A note on polynomial delay vs. incremental delay

Yann Strozecki

September 10, 2015

In this note we explore several problems related to enumeration complexity. In particular, we are interested by problems solvable by polynomial delay algorithms (the class **DelayP**), that is between each produced solution there is a delay polynomial in the *input* of the problem. The second class we are interested in, is the set of problems solvable by an incremental delay algorithm (the class **IncP**), that is the time to produce the ith solution is a polynomial in i and the size of the input. In both cases, we also require the solutions to be testable in polynomial time, that is all considered enumeration problems are included in **EnumP**. For more details on enumeration complexity, please refer to [7].

1 Relationship between IncP and DelayP

There is an alernative way of defining the class **IncP** which makes it relationship to **DelayP** more obvious.

Definition 1. $\mathbf{IncP_{k,l}}$ is the set of enumeration problems such that there is an algorithm which produces m solutions (if they exist) from an input of size n in time $O(m^k n^l)$.

$$\mathbf{IncP_k} = \bigcup_{l \geq 1} \mathbf{IncP_{k,l}}$$

$$\mathbf{IncP} = \bigcup_{k \geq 1} \mathbf{IncP_k}$$

One can argue that this definition is more natural from a practical point of view. Indeed we are mostly interested by bounding the time to obtain a large number of solutions rather than bounding the delay. One could hope that bounding the delay may ensure some regularity in the process of enumeration, but it turns out that the use of a large quantity of memory is sufficient to emulate a polynomial delay algorithm from an incremental one.

Theorem 2. $\mathbf{IncP}_1 = \mathbf{DelayP}$

Proof. The inclusion $\mathbf{IncP}_1 \supseteq \mathbf{DelayP}$ is clear, since to generate m solutions by an algorithm with delay $P(n)$, we need a time $mP(n)$.

Now, consider a problem in $\mathbf{IncP}_1$, that is we can generate the first m solutions in time less than mn^k for some k. We run this algorithm but instead of outputting the results we store them in memory. Then we use some counter to output a stored solution each time the simulated algorithm has spent a time n^k. Because the simulated algorithm generates m solutions in time mn^k, the process we have described will never run out of solutions stored in memory. The delay is bounded by n^k times some constant to deal with the counter and the storage of the solutions. □

Now that we have seen that **DelayP** is essentially the first level of the **IncP** hierarchy, the question of the separation of **DelayP** from **IncP** is implied by a proof that the **IncP** hierarchy is strictly increasing. Note that the difficulty of this question is that we ask for all our problems to be in **EnumP**. Otherwise we could separate unconditionally the **IncP** hierarchy by using the time hierarchy theorem.

Theorem 3. *If the strong exponential time hypothesis holds then for all* $a < b$, $\mathbf{IncP}_a \subsetneq \mathbf{IncP}_b$.

Proof. We consider the problem $Gap_t(DominatingSet)$, which is given a graph G of size n and an integer k list all dominating set of size k and all integers less than $n^{\frac{k}{t}}$. It is a padded version of enumerating all dominating sets. Remark that $Gap_t(DominatingSet) \in \mathbf{IncP}_t$, since we can first generate the $n^{\frac{k}{t}}$ trivial solutions and we have more than $(n^{\frac{k}{t}})^t$ time to generate the rest of the solutions, which is easy by a brute force search.

Now assume that $\mathbf{IncP}_a = \mathbf{IncP}_b$. It means, that $Gap_b(DominatingSet) \in \mathbf{IncP}_a$. Therefore there is an algorithm which, after a time $(n^{\frac{k}{b}})^a n^c$ where c is some constant, produces more than $n^{\frac{k}{b}}$ solutions if possible. Since we have generated more solutions thant there are trivial solutions (integers), we also have generated a dominating set of size k. Therefore we have proved that we can find a dominating set of size k in time $n^{\frac{a}{b}k+c}$. For k large enough, we have a complexity of $n^{k-\epsilon}$, since $a < b$, which contradicts $SETH$ by Theorem 2.1 of [5].

