



HAL
open science

Geometric Amortization of Enumeration Algorithms

Florent Capelli, Yann Strozecki

► **To cite this version:**

Florent Capelli, Yann Strozecki. Geometric Amortization of Enumeration Algorithms. 40th International Symposium on Theoretical Aspects of Computer Science (STACS 2023), Mar 2023, Hamburg, Germany. hal-03955911

HAL Id: hal-03955911

<https://hal.science/hal-03955911>

Submitted on 25 Jan 2023


HAL is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L'archive ouverte pluridisciplinaire **HAL**, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d'enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.

1 Geometric Amortization of Enumeration 2 Algorithms

3 Florent Capelli ✉ 

4 Université de Lille, CNRS, Inria, Centrale Lille, UMR 9189 - CRISTAL, F-59000 Lille, France

5 Yann Strozecki ✉ 

6 Université Paris Saclay, UVSQ, DAVID, France

7 — Abstract —

8 In this paper, we introduce a technique we call geometric amortization for enumeration algorithms,
9 which can be used to make the delay of enumeration algorithms more regular with little overhead on
10 the space it uses. More precisely, we consider enumeration algorithms having incremental linear delay,
11 that is, algorithms enumerating, on input x , a set $A(x)$ such that for every $t \leq \#A(x)$, it outputs
12 at least t solutions in time $O(t \cdot p(|x|))$, where p is a polynomial. We call p the incremental delay
13 of the algorithm. While it is folklore that one can transform such an algorithm into an algorithm
14 with maximal delay $O(p(|x|))$, the naive transformation may use exponential space. We show that,
15 using geometric amortization, such an algorithm can be transformed into an algorithm with delay
16 $O(p(|x|) \log(\#A(x)))$ and space $O(s \log(\#A(x)))$ where s is the space used by the original algorithm.
17 In terms of complexity, we prove that classes DelayP and IncP₁ with polynomial space coincide.

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

21 **2012 ACM Subject Classification** Theory of computation → Complexity classes

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

23 **Digital Object Identifier** 10.4230/LIPIcs.STACS.2023.20

24 **Related Version** <https://arxiv.org/abs/2108.10208>



© Florent Capelli and Yann Strozecki;
licensed under Creative Commons License CC-BY 4.0

40th International Symposium on Theoretical Aspects of Computer Science (STACS 2023).

Editors: Petra Berenbrink, Mamadou Moustapha Kanté, Patricia Bouyer, and Anuj Dawar; Article No. 20;
pp. 20:1–20:23



Leibniz International Proceedings in Informatics
LIPICs Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

25 **1** Introduction

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

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

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

62 While the delay is a natural way of measuring the quality of an enumeration algorithm, it
 63 might sometimes be too strong of a restriction. Indeed, if the enumeration algorithm is used
 64 to generate a subset of the solutions, it is often enough to have guarantees that the time
 65 needed to generate i solutions is reasonable for every i . For example, one could be satisfied
 66 with an algorithm that has the property that after a time $i \cdot p(n)$, it has output at least i
 67 solutions, where p is a polynomial and n the input size. In this paper, we refer to this kind
 68 of algorithm as IncP₁-enumerators¹ and call $p(n)$ *the incremental delay* of the algorithm.

69 While polynomial delay enumerators are IncP₁-enumerator, the converse is not true.

¹ The 1 in IncP₁ stands for the linear dependency of the incremental time in the number of solutions.

70 Indeed, IncP₁-enumerators do not have to output their solutions regularly. Take for example
 71 an algorithm that, on an input of size n , outputs 2^n solutions in 2^n steps, then nothing for
 72 2^n steps and finally outputs the last solution. It can be readily verified that this algorithm
 73 outputs at least i solutions after $2i$ steps for every $i \leq 2^n + 1$, and it is thus an IncP₁-
 74 enumerator. However, the delay of such an algorithm is not polynomial as the time spent
 75 between the output of the last two solutions is 2^n . Instead of executing the output instruction
 76 of this algorithm, one could store the solutions that are found in a queue. Then, every two
 77 steps of the original algorithm, one solution is removed from the queue and output. The
 78 IncP₁ property ensures that the queue is never empty when dequeued and we now have
 79 polynomial delay. Intuitively, the solutions being dense in the beginning, they are used to
 80 pave the gap until the last solution is found. While this strategy may always be applied
 81 to turn an IncP₁-enumerator into a polynomial delay algorithm, the size of the queue may
 82 become exponential in the size of the input. In the above example, after the simulation of 2^n
 83 steps, 2^n solutions have been pushed into the queue but only 2^{n-1} of them are output, so
 84 the queue still contains 2^{n-1} solutions. Unfortunately, an algorithm using exponential space
 85 may not be feasible. Therefore, much effort has been devoted to ensure that polynomial
 86 delay methods run with polynomial space [27, 3, 15, 17, 10].

87 **Contributions.** The main result of this paper is a proof that the regularization of an IncP₁-
 88 enumerator may be done without exponentially increasing the space used. More formally, we
 89 show that the class DelayP^{poly} of problems that can be solved by a polynomial delay and
 90 polynomial space algorithm is the same as the class IncP₁^{poly} of problems that can be solved
 91 by a polynomial space IncP₁-enumerator. In other words, we prove DelayP^{poly} = IncP₁^{poly},
 92 answering positively a question we raised in [11] and where only special cases were proven.
 93 Our result relies on a constructive method that we call *geometric amortization*. It turns any
 94 IncP₁-enumerator into a polynomial delay algorithm whose delay and space complexity are
 95 the incremental delay and space complexity of the original enumerator multiplied by a factor
 96 of $\log(S)$, where S is the number of solutions (Theorem 3). Interestingly, we also show that
 97 the total time can be asymptotically preserved.

