, Tobias Pröger, Gustavo Sacomoto, and Marie-France Sagot. Computing and listing st-paths in public transportation networks. *Theory of Computing Systems*, 62(3):600–621, 2018.

10 Caroline Brosse, Vincent Limouzy, and Arnaud Mary. Polynomial delay algorithm for minimal chordal completions. In Mikolaj Bojanczyk, Emanuela Merelli, and David P. Woodruff, editors, *49th International Colloquium on Automata, Languages, and Programming, ICALP 2022, July 4-8, 2022, Paris, France*, volume 229 of *LIPIcs*, pages 33:1–33:16. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2022. doi:10.4230/LIPIcs.ICALP.2022.33.

11 Florent Capelli and Yann Strozecki. Incremental delay enumeration: Space and time. *Discrete Applied Mathematics*, 268:179–190, 2019.

12 Florent Capelli and Yann Strozecki. Enumerating models of DNF faster: Breaking the dependency on the formula size. *Discrete Applied Mathematics*, 303:203–215, 2021.

13 Florent Capelli and Yann Strozecki. Geometric amortization of enumeration algorithms. *arXiv preprint arXiv:2108.10208*, 2021.

14 Nofar Carmeli and Markus Kröll. On the enumeration complexity of unions of conjunctive queries. In *Proceedings of the 38th ACM SIGMOD-SIGACT-SIGAI Symposium on Principles of Database Systems*, pages 134–148, 2019.

15 Sara Cohen, Benny Kimelfeld, and Yehoshua Sagiv. Generating all maximal induced subgraphs for hereditary and connected-hereditary graph properties. *Journal of Computer and System Sciences*, 74(7):1147–1159, 2008.

16 Alessio Conte, Roberto Grossi, Andrea Marino, and Luca Versari. Listing maximal subgraphs satisfying strongly accessible properties. *SIAM Journal on Discrete Mathematics*, 33(2):587–613, 2019.

17 Alessio Conte and Takeaki Uno. New polynomial delay bounds for maximal subgraph enumeration by proximity search. In *Proceedings of the 51st Annual ACM SIGACT Symposium on Theory of Computing*, pages 1179–1190, 2019.

18 Stephen A Cook and Robert A Reckhow. Time bounded random access machines. *Journal of Computer and System Sciences*, 7(4):354–375, 1973.

19 Arnaud Durand and Etienne Grandjean. First-order queries on structures of bounded degree are computable with constant delay. *ACM Trans. Comput. Log.*, 8(4):21, 2007.

20 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.

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

22 Leslie Ann Goldberg. *Efficient algorithms for listing combinatorial structures*. PhD thesis, University of Edinburgh, UK, 1991. URL: http://hdl.handle.net/1842/10917.

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

24 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.

25 Samir Khuller and Vijay V Vazirani. Planar graph coloring is not self-reducible, assuming P$\neq$ NP. *Theoretical Computer Science*, 88(1):183–189, 1991.

26 Donald E Knuth. Combinatorial algorithms, part 1, volume 4a of the art of computer programming, 2011.

27 Eugene L Lawler, Jan Karel Lenstra, and AHG Rinnooy Kan. Generating all maximal independent sets: NP-hardness and polynomial-time algorithms. *SIAM Journal on Computing*, 9(3):558–565, 1980.

28 Édouard Lucas. *Récréations mathématiques: Les traversées. Les ponts. Les labyrinthes. Les reines. Le solitaire. La numération. Le baguenaudier. Le taquin*, volume 1. Gauthier-Villars et fils, 1882.

29 Arnaud Mary and Yann Strozecki. Efficient enumeration of solutions produced by closure operations. *Discrete Mathematics & Theoretical Computer Science*, 21(3), 2019.

30 Kurt Mehlhorn. *Data structures and algorithms 1: Sorting and searching*, volume 1. Springer Science & Business Media, 2013.

31 Keisuke Murakami and Takeaki Uno. Efficient algorithms for dualizing large-scale hypergraphs. *Discrete Applied Mathematics*, 170:83–94, 2014.

32 Ronald C Read and Robert E Tarjan. Bounds on backtrack algorithms for listing cycles, paths, and spanning trees. *Networks*, 5(3):237–252, 1975.