□

This proof has been improved recently so that we can substitute ETH to SETH while working with Florent Capelli and Arnaud Durand. Thanks to Florent who have done most of the work and written the following proof.

Let $k \in \mathbb{N}$ and let

$$s_k = \inf \{d \mid \text{there exists a } O(2^{dn}) \text{ algorithm for } k\text{-SAT}\}.$$

The *Exponential Time Hypothesis* (ETH) states that for all $k \in \mathbb{N}$, $s_k > 0$. One immediate consequence of ETH is that there is no $2^{o(n)}$ algorithm for k-SAT. It is however less restrictive since there could be a $O(2^{\epsilon n})$ algorithm for k-SAT for all $\epsilon > 0$, but it could be to hard for a given n to select the right algorithm that could lead to a $2^{o(n)}$ algorithm for k-SAT.

It is easy to see that s_k is an increasing sequence since an instance of k-SAT is also an instance of l-SAT for $k < l$. Thus if $s_3 > 0$, then ETH holds.

Since $s_k \leq 1$ (we can solve k-SAT by bruteforcing the solution), we know that $(s_k)_{k\in\mathbb{N}}$ has a limit s. The *Strong Exponential Time Hypothesis* (SETH) states that $s = 1$. An immediate consequence of SETH is that for all $d < 1$, there is no $O(2^{dn})$ algorithm for SAT.

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

- the integers from 1 to $2^{nt} - 1$
- the solutions of ϕ duplicated 2^n times each, that is $\mathsf{Sol}(\phi) \times [2^n]$

Lemma 4. *Let $d < 1$. If we have a $O(2^{dn})$ algorithm for 3-SAT, then for all $a \in \mathbb{N}$, $R_{\frac{d}{a}}$ is in Inc_a.*

Proof. We enumerate the integers from 1 to $2^{\frac{dn}{b}} - 1$ and then call the algorithmm to find a solution of ϕ. We have enough time to run this algorithm since we are expected to give the next input in time $O(2^{\frac{dn}{b}b}) = O(2^{dn})$. If the formula is no satisfiable, then we stop the enumeration. Otherwise, we enumerate all copies of the discovered solution. We have then enough time to bruteforce the other solutions. □

Lemma 5. *If R_t is in Inc_a, then there exists a $O(2^{nta})$ algorithm for 3-SAT.*

Proof. Since R_t is in Inc_a, we have an algorithm for R_t that outputs m elements of $R_t(\phi)$ in time $O(m^a|\phi|^c)$ for a constant c. We can then output 2^{nt} elements of R_t in time $O(2^{nta}|\phi|^c)$. In this enumeration, we have necessarely enumerated at least one solution of ϕ which gives the expected algorithm. □

Lemma 6. *If $\mathsf{Inc}_a = \mathsf{Inc}_b$, then for all i, $R_{\frac{a^i}{b^{i+1}}}$ is in Inc_a.*

Proof. The proof is by induction on i. For $i = 0$, by Lemma 4, $R_{\frac{1}{b}}$ is in $\mathsf{Inc}_b = \mathsf{Inc}_a$ since we have a $O(2^n)$ bruteforce algoritm for 3-SAT.

Now if $R_{\frac{a^i}{b^{i+1}}}$ is Inc_a, then by Lemma 5, we have a $O(2^{dn})$ algorithm for $d = \frac{a^{i+1}}{b^{i+1}}$. Thus, by Lemma 4, $R_{\frac{d}{b}} = R_{\frac{a^{i+1}}{b^{i+2}}}$ is in $\mathsf{Inc}_b = \mathsf{Inc}_a$. □

Theorem 7. *If* ETH *holds, then* $\mathsf{Inc}_a \subsetneq \mathsf{Inc}_b$ *for all* $a < b$.