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

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

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

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

135 **Organization of the paper.** In Section 2, we introduce enumeration problems, the related
 136 computational model and complexity measures. Section 3 presents different techniques to
 137 turn an IncP_1 -enumerator into a DelayP-enumerator using a technique called geometric
 138 amortization. Interactive visualization of how geometric amortization works can be found at
 139 <http://florent.capelli.me/coussinet/>. In Section 3.3 we apply geometric amortization
 140 to incremental polynomial algorithms, showing that our result generalizes to the IncP_i
 141 hierarchy. Section 4 gives a method to transform many DelayP-enumerators of average delay
 142 $a(n)$ into a DelayP-enumerator with maximal delay $a(n) \log(K)$ where K is the number of
 143 solutions. We use it to obtain an algorithm for the problem of enumerating the models
 144 of a DNF formula. To outline the main ideas of our algorithms, they are presented using
 145 pseudocode with instructions to simulate a given Random Access Machine (RAM). The details
 146 on the complexity of using such instructions with minimal overhead are given in the appendix.

147 2 Preliminaries

148 **Enumeration problems.** Let Σ be a finite alphabet and Σ^* be the set of finite words built
 149 on Σ . We denote by $|x|$ the length of $x \in \Sigma^*$. Let $A \subseteq \Sigma^* \times \Sigma^*$ be a binary predicat. We
 150 write $A(x)$ for the set of y such that $A(x, y)$ holds. The enumeration problem Π_A is the
 151 function which associates $A(x)$ to x . The element x is called the *instance* or the *input*, while
 152 an element of $A(x)$ is called a *solution*. We denote the cardinality of a set $A(x)$ by $\#A(x)$.

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

158 **Computational model.** In this paper, we use the Random Access Machine (RAM) model
 159 introduced by Cook and Reckhow [18] with comparison, addition, subtraction and multipli-

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

171 A RAM M on input $x \in \Sigma^*$ produces a sequence of outputs y_1, \dots, y_s . The set of outputs
 172 of M is denoted by $M(x)$ and its cardinality by $\#M(x)$. We say that M solves Π_A if, on
 173 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*
 174 *repeated*. All registers are initialized with zero. The space used by the M is the sum of the
 175 bitsize of the integers stored in its registers, up to the register of the largest index accessed.

176 We denote by $T_M(x, i)$ the time taken by the machine M on input x before the i^{th}
 177 `OUTPUT` instruction is executed. When the machine is clear from the context, we drop the
 178 subscript M and write $T(x, i)$. The *delay* of a RAM which outputs the sequence y_1, \dots, y_s
 179 is the maximum over all $i \leq s$ of the time spent between the generation of y_i and y_{i+1} , that
 180 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
 181 first solution is output. The *postprocessing* is the time spent between the output of the last
 182 solution and the end of the computation. To simplify the presentation, we assume that there
 183 is no postprocessing, that is, a RAM solving an enumeration problem stops right after having
 184 output the last solution. This assumption does not affect the complexity classes studied in
 185 this paper, as the output of the last solution can be delayed to the end of the algorithm.

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

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

- 200 ■ `sol(M)` returns the solution that M has just output. If the last simulated step of M was
 201 not an output instruction, it returns `undef`. We abuse notation by writing `if(sol(M))`
 202 `then ...` to express the fact that we explore the `then` branch if and only if `sol(M)` is not
 203 `undef`.
 - 204 ■ `steps(M)` returns the number of steps of M that have been simulated.
- 205 If we have an upper bound $u(|x|)$ on the memory used by a machine of code I on input x , and
 206 if u is computable in time $t(|x|)$, we can implement `load` and `move` on a RAM with respective

207 complexity $O(t(|x|))$ and $O(1)$ and space $O(u(|x|))$. Indeed, it is sufficient to reserve $u(|x|)$
 208 contiguous registers in memory and shift all registers used by I so that it uses the reserved
 209 memory.

210 It is also possible to implement these instructions without knowing in advance the
 211 memory used by I but one has to use data structures able to dynamically adjust the memory
 212 used. In this case, `move` can be executed either in $O(1)$ with a small space overhead or in
 213 $O(\log(\log(|x|)))$ with no space overhead. We leave this improvement for a longer version of
 214 the paper (see [13]) and state the main results when a polynomial time computable upper
 215 bound $u(|x|)$ on the memory is known.

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

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

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

235 Observe that if M is a DelayP -enumerator then for all i we have $T(x, i) - T(x, 1) \leq$
 236 $\sum_{1 < j \leq i} d(|x|) = (i - 1)d(|x|)$. Hence $\text{DelayP} \subseteq \text{IncP}_1$. Polynomial delay is the most common
 237 notion of tractability in enumeration, because it guarantees both regularity and linear total
 238 time and also because it is relatively easy to prove that an algorithm has a polynomial
 239 delay. Indeed, most methods used to design enumeration algorithms such as backtrack search
 240 with a polynomial time extension problem [29], or efficient traversal of a supergraph of
 241 solutions [27, 3, 16], yield polynomial delay algorithms on many enumeration problems.

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

246 **3 From IncP_1 to DelayP**

247 **3.1 Geometric Amortization**

248 The folklore method (e.g., [23, 34, 14]) used to transform an IncP_1 -enumerator into a DelayP -
 249 enumerator that was sketched in the introduction uses a queue to delay the output of solutions.

250 This queue may however become of size exponential in the input size. To overcome this issue,
 251 we introduce a technique that we call *geometric amortization*, illustrated by Algorithm 1
 252 which regularizes the delay of an IncP_1 -enumerator with a space overhead of $O(\log(\#I(x)))$,
 253 which is polynomially bounded since I is in EnumP. To achieve this, we however have
 254 to compromise a bit on the delay which becomes $O((\log(\#I(x)) \cdot p(|x|)))$. Moreover, with
 255 geometric amortization, the solutions are not output in the same order as the order they
 256 are output by I . Algorithm 1 relies on the knowledge of an upper bound K of $\#I(x)$, but
 257 this assumption is relaxed in Section 3.2. We now proceed to prove the correctness and
 258 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 $K \geq \#I(x)$
Output : Enumerate $I(x)$ with delay $O(p(n) \cdot \log(K))$

```

1 begin
2    $N \leftarrow \lceil \log(K) \rceil$ ;
3   for  $i = 0$  to  $N$  do  $M[i] \leftarrow \text{load}(I, x)$  ;
4    $j \leftarrow N$ ;
5   while  $j \geq 0$  do
6     for  $b \leftarrow 2p(n)$  to 0 do
7       move( $M[j]$ );
8       if  $\text{sol}(M[j])$  and  $\text{steps}(M[j]) \in Z_j$  then
9         output( $\text{sol}(M[j])$ );
10         $j \leftarrow N$ ;
11        break;
12    if  $b = 0$  then  $j \leftarrow j - 1$ ;