33 Frank Ruskey. Combinatorial generation. *Preliminary working draft. University of Victoria, Victoria, BC, Canada*, 11:20, 2003.

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

35 Yann Strozecki. Enumeration complexity. *Bulletin of EATCS*, 1(129), 2019.

36 James C Tiernan. An efficient search algorithm to find the elementary circuits of a graph. *Communications of the ACM*, 13(12):722–726, 1970.

37 Nikolaos Tziavelis, Wolfgang Gatterbauer, and Mirek Riedewald. Any-k algorithms for enumerating ranked answers to conjunctive queries, 2022. doi:10.48550/arXiv.2205.05649.

38 Takeaki Uno. Constant time enumeration by amortization. In *Workshop on Algorithms and Data Structures*, pages 593–605. Springer, 2015.

39 Kunihiro Wasa and Kazuhiro Kurita. Enumeration of enumeration algorithms and its complexity. https://kunihirowasa.github.io/enum/problem_list. Accessed: 2021-10-31.

A Oracles to RAM

In this Appendix, the size of the input of the algorithm is denoted by n. We assume in this section that the polynomial $p(n)$ is the *known* delay of I, the simulated RAM. The complexity of any operation in the RAM model, say $a+b$ is $(\log(a)+\log(b))/\log(n)$. If a and b are bounded by some polynomial in n, then $(\log(a)+\log(b))/\log(n) < C$ for some constant C. All integers used in this section are bounded by a polynomial in n and can thus be manipulated in constant time and stored using constant space. We assume an infinite supply of *zero-initialized memory*, that is all registers of the machines we use are first initialized to zero. It is not a restrictive assumption, since we can relax it, by using a lazy initialization method (see [30] 2, Section III.8.1) for all registers, for only a constant time and space overhead for all memory accesses.

A.1 Pointers and Memory

To implement extensible data structures, we need to use pointers. A pointer is an integer, stored in some register, which denotes the index of the register from which is stored an element. In this article, the value of a pointer is always bounded by a polynomial in n, thus it requires constant memory to be stored. Using pointers, it is easy to implement linked lists, each element contains a pointer to its value and a pointer to the next element of the list. Following a pointer in a list can be done in constant time. Adding an element at the end of a list can be done in constant time if we maintain a pointer to the last element. We also use arrays, which are a set of consecutive registers of known size.

In our algorithms, we may need memory to extend a data structure or to create a new one, but we never need to free the memory. Such a memory allocator is trivial to implement: we maintain a register containing the value F, such that no register of index larger than F is used. When we need k consecutive free registers to extend a data structure, we use the registers from F to $F+k-1$ and we update F to $F+k$.

A.2 Counters

All algorithms presented in this paper rely, sometimes implicitly, on our ability to efficiently maintain counters, for example, to keep track of the number of steps of a RAM that have been simulated so far. Implementing them naively by simply incrementing a register would result in efficiency loss since these registers may end up containing values as large as $2^{\text{poly}(n)}$ and we could not assume that this register can be incremented, compared, or multiplied in constant time in the uniform cost model that we use in this paper.

To circumvent this difficulty, we introduce in this section a data structure that allows us to work in constant time with counters representing large values. Of course, we will not be able to perform any arithmetic operations on these counters. However, we show that our counter data structure enjoys the following operations in constant time: $\mathsf{inc}(c)$ increases the counter by 1 and $\mathsf{mbit}(c)$ returns the index of the most significant bit of the value encoded by c. In other words, if $k = \mathsf{mbit}(c)$ then we know that $\mathsf{inc}(c)$ has been executed at least 2^k times and at most 2^{k+1} times since the initialization of the counter.

The data structure is based on Gray code encoding of numbers. A Gray code is an encoding enjoying two important properties: the Hamming distance of two consecutive elements in the Gray enumeration order is one and one can produce the next element in the order in constant time. The method we present in this section is inspired by Algorithm G presented in [26] which itself is inspired by [7] for the complexity. The only difference with Algorithm G is that we maintain a stack containing the positions of the 1-bits of the code in increasing order so that we can retrieve the next bit to switch in constant time which is not obvious in Algorithm G. Our approach is closer to the one presented in Algorithm L of [26] but for technical reasons, we could not use it straightforwardly.