Proof. If there exists $a < b$ such that $\mathsf{Inc}_a = \mathsf{Inc}_b$, then by Lemma 6, for all i, $R_{\frac{a^i}{b^{i+1}}}$. Thus by Lemma 5, we have a $2^{d_i n}$ algorithm for 3-SAT , where $d_i = \left(\frac{a}{b}\right)^i$. Thus $s_3 = 0$ which violates ETH. □

Open problem :

1. Is it possible to prove such a separation under a weaker conjecture, such as $\mathsf{P} \neq \mathsf{NP}$?
2. Are $\mathbf{IncP}_1$ and **DelayP** different, when we add the constraint of using a space polynomial in the input ?

2 Problem with a delay polynomial in the size of one solution

Ideally, we would like to enumerate the solutions of a problem with a delay polynomial (and even linear) in the size of each solution. It has been proved for the enumeration of solutions to a MSO querie over a bounded tree width structure [1]. In this section we present several problems, whose solutions can be much smaller thant the size of the input. It turns out that these problems all admit simple polynomial delay algorithms. However the delay depends on the size of the input and not only on the size of the solutions. We show reductions between those problems, such that an algorithm whose delay depends only on the size of the solutions for one would yield the same kind of algorithm for the others. Ultimately, we would like to prove that such algorithms do not exist, modulo some complexity hypothesis.

2.1 The Restricted Solutions Tree Method

Here we explain a very classical and natural enumeration method that we call the *Restricted Solutions Tree Method*. (trouver des références) We will see the solutions as functions from $[n]$ to $[k]$. In practice solutions are often

subsets of $[n]$ which means that $k = 1$ and the function is the characteristic function of the subset.

The algorithm is a depth first traversal of a tree whose nodes are partial solutions. The nodes of the tree will be all function g from $[l]$ to $[k]$, for all $l \leq n$, such that g is a restriction on $[l]$ of a function g' which is a solution. The children of the node g will be the functions defined over $[l+1]$, which restricted to $[l]$ are equal to g.

The leaves of this tree are the solutions of our problem, therefore a depth first traversal will visit all leaves which allows to enumerate all solutions. We want an enumeration algorithm with a delay polynomial in n. Since a branch of the tree is of size n, we need to be able to find the children of a node in a time polynomial in n to obtain a polynomial delay (we assume that k is a constant). Therefore the problem is reduced to the following *decision* problem: given g from $[l]$ to $[k]$ is there a g' solution such that g is its restriction on $[l]$? This problem is called the extension problem, that is can we extend a partial solution to a full solution.

Proposition 8. *Given an enumeration problem A, whose solutions to an instance can be seen as functions from $[n]$ to $[k]$ where n is polynomially related to the size of the instance. If the extension problem associated to A is in* P*, then A is in* **DelayP**.

2.2 DNF

We introduce a very simple and natural problem, the enumeration of the solutions of a DNF formula. It is a task one may want to achieve to compute an equivalent CNF formula (dualization). Remark that, contrary to CNF, it is easy to find one solution if there is one and we will show how to enumerate all of them relatively efficiently. On the other hand counting all solutions of a DNF formula is a ♯P-complete [8].

ENUM·*DNF*

Input: ϕ a DNF formula with n clauses and k variables

Output: an enumeration of all variables assignment satisfying ϕ

Remark that generating the model of a single clause can be done by a Gray code enumeration, that is in constant delay (see [2]). Then we can do the union of the solutions of each clause to obtain the whole set of solutions. That can be done with a polynomial delay of $O(kn^2)$ as explained in proposition 2.40 of [7]. We will see in the next proposition, that the restricted solutions tree method has a better complexity.