```

259 ► **Theorem 3.** Given an IncP_1 -enumerator I with incremental delay $p(n)$ and space complex-
 260 ity $s(n)$ and given $K \geq \#I(x)$, one can construct a DelayP-enumerator I' which enumerates
 261 $I(x)$ on any input $x \in \Sigma^*$ with delay $O(\log(K)p(n))$ and space complexity $O(s(n)\log(K))$.

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

264 Since $\text{IncP}_1 \subseteq \text{EnumP}$, we know that there is a polynomial $q(n)$ such that every element of
 265 $I(x)$ is of size at most $q(|x|)$ and by choosing $K = |\Sigma|^{q(n)}$, we have that $\log(K)$ is polynomially
 266 bounded and the following is a direct corollary of Theorem 3:

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

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

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

281 Algorithm 1 starts by moving $M[N]$. It is given a budget of $2p(n)$ steps. If these $2p(n)$
 282 steps are executed without finding a solution in Z_N , $M[N - 1]$ is then moved similarly with a
 283 budget of $2p(n)$ steps. It continues until one machine $M[j]$ finds a solution in its zone Z_j . In
 284 this case, the solution is output and the algorithm proceeds back with $M[N]$. The algorithm
 285 stops when $M[0]$ has left Z_0 , that is when $p(n) + 1$ steps of $M[0]$ have been simulated².

286 **Bounding the delay.** From the above description of Algorithm 1, between two outputs, the
 287 variable j takes at most $N + 1$ values (from N to 0) and at most $2p(n)$ move instructions
 288 are executed for each machine $M[j]$. A move instruction can be executed in $O(1)$ (see
 289 Appendix A). Moreover, the size of b being $O(\log(n))$, we can increment it in $O(1)$ in the
 290 RAM model we consider. Finally, we have to compare $\text{steps}(M[i])$ with integers of values
 291 in $O(\log(K)p(n))$. Manipulating such integers in the RAM model would normally cost
 292 $O(\log(K)/\log(n))$. However, we give in Appendix A.2 a method using Gray Code encodings
 293 which allows us to increment $\text{steps}(M[i])$ and to detect when it enters and exits Z_i in $O(1)$.
 294 Thus, the overall delay of Algorithm 1 is $O(\log(K)p(n))$.

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

298 **Correctness of Algorithm 1.** It remains to show that Algorithm 1 correctly outputs $I(x)$
 299 on input x . Recall that a solution of $I(x)$ is enumerated by $M[i]$ if it is produced by I at
 300 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
 301 at most $\#I(x)p(n)$, it is clear that $Z_0 \uplus \dots \uplus Z_N \supseteq [1, Kp(n)] \supseteq [1, \#(I)p(n)]$ covers every
 302 solution and that each solution is produced at most once. Thus, it remains to show that
 303 when the algorithm stops, $M[i]$ has moved by at least $2^i p(n)$ steps, that is, it has reached
 304 the end of Z_i and output all solutions in this zone.

305 We study an execution of Algorithm 1 on input x . For the purpose of the proof, we
 306 only need to look at the values of $\text{steps}(M[0]), \dots, \text{steps}(M[N])$ during the execution of the
 307 algorithm. We thus say that the algorithm is in state $c = (c_0, \dots, c_N)$ if $\text{steps}(M[i]) = c_i$ for
 308 all $0 \leq i \leq N$. We denote by S_i^c the set of solutions that have been output by $M[0], \dots, M[i]$
 309 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
 310 $k \in Z_j$ for $j \leq i$ and $k \leq c_j = \text{steps}(M[j])$. We claim the following invariant:

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

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

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.

314 Now assume the statement holds at state c' and let c be the next state. Let $i < N$. If
 315 $|S_i^c| = |S_i^{c'}|$, then the inequality still holds since $c_{i+1} \geq c'_{i+1}$ and $c'_{i+1} \geq 2p(n)|S_i^{c'}| = 2p(n)|S_i^c|$
 316 by induction. Otherwise, we have $|S_i^c| = |S_i^{c'}| + 1$, that is, some simulation $M[k]$ with $k \leq i$
 317 has just output a solution. In particular, the variable j has value $k \leq i < N$. Let c'' be
 318 the first state before c' such that variable j has value $i + 1$ and b has value $2p(n)$, that is,
 319 c'' is the state just before Algorithm 1 starts the for loop to move $M[i + 1]$ by $2p(n)$ steps.
 320 No solution has been output between state c'' and c' since otherwise j would have been
 321 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
 322 moved by $2p(n)$ steps in the for loop without finding a solution. By induction, we have
 323 $c''_{i+1} \geq 2p(n)|S_i^{c''}| = 2p(n)|S_i^{c'}|$. Thus $c_{i+1} \geq c''_{i+1} + 2p(n) = 2p(n)(|S_i^{c'}| + 1) = 2p(n)|S_i^c|$
 324 which concludes the induction. ◀

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

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

330 Now assume $c_j \geq 2^j p(n)$ for every $j < i$. This means that for every $j < i$, $M[j]$ has been
 331 moved at least to the end of Z_j . Thus, $M[j]$ has found every solution in Z_j . Since it holds
 332 for every $j < i$, it means that $M[0], \dots, M[i - 1]$ have found every solution in the interval
 333 $K = [1, 2^{i-1} p(n)]$. Since I is an IncP₁-enumerator with delay $p(n)$ and since $2^{i-1} \leq \#I(x)$ by
 334 definition of N , K contains at least 2^{i-1} solutions, that is, $|S_{i-1}^c| \geq 2^{i-1}$. Applying Lemma 5
 335 gives that $c_i \geq 2^{i-1} \cdot 2p(n) = 2^i p(n)$. ◀

336 The correctness of Algorithm 1 directly follows from Corollary 6. Indeed, it means that
 337 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
 338 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 .