We assume in the following that we have a data structure for a stack supporting initialization, push and pop operations in constant time and using $O(s)$ registers in memory where s is the size of the stack (it can be implemented by a linked list).

Counters with a known upper bound on the maximal value. We start by presenting the data structure when an upper bound on the number of bits needed to encode the maximal value to be stored in the counter is known. For now on, we assume that the counter will be incremented at most $2^k - 1$ times, that is, we can encode the maximal value of the counter using k bits.

To initialize the data structure, we simply allocate k consecutive registers $R_0, \dots, R_{k-1}$ initialized to 0, which can be done in constant time since the memory is assumed to be initialized to 0, and we initialize an empty stack S. Moreover, we have two other registers A and M initialized to 0.

We will implement mbit and inc to ensure the following invariants: the bits of the Gray Code encoding the value of the counter are stored in $R_0, \dots, R_{k-1}$. A contains the parity of the number of 1 in $R_0, \dots, R_{k-1}$. M contains an integer smaller than k that is the position of the most significant bit in the Gray Code (the biggest $j \leq k-1$ such that R_j contains 1). Finally, S contains all positions j such that R_j is set to 1 in decreasing order (that is if $j < j'$ are both in S, j will be poped before j').

To implement mbit, we simply return the value of M. It is well-known and can be easily shown that the most significant bit of the Gray Code is the same as the most significant bit of the value it represents in binary so if the invariant is maintained, M indeed contains a value j such that the number of times $\mathsf{inc}(c)$ has been executed is between 2^j and $2^{j+1} - 1$.

To implement inc, we simply follow Algorithm G from [26]. If A is 0 then we swap the value of R_0. Otherwise, we swap the value of R_{j+1} where j is the smallest position such that $R_j = 1$ (if j is $k-1$ then we have reached the maximal value of the code which we have assumed to be impossible, see below to handle unbounded counters). One can find j in constant time by just popping the first value in S, which works if the invariant is maintained. Now, one has to update the auxiliary memory: A is replaced by $1 - A$ so that it still represents the parity of the number of 1 in the Gray Code. To update S, we proceed as follows: if A is 0 then either R_0 has gone from 0 to 1, in which case we have to push 0 in S or R_0 has gone from 1 to 0, in which case we have to pop one value in S, which will be 0 since S respects the invariant. It can be readily proven that this transformation preserves the invariant on S. Now, if A is 1, then either the value of R_{j+1} has gone from 0 to 1 which means that we have to push $j+1$ and j on the stack (j is still the first bit containing 1 so it has to be pushed back on the top of the stack and $j+1$ is the next bit set to 1 so it has to be just after j in S). Or the value of R_{j+1} has gone from 1 to 0. In this case, it means that after having popped j from S, $j+1$ sits at the top of S. Since R_{j+1} is not 0, we have to pop $j+1$ from S and push back j. Again, it is easy to see that these transformations preserve the invariant on S. Moreover, we never do more than 2 operations on the stack so this can be done in constant time. Finally, if R_{j+1} becomes 1 and $j+1 > M$, we set M to $j+1$.

Observe that we are using $2k+2$ registers for this data structure since the stack will never hold more than k values.

Unbounded counters. To handle unbounded counters, we start by initializing a bounded counter c_0 with k bits (k can be chosen arbitrarily, $k = 1$ works). When c_0 reaches its maximal value, we just initialize a new counter c_1 with $k+1$ bits and modify it so it contains the Gray Code of c_0 (with one extra bit) and copy its stack S and the values of A and M.

This can be done in constant time thanks to the following property of Gray code: the Gray code encoding of $2^k - 1$ contains exactly one bit set to 1 at position $k-1$. Thus, to copy the value of c_0, we only have to swap one bit in c_1 (which has been initialized to 0 in constant time). Moreover, the stack of c_0 containing only positions of bit set to 1, it contains at this point only the value $k-1$ that we can push into the stack of c_1. Copying registers A and M is obviously in constant time.