Proposition 9. *There is an algorithm which enumerates all models of a DNF formula with a delay $O(nk)$.*

Proof. We define the data structures that we maintain in the DFS of the restricted solution tree to decide efficiently the extension problem. First we compute for each variable and negated variable the list of clauses where its negation appears. We also maintain the list of clauses which can be true and the ones which are false for the current partial assignment as well as the number of still valid clauses. Then at some step of the algorithm, we have already built an assignment of the l first variables and we choose an assignment for the variable x_{l+1}. We go over the list of clauses associated to negation of x_{l+1} and disable them then we update the number of valid clauses. If this number is 0, there is no possible extension and we go back in the enumeration tree, otherwise there exist some valid extension. Between two produced solutions, the number of nodes visited in the tree is at most $2k$ and the time to maintain the array of valid clauses is proportional to the sum of the lists associated to the variables that is $O(nk)$. Therefore the delay is $O(nk)$. □

Remark that the total time is used to do the enumeration is $O(nkt)$ with t the total number of solutions. On the other hand enumerating the solutions of every clause and not caring about repetition can be done in time $O(nt)$. Therefore we are not far from optimality to take care of redundancy in the union and any improvement should come from a different point of view on the problem.

One natural restriction is to consider monotone DNF formula (without negation). For monotone CNF, the problem is equivalent to hitting set enumeration, studied in [4]. An efficient hill climbing algorithm is used to obtain a $O(n)$ delay algorithm.

We maintain datastructures along with a current assignment to a CNF formula. Let $N(x) = \{c \text{ clause which includes } x\}$. For each clause c, let $cov(c) = \{x \in c \mid x \text{ is positive in the assignment}\}$ and $crit(x) = \{c \in N(x) \mid cov(c) = 1\}$ and let $cand = \{x \text{ positive} \mid crit(x) = \emptyset\}$. The algorithm starts by the all true assignment and try recusively to set each variable in cand to false. When it sets x to false, the algorithm goes over the list $N(x)$ and for each c in this list remove x from $cov(c)$. When a $cov(c)$ becomes of size one, the algorithm adds c to $crit(y)$, for the relevant y and remove y from $cand$. This can be implemented in $O(n)$ when n is the number of clauses of the CNF. The set $cov(y)$ must be represented by both an array and a list, each element of the array pointing to the corresponding element in the list.

This method works for monotone CNF only, and we are stil not able to extend it to monotone DNF. One can hope to reduce the enumeration of solutions of a monotone DNF to the enumeration of solutions of a montone CNF. It is certainly not possible the other way around, since it is easy to approximate the number of solutions of a DNF but hard for a monotone CNF.

The maximal cover enumeration problem is the following: given an hypergraph on n vertices and m hyperedges $\mathcal{H}$, we want to enumerate all set of hyperedges $S \subseteq \mathcal{H}$ such that no hyperedge $e \notin S$ is covered. If one is able to solve the problem with delay $O(m)$, we would obtain a $O(m)$ delay for enumeration of the models of DNF. It could well be that this problem is harder than enumerating the models of DNF, even though it is easy to solve with a polynomial delay.

Open question:

1. Can we use the hill climbing algorithm to achieve a delay $O(n)$ for montone or general DNF formulas ?

2. Can we give a reduction between DNF and monotone DNF which respects enumeration complexity ?

3. There is a generalization of DNF called DNNF. Its also easy to enumerate its solutions in polynomial delay. Can we reduce it to DNF ? Can we prove some hardness result for this problem ?

2.3 Enum·$\mathbf{\Sigma}_1$

This problem and the reduction is from [2]. $\mathbf{\Sigma}_1$ is the class of formula $\exists x \phi(x, X)$ where x is a tuple of k first order variables and ϕ is quantifer free. X is a set of free variables.

Proposition 10. *Let $\exists x \phi(x, X) \in \mathbf{\Sigma}_1$, M a model and k the size of the tuple x. There is an algorithm which enumerates all X satisfying $\exists x \phi(x, X) \in \mathbf{\Sigma}_1$ in M with a delay polynomial in $|M|$.*

The proof from [2] relies on replacing the existential quantifiers by a union over all k-tuples of elements in M. Then the enumeration can be carried out by the method to generate an union of solutions. Contrary to what was stated in [2], the delay is $|M|^{2k}$. The following reduction to DNF yields an enumeration algorithm with delay $|M|^{k+1}$ for monadic variables.