339 3.2 Improving Algorithm 1

340 One drawback of Algorithm 1 is that it needs to know in advance an upper bound K on
 341 $\#I(x)$ since it uses it to determine how many simulations of I it has to maintain. In theory,
 342 such an upper bound exists because I is assumed to be in EnumP and it is often known, *e.g.*,
 343 $|\Sigma|^N$ where N is an upper bound on the size of the output. In practice, however, it might be
 344 cumbersome to compute it or it may hurt efficiency if the upper bound is overestimated. It
 345 turns out that one can remove this hypothesis by slightly modifying Algorithm 1. The key
 346 observation is that during the execution of the algorithm, if $M[i]$ has not entered Z_i , it is
 347 simulated in the same way as $M[i + 1], \dots, M[N]$. Indeed, it is not hard to see that $M[j]$ is
 348 always ahead of $M[i]$ for $j > i$ and that if $M[i]$ is not in Z_i , it will not output any solution
 349 in the loop at Line 6, hence this iteration of the loop will move $M[i]$ by $2p(n)$ steps, just like
 350 $M[j]$ for $j > i$. Hence, Algorithm 1 can be improved in the following way: we start with
 351 only two simulations $M[0], M[1]$ of I . Whenever $M[1]$ is about to enter Z_1 , we start $M[2]$ as
 352 an independent copy of $M[1]$. During the execution of the algorithm, we hence maintain a
 353 list M of simulations of I and each time the last simulation $M[N]$ is about to enter Z_N , we
 354 copy it into a new simulation $M[N + 1]$. The hardest part of implementing this idea is to
 355 show that one can copy simulation $M[N]$ without affecting the overall delay of the algorithm.
 356 That can be achieved by lazily copying parts of $M[N]$ whenever we move $M[N + 1]$. The
 357 details are given in Appendix A.4.