To summarize, we have proven the following:

▶ **Theorem 16.** *There is a data structure* Counter *that can be initialized in constant time and for which operations* inc *and* mbit *can be implemented in constant time with the following semantic:* mbit(c) *returns an integer j such that v is between 2^j and $2^{j+1} - 1$ where v is the number of time* inc(c) *has been executed since the initialization of c. Moreover, the data structure uses $O(\log(v)^2)$ register.*

One could make the data structure more efficient in memory by lazily freeing the memory used by the previous counters so that it is $O(\log(v))$. However, such an optimization is not necessary for our purpose.

A.3 Instructions load, move and steps for Known Parameters

In this section, we explain formally how one can simulate a given RAM as an oracle with good time and memory guarantees. More precisely, we explain how one can implement the instructions load, move and steps that we are using in our algorithms so that their complexity is $O(1)$ and their memory usage is $O(s(n))$ where $s(n)$ is the memory used by I the simulated RAM on an input of size n. We do the following assumptions: we know an upper bound for both values $s(n)$ and $\lceil \log(\sharp I(x)) \rceil$. We also assume that $s(n)$ is bounded by a polynomial. Note that $\lceil \log(\sharp I(x)) \rceil$ is polynomial in n, since we consider only machines solving problems in EnumP.

Configuration. Instruction load(I, x) returns a structure M which stores the **configuration** of I when its runs on input x. A configuration of I is the content of the registers up to the last one which has been accessed and the state of the machine, i.e. the index of the next instruction to be executed by I. Moreover, the number of executed move(M) instructions is also part of the configuration to support the steps instruction.

Remark that we make explicit that machine I is simulated, by giving it as argument of load. However, the amortization algorithms we design all use load only on the machine I. They must be understood as a method to build an amortized algorithm for each I. Therefore, we do not need a universal machine to simulate I when executing a move(M) instruction.

To simulate I in constant time, the crucial part is to be able to read and write the ith register of I as stored in M in constant time. If we know a bound $s(n)$ on the space used by I, and a bound on the number of solutions $\sharp I(x)$ as in Algorithm 1, the structure M is very simple. For a structure M, we reserve $s(n)$ registers which are mapped one to one to the registers R_1 up to $R_{s(n)}$ of I. We also require 1 register to store the index of the current instruction to be executed by I. We also initialize a counter c to 0 as explained in Section A.2 for steps(M) to keep track of the number of steps that have been simulated so far. This counter will use up to $O(\log(\sharp I(x))^2)$ registers. To really account for steps(M), one should increment c each time an instruction move is executed. However, in Algorithm 1 and Algorithm 2, one need to compare steps(M) with another value. We explain below how one can adapt this counter so that this comparison is constant time for both algorithms.

Let $m = s(n) + 2\lceil\log(\sharp I(x))\rceil + 2$, then for all j from 0 to $\lceil\log(\sharp I(x))\rceil + 1$, the structure $M[j]$ uses the registers from jm to $(j+1)m - 1$. Hence, if $M[j]$ must simulate the access of I to register R_i, it accesses the register R_{jm+i}. This operation is in constant time, since it requires to compute $jm + i$, where i, m and j are polynomial in n.

At Line 8 of Algorithm 1, one has to determine whether the number of steps simulated is in $[2^{j-1}p(n) + 1, 2^j p(n)]$. To check this inequality in constant time, we simply initialize a counter c_j as in Section A.2. Instead of incrementing it each time move$(M[j])$ is called, we increment it every $p(n)$ calls to move. This can easily be done by keeping another register R which is incremented each time move is called and whenever it reaches value $p(n)$, it is reset to 0 and c_j is incremented. Now to decide whether $M[j]$ enters its zone, it is sufficient to test whether mbit$(c_j) = j - 1$. The first time it happens, then exactly $2^{j-1}p(n)$ steps of $M[j]$ have been executed, so it will enter its zone in the next move, so we can remember it to start the enumeration. When mbit(c_j) becomes j, it means that $2^j p(n)$ steps of $M[j]$ have been executed, that is, $M[j]$ leaves its zone. Thus, we can perform the check of Line 8 in constant time.