Proposition 11. *Assume we can enumerate the solutions of DNF with precomputation $g(n,k)$ and delay $f(n,k)$, where k is the number of variables of the formula, and n the number of clauses. Then for all formulas $\Phi \in \Sigma_1$,* Enum·Φ *with monadic free variables can be solved with precomputation $g(d^l,d)$ and delay $f(d^l,d)$, where d is the size of the domain and l is the number of existentially quantified variables.*

Proof. To prove that, fix a formula $\Phi \equiv \exists x \phi(x,T)$, where x is an existentially quantified tuple of size l. Let $\mathcal{S}$ be the input structure and D its domain of size n. Let $\tilde{\Phi}$ be the disjunction of the d^l formulas $\Phi(x^*,T)$ where x^* ranges over D^l. Each of the formula $\Phi(x^*,T)$ can be written as a DNF formula, whose variables are the monadic variables applied to some x^*. There are at most $O(l)$ distincts variables, therefore the clauses are of size at most $O(l)$. Therefore the formula $\tilde{\Phi}$ is in DNF, has $O(d)$ variables and its solutions are in bijection with the solutions of Φ. □

The reciprocal is also true.

2.4 Closure by union

Union closure

Input: An hypergraph H with n hyperedges and k vertices

Output: enumerate all unions of edges of H

Remark that if we replace the word union by extension, we exactly get the problem monotone-DNF. It is also reminiscent of the problem of listing hypergraph transversals, but it is much simpler as we will see.

Proposition 12. *There is an algorithm which solves Union closure with a delay $O(nk)$.*

Proof. We use the restricted solutions tree traversal and datastructures similar to the ones used for DNF. For each vertex, build the list of hyperedges it appears in. We maintain the array of hyperedges which cover a partial solution (a subset of vertices) and also an array which maintain the number of time each vertex is covered by an hyperedge. Each time we extend a partial solution, either we add the vertex v and the only thing we have to do is check wether this vertex is covered or we choose to note take the vertex and we remove all hyperedges not already removed which contains it. If some vertex is then covered by zero hyperedges and is in the partial solution then the extension is not possible. Between the output of two solutions we will at most go over all k vertices and thus we will do in total at most $O(kn)$ operations to maintain the datastructures. □

The similarity of the algorithm with the one to solve DNF enumeration is just an illustration of a deeper connection between those two problems, given in the next proposition.

Proposition 13. *There is a parsimonious reduction from monotone-DNF to Union closure.*

Proof. Let $\phi \equiv \bigvee_{i=1}^{n} C_i$ where the C_i are clauses over the variables $x_1, \ldots, x_k$. We build an hypergraph H over the domain $[k]$. For each clause C_i, let e_i be the hyperedge $\{i \mid x_i \in C_i\}$ and we also add the hyperedges $e_i \cup \{x_j\}$ for all $x_j \notin E$. There is a bijection between the union of hyperedges and the solution of the formula ϕ. □

Since the reduction is parsimonious, counting solutions to Union closure is ♯P-hard. Moreover an instance of Union closure which is a r-uniform hypergraph can be reduced to a $r - DNF$ formula.

Open problem:

1. Is there a reduction from DNF to Union closure ?
2. Is there a reduction from Union Closure to DNF ?

2.5 Fixing a parameter

A few tight reductions and open problems.

Open problem: if we restrict the inputs of union closure to graphs, can we do better than a $O(n^3)$ delay ? Same question for $2DNF$.

3 Extension to the CSL paper

Here we study the problem Enum·**FO** studied in [2]. It has been proved [6] that when universal quantifiers are allowed, it is hard to even find the first solution. It is hard even if we ask the model to be of bounded degree. That is why we study a stronger restriction on the model : its signature must be *empty*.

The problem is thus the following, given some integer n in unary which represents the size of the domain, and a formula $\phi(X)$ with free second order variables, enumerate all X such that $([n], X) \models \phi$. First remark that by Fagin theorem [3] the logic $\exists SO$ captures NP. From that result, we obtain that finding one solution which satisfies $\phi(X)$ is NP-hard if the size of the domain is given in unary or in NEXP-hard if the size is given in binary.