20:10 Geometric Amortization of Enumeration Algorithms

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

```

358 By implementing this idea, one does not need to know an upper bound on $\#I(x)$ anymore:
359 new simulations will be created as long as it is necessary to discover new solutions ahead.
360 The fact that one has found every solution is still witnessed by the fact that $M[0]$ reaches
361 the end of Z_0 . This improvement has yet another advantage compared to Algorithm 1: it
362 has roughly the same total time as the original algorithm. Hence, if one is interested in
363 generating every solution with a polynomial delay from an IncP₁-enumerator, our method
364 may make the maximal delay worse but does not change much the time needed to generate
365 all solutions.

366 **Correctness of Algorithm 2.** Correctness of Algorithm 2 can be proven in a similar way as
367 for Algorithm 1. Lemma 5 still holds for every state, where N in the statement has to be
368 replaced by $\text{length}(M) - 1$. The proof is exactly the same but we have to verify that when
369 a new simulation is inserted into M , the property still holds. Indeed, let c be a state that
370 follows the insertion of a new simulation (Line 8). We have now $\text{length}(M) - 1 = N + 1$
371 (thus the last index of M is $N + 1$). Moreover, we claim that $S_{N+1}^c = S_N^c$. Indeed, at this
372 point, the simulation $M[N + 1]$ has not output any solution. Moreover, by construction,
373 $c_N = \text{steps}(M[N]) = \text{steps}(M[N + 1]) = c_{N+1}$. Since $c_N \geq 2p(n)|S_N^c|$ by induction, we
374 have that $c_{N+1} \geq 2p(n)|S_{N+1}^c|$. Moreover, the following adaptation of Corollary 6 holds for
375 Algorithm 2.

376 ► **Lemma 7.** *Let c be the state reached when Algorithm 1 stops. Then $N := \text{length}(M) - 1 =$
377 $1 + \log(\#I(x))$ and for every $i \leq N$, $c_i \geq 2^i p(n)$.*

378 **Proof.** The lower bound $c_i \geq 2^i p(n)$ for $i \leq N$ is proven by induction exactly as in the proof
379 of Lemma 5. The induction holds as long as $2^{i-1} \leq \#I(x)$, because we need this assumption
380 to prove that there are at least 2^{i-1} solutions in the interval $[1, 2^{i-1}p(n)]$. Now, one can

381 easily see that if $i \leq 1 + \log(\#I(x))$ and $c_i \geq 2^i p(n)$ then the simulation $M[i]$ has reached
 382 $2^{i-1} p(n)$ at some point and thus, has created a new simulation $M[i+1]$. Thus, by induction,
 383 the algorithms creates at least $1 + \log(\#I(x)) = N$ new simulations. Thus $\text{length}(M) \geq N + 1$
 384 (as M starts with one element).

385 Finally, observe that $M[N]$ outputs solutions in the zone $Z_N = [2^{N-1} p(n) + 1, 2^N p(n)]$
 386 and that $2^{N-1} p(n) = \#I(x) p(n)$ which is an upper bound on the total time of I on input
 387 x . Thus, the simulation $M[N]$ will end without creating a new simulation. In other words,
 388 $\text{length}(M) - 1 = N$. ◀

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

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

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

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

415 ▶ **Theorem 8.** *Given an IncP_1 -enumerator I with incremental delay $p(n)$, space complexity*
 416 *$s(n)$ and total time $T(n)$, one can construct a DelayP-enumerator I' which enumerates $I(x)$*
 417 *on any input $x \in \Sigma^*$ with space complexity $O(s(n) \log(\#I(x)))$, delay $O(\log(\#I(x))p(n))$ and*
 418 *total time $O(T(n))$.*

419 We observe that Algorithm 2 can be modified so that it can work with IncP_1 -enumerators
 420 having a preprocessing. Indeed one only needs, as a preprocessing step of Algorithm 2, to
 421 run the first simulation created by the algorithm until it outputs its first solution to be in
 422 the same state as the case where there is no preprocessing.

423 We need to know two additional parameters (or an upper bound on them) to run
 424 Algorithm 1: the space of the amortized algorithm and its incremental delay. By using
 425 dynamic data structures, one could adapt our algorithm when the space used by the

426 enumerator is not known for a very small overhead. Moreover, it is possible to give a lower
 427 bound showing that one cannot get a $O(p(n))$ polynomial delay when the incremental delay
 428 $p(n)$ is unknown (if I is a blackbox). We leave this improvement for a longer version of this
 429 paper.

430 3.3 Geometric Amortization for IncP_i with $i > 1$

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

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

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

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

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

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

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

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

457 The inclusion $\text{IncP}_{a+1}^{\text{poly}} \subseteq \text{UsualIncP}_a^{\text{poly}}$ is done by geometric amortization, by adapting
 458 Algorithm 1. Let I be an algorithm solving a problem in IncP_{a+1} . We assume we know
 459 $t^{a+1}p(n)$, a bound on its incremental time.

460 Then, the only modification we do in Algorithm 1 is to maintain a counter S of the
 461 number of output solutions and modify the initialization of b in the for loop at line 6 to
 462 $S^a(a+1)2p(n)$. By construction of the amortization algorithm, the delay between two
 463 solutions before the algorithm ends is bounded by $S^a(a+1)2p(n)\log(s)$, where S is the
 464 number of solutions output up to this point of the algorithm and s the total number of
 465 solutions. Thus, the algorithm is in UsualIncP_a .

466 We still have to prove that all solutions are enumerated by the algorithm. Assume that
 467 the first $i+1$ machines $M[0], \dots, M[i]$ have output all the solutions in their zones, then we
 468 prove as in Corollary 6, that the machine $M[i+1]$ has also output all its solutions. The

469 number of solutions output by $M[0], \dots, M[i]$ is the number of solutions output by I up to
 470 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
 471 time $t^{a+1} p(n)$. Hence, $s_i \geq 2^{i/(a+1)}$.

472 When a solution is output by a machine $M[j]$ with $j \leq i$, then j is set to N and all
 473 machines $M[k]$ with $k > i$ move by at least $S^a(a+1)2p(n)$ steps where S is the current
 474 number of output solutions before $M[i]$ moves again. Hence, we can lower bound the number
 475 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
 476 $\sum_{S=0}^n S^a \geq \int_0^n S^a dS \geq n^{a+1}/(a+1)$, the number of moves of $M[i+1]$ is larger than $2^{i+1} p(n)$
 477 which is the upper bound of its zone. ◀

478 4 Other Applications of Geometric Amortization

479 4.1 Amortizing Self-Reducible Problems

480 Given an enumeration problem Π_A , we assume from now on, to lighten the exposition, that
 481 the solutions in $A(x)$ are sets over some universe $U(x)$. From A , we define the predicate \tilde{A}
 482 which contains the pairs $((x, a, b), y)$ such that $y \in A(x)$ and $a \subseteq y \subseteq b$. From this predicate,