A.4 Instruction copy

Algorithm 2, which does not require to know $\#I(x)$, relies on an instruction copy. This instruction takes as a parameter a data structure M storing the configuration of a RAM and returns a new data structure M' of the same machine starting in the same configuration (an exact copy of the memory). A straightforward way of implementing copy would be to copy every register used by the data structure M in a fresh part of the memory. However, this approach may be too expensive since we need to copy the whole memory used by M. Since we are guaranteed to have one output solution between each copy instruction, the delay of Algorithm 1 becomes $O(\log(\#I(x))(p(n) + s(n)))$.

In this section, we explain how one can lazily implement this functionality so that the memory of M is copied only when needed. This method ensures that copy runs in $O(1)$, however, there is an overhead to the cost of the instruction move. We show it still runs in $O(1)$ if the memory usage of I is well behaved, otherwise the overhead is small and exists only when $\log(\#I(x)) \leq \log(n)^2$.

Let us explain how the data structure M is lazily copied. The data structure contains a register for the index of the current instruction, a counter of the number of steps and an array to represent the registers of I the simulated machine. The counter in M' is stored as in Theorem 16. It is initialized so that it represents the value $2^{j-1}p(n)$ and it counts up to $2^{j+1}p(n)$. This value is represented by a regular counter of value 0 and the Gray code counter contains the $2^{j-1}p(n)$th integer in Gray code order for integers of size $j+1$. This number is equal to $2^{j-1}+2^{j-2}$, which has only two one bits (the second and the third), hence it can be set up in constant time. The auxiliary structure is the list of ones of the integer, which is here of size two and can thus be set up in constant time.

We explain how we lazily copy an array. Assume we want to create an exact copy of the array A of size m. We create both A' and U of size m initialized to zero. The value of $U[r]$ is 0 if $A'[r]$ has not been copied from $A[r]$ and 1 otherwise. Each time $\mathsf{move}(M)$ is executed and it modifies the value $A[r]$, if $U[r]=0$, it first set $A'[r]=A[r]$ and $U[r]=1$. Each time $\mathsf{move}(M')$ is executed and reads the value $A'[r]$, if $U[r]=0$, it first set $A'[r]=A[r]$ and $U[r]=1$. This guarantees that the value of A' is always the same as if we had completely copied it from A when the instruction $\mathsf{copy}(M)$ is executed. The additional checks and updates of U add a constant time overhead to move. Moreover, we maintain a simple counter c, and each time a $\mathsf{move}(M')$ operation is executed, if $U[c]=0$, we set $A'[c]=A[c]$ and $U[c]=1$. When $c=m$, the copy is finished and we can use A and A' as before, without checking U.

The described implementation of the copy operation is in constant time. The move instruction, modified as described, has a constant overhead for each lazy copy mechanism in action. To evaluate the complexity of Algorithm 2, we must evaluate the number of active copies. We prove, that when $s(n)$ is known, a variant of Algorithm 2 has only a single active copy mechanism at any point in time.

▶ **Theorem 17.** *Given an* IncP_1*-enumerator I, its incremental delay $p(n)$ and its space complexity $s(n)$, one can construct a* DelayP*-enumerator I' which enumerates $I(x)$ on input $x \in \Sigma^*$ with delay*

$$O(\log(\#I(x))p(n))$$

and space complexity $O(s(n)\log(\#I(x)))$.

Proof. We use a hybrid version of Algorithm 1 and Algorithm 2. First, in the preprocessing step, I is run for $s(n)$ steps. If the computation terminates before $s(n)$ steps, then we store all solutions during the preprocessing and enumerate them afterward.

Otherwise, let i be the integer such that $2^{i-1}p(n) \leq s(n) < 2^{i}p(n)$. We run Algorithm 1 with $\log(s(n)/p(n))$ as a bound on the number of solutions. It means that $M[0]$ up to $M[i]$ are loaded in the preprocessing. Since the number of solutions can be larger than $\log(s(n)/p(n))$, we need machines $M[j]$ for $j > i$. These machines are created dynamically as in Algorithm 2. When a machine $M[j]$ is created, it is lazily copied from $M[j-1]$ using copy. There are at least $2^{j-1}p(n) \geq 2^{i}p(n) \geq s(n)$ instructions executed before the next copy instruction. Therefore, a single lazy copy is active at any point of the algorithm, which proves that the delay is $O(\log(\#I(x))p(n))$. ◀