Since the case with general second order variable is hard, we restrict ourselves to the case of monadic second order variables only. Warning: all the following is implied by the results of Courcelle [1].

We will show how to obtain a nice normal form and a simpler problem from ϕ, when it is in Π_1. From that simple form, a good enumeration algorithm can be reduced to solving some *decision* problem in polynomial time. In fact we will show that for all possible ϕ we can reduce the enumeration problem to a decision one.

We consider $\forall x \phi(x, X)$, where ϕ is a quantifier free formula. Let k be the number of first order variable x and l the number of second order variable X in ϕ. We write ϕ in CNF form, with at worst, an exponential blowup of the size of the formula:

$$\forall x \bigwedge_i \bigvee_j T_{i,j}$$

where the T_i's are of the form $x \in X$ or $x \notin X$. This formula can be rewritten

$$\bigwedge_i (\forall x \bigvee_j T_{i,j})$$

For each j, we rewrite $\bigvee_j T_{i,j}$ as $\bigvee_{j=1}^{k} T'_{i,j}$ where $T'_{i,j}$ is a disjunction of $x_j \in X$ or $x_j \notin X$. In other word we have regrouped all terms involving the same first order variable. Remark that if if we have $x_j \in X_a$ and $x_j \notin X_a$ in $T'_{i,j}$ then it is always true. Now remark that $\forall x \bigvee_{j=1}^{k} T'_{i,j}$ is equivalent to $\bigvee_{j=1}^{k} \forall x_j T'_{i,j}$ because each $T'_{i,j}$ depends on single distinct variable.

The formula ϕ is now equivalent to $\bigwedge_i \bigvee_{j=1}^{k} \forall x_j T'_{i,j}$. We can expand the conjunction to obtain $\bigvee_f \bigwedge_i \forall x_{f(i)} T'_{i,f(i)}$. We want to enumerate the X solutions to this formula and we have proved that we can get rid of the union at the beginning of a formula in [2] for a delay multiplied by the size of the union which is here a constant. Therefore we have to devise an enumeration algorithm for $\bigwedge_i \forall x_{f(i)} T'_{i,f(i)}$.

We propose a classical generation procedure by successive extension of

partial solution. Let n be the size of the domain, we associate to $\bigwedge_i \forall x T'_i$ a tree whose leaves are the solutions we want to generate. A node is a partial solution, that is it is labeled by $i \in X_j$ or $i \notin X_j$ for each j and each i less than a value p. A node must have a label which is the projection of a solution on $[p]$ for some p. There is an edge between two nodes if one is a partial solution on $[p+1]$ and the other its restriction on $[p]$. The enumeration algorithm does a depth first traversal of this tree and outputs its leaves. To achieve polynomial delay, we must be able to decide in polynomial time whether some partial solution can be extended. Because the formula we want to satisfy only involves universal quantifier, it is enough to test whether it is true on the partial solution $[p]$ in time linear in p. In fact by an appropriate precomputation, we can go from a solution on $[p]$ to a solution on $[p+1]$ in constant time. Therefore the whole algorithm has a delay linear in n.

Remark that the enumeration procedure can be used for any formula ϕ but the problem of the possibility of extending a partial solution becomes much harder. In particular, we must be able to decide wether there is a solution to the formula.

4 Random Enumeration Algorithms

In this section, we try to explore the relationship between enumeration compexity as understood in this note and enumeration that is combinatorial counting of all kind of structures. Often those counting proofs yields ways of generating all structures of some size or satisfying some constraint uniformly or according to some known distribution. We show that it is then possible to enumerate them with a polynomial delay.

Definition 14. Let $\text{Enum}\cdot A \in \mathbf{EnumP}$, we say that an algorithm is a solution generator for $\text{Enum}\cdot A$, if given an input x it outputs an element of $A(x)$.