 483 we define a self-reducible³ variant of Π_A and the extension problem $\text{EXTSOL}\cdot A$ defined as
 484 the set of triples (x, a, b) such that there is a y in $\tilde{A}(x, a, b)$.

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

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

- 495 ■ if $a = b$, a is output and the algorithm stops
- 496 ■ choose $u \in b \setminus a$;
- 497 ■ if $(x, a \cup \{u\}, b) \in \text{EXTSOL}\cdot A$, compute some auxiliary data D_1 from D and make a
 498 recursive call on $(x, a \cup \{u\}, b)$;
- 499 ■ if $(x, a, b \setminus \{u\}) \in \text{EXTSOL}\cdot A$, compute some auxiliary data D_2 from D_1 and make a
 500 recursive call on $(x, a, b \setminus \{u\})$, then compute D from D_2 .

501 Flashlight search can be seen as a depth-first traversal of a *partial solutions tree*. A
 502 node of this tree is a pair (a, b) such that $(x, a, b) \in \text{EXTSOL}\cdot A$. Node (a, b) has children
 503 $(a \cup \{u\}, b)$ and $(a, b \setminus \{u\})$ if they are nodes. A leaf is a pair (a, a) and the root is $(\emptyset, U(x))$.
 504 The *cost* of a node (a, b) is the time to execute the flashlight search on (x, a, b) *except the*
 505 *time spent in recursive calls*. Usually, the cost of a node comes from deciding $\text{EXTSOL}\cdot A$
 506 and modifying the global data structure D used to solve $\text{EXTSOL}\cdot A$ faster.

507 The cost of a path in a partial solution tree is the sum of the costs of the nodes in the
 508 path. We define the *path time* of a flashlight search algorithm as the maximum over the cost
 509 of all paths from the root. Twice the path time bounds the delay since, between two output
 510 solutions, a flashlight search traverses at most two paths in the tree of partial solutions.

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

511 To our knowledge, all bounds on the delay of flashlight search are proved by bounding the
 512 path time. The path time is bounded by $\sharp U(x)$ times the complexity of solving $\text{EXTSOL}\cdot A$.
 513 Auxiliary data can be used to amortize the cost of evaluating $\text{EXTSOL}\cdot A$ repeatedly, generally
 514 to prove that the path time is equal to the complexity of solving $\text{EXTSOL}\cdot A$ once, e.g., when
 515 generating minimal models of monotone CNF [31].

516 Using flashlight search, we obtain that $\Pi_A \in \text{DelayP}$ if $\text{EXTSOL}\cdot A \in \text{P}$ and indeed many
 517 enumeration problems are in DelayP because their extension problem are in P , see e.g.,
 518 [38, 29]. However, there are NP-hard extension problems whose enumeration problem is in
 519 DelayP , e.g., the extension of a maximal clique, whose hardness can be derived from the fact
 520 that finding the largest maximal clique in lexicographic order is NP-hard [23].

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

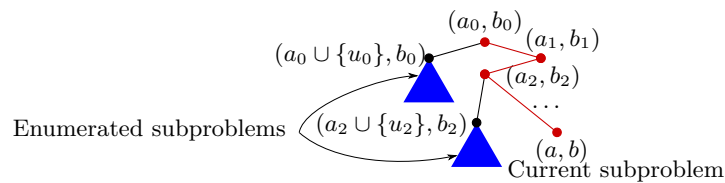
527 To make sense of very low complexity enumeration algorithms, we may separate the
 528 preprocessing $T(x, 1)$ from the rest of the computation. We say that a machine with
 529 preprocessing has incremental delay $d(n)$ if, for all x and i , $T(x, i) - T(x, 1) \leq i \cdot d(|x|)$. The
 530 preprocessing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

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

540 **Proof.** Let I be the flashlight search algorithm solving Π_A . Let us first describe an algorithm
 541 I' in incremental linear time, which produces the same solutions as I on any input x of size
 542 n . The preprocessing of I' is to run I for $p(n)$ steps and to store each solution output in
 543 a queue. It takes a time at most $O(p(n)b(n))$ since there are at most $p(n)$ solutions of size
 544 $b(n)$ to store in the queue. The queue requires an additional space of $O(p(n)b(n))$. After the
 545 preprocessing, we first output all solutions in the queue and then I is simulated for the rest
 546 of its run and the solutions output by I are output by I' right away.

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

552 If the queue is empty, the number of solutions output by I' is the same as the number
 553 of solutions output by I after running for a time $p(n) + t$. At this point in time, the
 554 flashlight search is considering some node (a, b) of the partial solutions tree and we denote
 555 by $(\emptyset, U(x)) = (a_0, b_0), \dots, (a_i, b_i) = (a, b)$ the path from the root to (a, b) . The time spent
 556 on the nodes of this path is bounded by $p(n)$, the path time of I . Hence, I spends at least a
 557 time t in the subtrees whose root is a child of some (a_i, b_i) .



■ **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.

558 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
 559 been either completely explored by the flashlight search or not at all, as shown in Figure 1.
 560 Since I is a flashlight search, it works recursively on subtrees, corresponding to subproblems.
 561 If a subtree rooted at (c, d) has been completely explored, then the flashlight search has
 562 recursively solved the problem $\tilde{A}(x, c, d)$. By definition of the average delay, the solutions
 563 in $\tilde{A}(x, c, d)$ have been produced by flashlight search in total time less than $d(n)\#\tilde{A}(x, c, d)$.
 564 Hence, the subproblems entirely solved by I contribute at least $t/d(n)$ solutions. Therefore,
 565 in time Ct , I' outputs at least $t/d(n)$ solutions.

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

569 4.2 Enumeration of the Models of DNF Formulas

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

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

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

589 **Proof.** The algorithm of [12] enumerates all models with average delay $O(nm^{1-\log_3(2)})$ and
 590 the space used is the representation of the DNF formula by a trie, that is $O(mn)$. We apply
 591 Theorem 12 to this algorithm. We have a bound on the incremental delay, the space used
 592 and the number of solutions, hence we can use Theorem 3 to do the geometric amortization
 593 without overhead in the method of Theorem 12. The auxiliary queue used in Theorem 12 is

594 of size n^2m , since the path time is nm . The number of models is bounded by 2^n , hence the
 595 delay obtained by amortization is $O(n^2m^{1-\log_3(2)})$ and the space $O(n^2m)$, which proves the
 596 theorem. \blacktriangleleft

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

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

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

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

610 ——— References ———

- 611 1 Alfred V Aho and John E Hopcroft. *The design and analysis of computer algorithms*. Pearson
 612 Education India, 1974.
- 613 2 Ricardo Andrade, Martin Wannagat, Cecilia C Klein, Vicente Acuña, Alberto Marchetti-
 614 Spaccamela, Paulo V Milreu, Leen Stougie, and Marie-France Sagot. Enumeration of minimal
 615 stoichiometric precursor sets in metabolic networks. *Algorithms for Molecular Biology*, 11(1):25,
 616 2016.
- 617 3 David Avis and Komei Fukuda. Reverse search for enumeration. *Discrete Applied Mathematics*,
 618 65(1-3):21–46, 1996.
- 619 4 Guillaume Bagan. *Algorithms and complexity of enumeration problems for the evaluation of*
 620 *logical queries*. PhD thesis, Université de Caen, France, 2009.
- 621 5 JoséL Balcázar. Self-reducibility. *Journal of Computer and System Sciences*, 41(3):367–388,
 622 1990.
- 623 6 Dominique Barth, Olivier David, Franck Quessette, Vincent Reinhard, Yann Strozecki, and
 624 Sandrine Vial. Efficient generation of stable planar cages for chemistry. In *International*
 625 *Symposium on Experimental Algorithms*, pages 235–246. Springer, 2015.
- 626 7 James R Bitner, Gideon Ehrlich, and Edward M Reingold. Efficient generation of the binary
 627 reflected gray code and its applications. *Communications of the ACM*, 19(9):517–521, 1976.
- 628 8 Thomas Bläsius, Tobias Friedrich, Julius Lischeid, Kitty Meeks, and Martin Schirneck. Effi-
 629 ciently enumerating hitting sets of hypergraphs arising in data profiling. In *2019 Proceedings*
 630 *of the Twenty-First Workshop on Algorithm Engineering and Experiments (ALENEX)*, pages
 631 130–143. SIAM, 2019.
- 632 9 Kateřina Böhmová, Luca Häfliger, Matúš Mihalák, Tobias Pröger, Gustavo Sacomoto, and
 633 Marie-France Sagot. Computing and listing st-paths in public transportation networks. *Theory*
 634 *of Computing Systems*, 62(3):600–621, 2018.
- 635 10 Caroline Brosse, Vincent Limouzy, and Arnaud Mary. Polynomial delay algorithm for minimal
 636 chordal completions. In Mikolaj Bojanczyk, Emanuela Merelli, and David P. Woodruff, editors,
 637 *49th International Colloquium on Automata, Languages, and Programming, ICALP 2022, July*
 638 *4-8, 2022, Paris, France*, volume 229 of *LIPICs*, pages 33:1–33:16. Schloss Dagstuhl - Leibniz-
 639 Zentrum für Informatik, 2022. URL: <https://doi.org/10.4230/LIPICs.ICALP.2022.33>,
 640 doi:10.4230/LIPICs.ICALP.2022.33.

- 641 11 Florent Capelli and Yann Strozecki. Incremental delay enumeration: Space and time. *Discrete*
642 *Applied Mathematics*, 268:179–190, 2019.
- 643 12 Florent Capelli and Yann Strozecki. Enumerating models of DNF faster: Breaking the
644 dependency on the formula size. *Discrete Applied Mathematics*, 303:203–215, 2021.
- 645 13 Florent Capelli and Yann Strozecki. Geometric amortization of enumeration algorithms. *arXiv*
646 *preprint arXiv:2108.10208*, 2021.
- 647 14 Nofar Carmeli and Markus Kröll. On the enumeration complexity of unions of conjunctive
648 queries. In *Proceedings of the 38th ACM SIGMOD-SIGACT-SIGAI Symposium on Principles*
649 *of Database Systems*, pages 134–148, 2019.
- 650 15 Sara Cohen, Benny Kimelfeld, and Yehoshua Sagiv. Generating all maximal induced subgraphs
651 for hereditary and connected-hereditary graph properties. *Journal of Computer and System*
652 *Sciences*, 74(7):1147–1159, 2008.
- 653 16 Alessio Conte, Roberto Grossi, Andrea Marino, and Luca Versari. Listing maximal subgraphs
654 satisfying strongly accessible properties. *SIAM Journal on Discrete Mathematics*, 33(2):587–
655 613, 2019.
- 656 17 Alessio Conte and Takeaki Uno. New polynomial delay bounds for maximal subgraph
657 enumeration by proximity search. In *Proceedings of the 51st Annual ACM SIGACT Symposium*
658 *on Theory of Computing*, pages 1179–1190, 2019.
- 659 18 Stephen A Cook and Robert A Reckhow. Time bounded random access machines. *Journal of*
660 *Computer and System Sciences*, 7(4):354–375, 1973.
- 661 19 Arnaud Durand and Etienne Grandjean. First-order queries on structures of bounded degree
662 are computable with constant delay. *ACM Trans. Comput. Log.*, 8(4):21, 2007.
- 663 20 Thomas Eiter, Georg Gottlob, and Kazuhisa Makino. New results on monotone dualization
664 and generating hypergraph transversals. *SIAM Journal on Computing*, 32(2):514–537, 2003.
- 665 21 Michael Fredman and Leonid Khachiyan. On the complexity of dualization of monotone
666 disjunctive normal forms. *Journal of Algorithms*, 21(3):618–628, 1996.
- 667 22 Leslie Ann Goldberg. *Efficient algorithms for listing combinatorial structures*. PhD thesis,
668 University of Edinburgh, UK, 1991. URL: <http://hdl.handle.net/1842/10917>.
- 669 23 David S Johnson, Mihalis Yannakakis, and Christos H Papadimitriou. On generating all
670 maximal independent sets. *Information Processing Letters*, 27(3):119–123, 1988.
- 671 24 Leonid Khachiyan, Endre Boros, Khaled Elbassioni, Vladimir Gurvich, and Kazuhisa Makino.
672 On the complexity of some enumeration problems for matroids. *SIAM Journal on Discrete*
673 *Mathematics*, 19(4):966–984, 2005.
- 674 25 Samir Khuller and Vijay V Vazirani. Planar graph coloring is not self-reducible, assuming $p \neq$
675 np . *Theoretical Computer Science*, 88(1):183–189, 1991.
- 676 26 Donald E Knuth. Combinatorial algorithms, part 1, volume 4a of the art of computer
677 programming, 2011.
- 678 27 Eugene L Lawler, Jan Karel Lenstra, and AHG Rinnooy Kan. Generating all maximal
679 independent sets: NP-hardness and polynomial-time algorithms. *SIAM Journal on Computing*,
680 9(3):558–565, 1980.
- 681 28 Édouard Lucas. *Récréations mathématiques: Les traversées. Les ponts. Les labyrinthes. Les*
682 *reines. Le solitaire. La numération. Le baguenaudier. Le taquin*, volume 1. Gauthier-Villars et
683 fils, 1882.
- 684 29 Arnaud Mary and Yann Strozecki. Efficient enumeration of solutions produced by closure
685 operations. *Discrete Mathematics & Theoretical Computer Science*, 21(3), 2019.
- 686 30 Kurt Mehlhorn. *Data structures and algorithms 1: Sorting and searching*, volume 1. Springer
687 Science & Business Media, 2013.
- 688 31 Keisuke Murakami and Takeaki Uno. Efficient algorithms for dualizing large-scale hypergraphs.
689 *Discrete Applied Mathematics*, 170:83–94, 2014.
- 690 32 Ronald C Read and Robert E Tarjan. Bounds on backtrack algorithms for listing cycles, paths,
691 and spanning trees. *Networks*, 5(3):237–252, 1975.

20:18 Geometric Amortization of Enumeration Algorithms

- 692 33 Frank Ruskey. Combinatorial generation. *Preliminary working draft. University of Victoria,*
693 *Victoria, BC, Canada*, 11:20, 2003.
- 694 34 Yann Strozecki. *Enumeration complexity and matroid decomposition*. PhD thesis, Université
695 Paris Diderot - Paris 7, 2010.
- 696 35 Yann Strozecki. Enumeration complexity. *Bulletin of EATCS*, 1(129), 2019.
- 697 36 James C Tiernan. An efficient search algorithm to find the elementary circuits of a graph.
698 *Communications of the ACM*, 13(12):722–726, 1970.
- 699 37 Nikolaos Tziavelis, Wolfgang Gatterbauer, and Mirek Riedewald. Any-k algorithms for
700 enumerating ranked answers to conjunctive queries, 2022. URL: [https://arxiv.org/abs/](https://arxiv.org/abs/2205.05649)
701 [2205.05649](https://arxiv.org/abs/2205.05649), doi:10.48550/ARXIV.2205.05649.
- 702 38 Takeaki Uno. Constant time enumeration by amortization. In *Workshop on Algorithms and*
703 *Data Structures*, pages 593–605. Springer, 2015.
- 704 39 Kunihiro Wasa and Kazuhiro Kurita. Enumeration of enumeration algorithms and its com-