Definition 15. An *almost uniform random* solution generator is a randomized algorithm which generates each element in $A(x)$ with a probability greater than $\frac{1}{P(|x|)|A(x)|}$ where P is a polynomial.

Theorem 16. *If* $\text{Enum}\cdot A \in \mathbf{EnumP}$ *has a polytime almost uniform random solution generator, then* $\text{Enum}\cdot A$ *is in randomized* $\mathbf{IncP}_1$.

The proof of this theorem relies on a proper use of the coupon collector problem. We give a quick scketch here.

Proof. Assume we have a random solution generator for ENUM·A which works in time P_1, such that the output probability of each solution is greater than $\frac{1}{P_2(|x|)|A(x)|}$.

The algorithm to generate the solution is simple, it keeps generating solutions and add them to a self balanced tree to detect whether they have appeared before. It stops when no new solutions have been generated fo a while.

We have to define the criteria to stop the algorithm so that the algorithms finds all solutions with good probability and so that it is in randomized $\mathbf{IncP}_1$. Assume we have found n solutions and that there are at least $n+1$ solutions. The probability that we get k already generated solutions in a row is $(1 - \frac{1}{P_2(|x|)(n+1)})^k$. This probability is bounded by a constant if we set $k = nP_2(|x|)$. Now, we choose k to be $KnP_2(|x|)\log(n)\log(\epsilon^{-1})$ with K an integer such that the probability of generating k known solutions is bounded by $\frac{\epsilon}{10n^2}$. The probability the algorithm stops while there is still a solution to be found is bounded by $\sum_{i=1}^{\infty} \frac{\epsilon}{10i^2} \leq \epsilon$. Remark that the time the algorithm waits to stop depends on the number of already found solutions. Moreover remark that this time

This incremental delay algorithm can be turned into a polynomial delay one as in the deterministic case. We add a counter c that we increment each time the random generator outputs a new solution in the algorithm. Moreover, we store the solutions have been found in order and we to output them later. By definition, we know a bound on the number of solutions $|A(x)|$, thus we can bound $\log(|A(x)|))$ by $|x|^d$ for some integer d. Each time c has been increased by $K|x|^d P_2(|x|)\log(\epsilon^{-1})$, for some constant K we output the first solution stored in memory that has not been outputted. We must prove that with high probability, we never run out of solution to output during the algorithm and when it is the case, we say that algorithm stops (and thus fails). Again, we apply coupon collector. Assume we have generated less than n differents solutions amongst $n|x|^d P_2(|x|)\log(\epsilon^{-1})$ n consecutive outputs of the random generator. The worst case is when n is the number of solutions and we can bound the probability of the latter event by $\frac{\epsilon}{10n^2}$. Thus the probability that the whole algorithm run out of solutions is bounded by ϵ.

In conclusion, the described algorithm will output all solutions with a polynomial delay and with probability larger than $1 - \epsilon$. □

Remark 17. In the previous proof, the algorithm uses a space proportional

to the number of solutions to detect repetitions. It is not possible to avoid that since we have no control on the generator and therefore no control on the possible repetitions.

5 Polynomial Space

References

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

[2] Arnaud Durand and Yann Strozecki. Enumeration complexity of logical query problems with second-order variables. In *Proceedings of the 20th Conference on Computer Science Logic*, pages 189–202, 2011.

[3] R. Fagin. Generalized first-order spectra and polynomial-time recognizable sets. *American Mathematical Society*, pages 43–74, 1974.

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

[5] Mihai Patrascu and Ryan Williams. On the possibility of faster sat algorithms. In *SODA*, volume 10, pages 1065–1075. SIAM, 2010.

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

[7] Y. Strozecki. *Enumeration complexity and matroid decomposition*. PhD thesis, Université Paris Diderot - Paris 7, 2010.

[8] L.G. Valiant. The complexity of computing the permanent. *Theoretical computer science*, 8(2):189–201, 1979.