705 plexity. 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: $\text{inc}(c)$ increases the counter by 1 and $\text{mbit}(c)$ returns the index of the most significant bit of the value encoded by c . In other words, if $k = \text{mbit}(c)$ then we know that $\text{inc}(c)$ 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

750 Algorithm G is that we maintain a stack containing the positions of the 1-bits of the code in
 751 increasing order so that we can retrieve the next bit to switch in constant time which is not
 752 obvious in Algorithm G. Our approach is closer to the one presented in Algorithm L of [26]
 753 but for technical reasons, we could not use it straightforwardly.

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

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

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

766 We will implement `mbit` and `inc` to ensure the following invariants: the bits of the Gray
 767 Code encoding the value of the counter are stored in R_0, \dots, R_{k-1} . A contains the parity of
 768 the number of 1 in R_0, \dots, R_{k-1} . M contains an integer smaller than k that is the position
 769 of the most significant bit in the Gray Code (the biggest $j \leq k - 1$ such that R_j contains
 770 1). Finally, S contains all positions j such that R_j is set to 1 in decreasing order (that is if
 771 $j < j'$ are both in S , j will be popped before j').

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

776 To implement `inc`, we simply follow Algorithm G from [26]. If A is 0 then we swap the
 777 value of R_0 . Otherwise, we swap the value of R_{j+1} where j is the smallest position such
 778 that $R_j = 1$ (if j is $k - 1$ then we have reached the maximal value of the code which we
 779 have assumed to be impossible, see below to handle unbounded counters). One can find
 780 j in constant time by just popping the first value in S , which works if the invariant is
 781 maintained. Now, one has to update the auxiliary memory: A is replaced by $1 - A$ so that it
 782 still represents the parity of the number of 1 in the Gray Code. To update S , we proceed as
 783 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
 784 or R_0 has gone from 1 to 0, in which case we have to pop one value in S , which will be 0
 785 since S respects the invariant. It can be readily proven that this transformation preserves
 786 the invariant on S . Now, if A is 1, then either the value of R_{j+1} has gone from 0 to 1 which
 787 means that we have to push $j + 1$ and j on the stack (j is still the first bit containing 1 so it
 788 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
 789 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
 790 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
 791 $j + 1$ from S and push back j . Again, it is easy to see that these transformations preserve
 792 the invariant on S . Moreover, we never do more than 2 operations on the stack so this can
 793 be done in constant time. Finally, if R_{j+1} becomes 1 and $j + 1 > M$, we set M to $j + 1$.

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

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

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

806 To summarize, we have proven the following:

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

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

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

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

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

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

833 To simulate I in constant time, the crucial part is to be able to read and write the i th
 834 register of I as stored in M in constant time. If we know a bound $s(n)$ on the space used
 835 by I , and a bound on the number of solutions $\#I(x)$ as in Algorithm 1, the structure M
 836 is very simple. For a structure M , we reserve $s(n)$ registers which are mapped one to one
 837 to the registers R_1 up to $R_{s(n)}$ of I . We also require 1 register to store the index of the
 838 current instruction to be executed by I . We also initialize a counter c to 0 as explained in
 839 Section A.2 for **steps**(M) to keep track of the number of steps that have been simulated so
 840 far. This counter will use up to $O(\log(\#I(x))^2)$ registers. To really account for **steps**(M), one

841 should increment c each time an instruction `move` is executed. However, in Algorithm 1 and
 842 Algorithm 2, one need to compare `steps(M)` with another value. We explain below how one
 843 can adapt this counter so that this comparison is constant time for both algorithms.

844 Let $m = s(n) + 2\lceil \log(\#I(x)) \rceil + 2$, then for all j from 0 to $\lceil \log(\#I(x)) \rceil + 1$, the structure
 845 $M[j]$ uses the registers from jm to $(j+1)m - 1$. Hence, if $M[j]$ must simulate the access of
 846 I to register R_i , it accesses the register R_{jm+i} . This operation is in constant time, since it
 847 requires to compute $jm + i$, where i , m and j are polynomial in n .

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

859 A.4 Instruction copy

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

868 In this section, we explain how one can lazily implement this functionality so that the
 869 memory of M is copied only when needed. This method ensures that `copy` runs in $O(1)$,
 870 however, there is an overhead to the cost of the instruction `move`. We show it still runs in
 871 $O(1)$ if the memory usage of I is well behaved, otherwise the overhead is small and exists
 872 only when $\log(\#I(x)) \leq \log(n)^2$.

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

882 We explain how we lazily copy an array. Assume we want to create an exact copy of the
 883 array A of size m . We create both A' and U of size m initialized to zero. The value of $U[r]$
 884 is 0 if $A'[r]$ has not been copied from $A[r]$ and 1 otherwise. Each time `move(M)` is executed
 885 and it modifies the value $A[r]$, if $U[r] = 0$, it first set $A'[r] = A[r]$ and $U[r] = 1$. Each time
 886 `move(M')` is executed and reads the value $A'[r]$, if $U[r] = 0$, it first set $A'[r] = A[r]$ and
 887 $U[r] = 1$. This guarantees that the value of A' is always the same as if we had completely

888 copied it from A when the instruction $\text{copy}(M)$ is executed. The additional checks and
 889 updates of U add a constant time overhead to move . Moreover, we maintain a simple counter
 890 c , and each time a $\text{move}(M')$ operation is executed, if $U[c] = 0$, we set $A'[c] = A[c]$ and
 891 $U[c] = 1$. When $c = m$, the copy is finished and we can use A and A' as before, without
 892 checking U .

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

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

$$901 \quad O(\log(\#I(x))p(n))$$

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

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

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