Enumerating models of DNF faster: breaking the dependency on the formula size

Florent Capelli, Yann Strozecki

Enumerating models of DNF faster: breaking the dependency on the formula size

Florent Capelli[1] and Yann Strozecki[2]

[1]Université de Lille, CRIStAL Laboratory, France, florent.capelli@univ-lille.fr
[2]Université de Versailles Saint-Quentin-en-Yvelines, DAVID Laboratory, France

October 9, 2018

Abstract

In this article, we study the problem of enumerating the models of DNF formulas. The aim is to provide enumeration algorithms with a delay that depends polynomially on the size of each model and not on the size of the formula. We succeed for two subclasses of DNF formulas: we provide a constant delay algorithm for k-DNF with fixed k by an appropriate amortization method and we give a polynomial delay algorithm for monotone formulas. We then focus on the *average delay* of enumeration algorithms and show that we can bring down the dependency of the average delay to the *square root* of the formula size and even to a logarithmic dependency for monotone formulas.

1 Introduction

An enumeration problem is the task of listing a set of elements without redundancies, usually corresponding to the solutions of a search problem, such as enumerating the spanning trees of a graph or the satisfying assignments of a formula. One way of measuring the complexity of an enumeration algorithm is the *total time* needed to compute all solutions. When the total time depends both on the *input and output*, an algorithm is called *output sensitive*. It is considered tractable and said to be *output polynomial* when it can be solved in polynomial time in the size of the input and the output.

Output sensitivity is relevant when *all* elements of a set must be generated, for instance to build a library of interesting objects to be studied by experts, as it is done in biology, chemistry or network analytics [5, 2, 6]. However, when the output is large with regard to the input, output polynomiality is not enough to capture tractability. Indeed, if one wants only a good solution or some statistic on the set of solutions, it can be interesting to generate only a fraction of the solutions. A good algorithm for this purpose must guarantee that the more time it has, the more solutions it generates. To measure the efficiency of such an algorithm, we need to compute the *delay* between two consecutive solutions. A first guarantee that we can expect is to have a good *average delay* (sometimes referred to as *amortized delay*), that is, to measure the total time divided by the number of solutions. There are many enumeration algorithms which are in constant amortized time or CAT, usually for the generation of combinatorial objects, such as the unrooted trees of a given size [26], the linear extensions of a partial order [19] or the integers by Gray code [14]. Uno also proposed [25] a general method to obtain constant amortized time algorithms, which can be applied, for instance, to find the matchings or the spanning trees of a graph.

However, when one wants to process a set in a streaming fashion such as the answers of a database query, it may be interesting to guarantee the regularity of the delay, usually by bounding it by a polynomial in the input size. We refer to such algorithms as *polynomial delay* algorithms. Many problems admit such algorithms, e.g. enumerate the cycles of a graph [20], the satisfying assignments of variants of SAT [8] or the spanning trees and connected induced subgraphs of a graph [3]. They are all based on similar techniques (see [16] for a survey).

For some problems, the size of the input may be much larger than the size of the generated solutions, which makes polynomial delay an unsatisfactory measure of efficiency. In that case, algorithms whose delay depends on the size of a single solution are naturally more interesting than polynomial delay or output polynomial algorithms. We say that an algorithm is in *strong polynomial delay* when the delay between two consecutive solutions is polynomial in the size of the last solution. To make this notion robust and more relevant, a precomputation step is allowed before the enumeration, in time polynomial in the input size, so that the algorithm can read the whole input and set up useful data structures. Observe that the notion of strong polynomial delay is also well suited for infinite enumeration where the size of the solutions grows arbitrarily [10].

In this paper, we focus on the notion of strong polynomial delay. While strong polynomial delay is a very desirable property for an algorithm it is rarely considered, since polynomial delay is often seen as the be-all end-all answer in enumeration. We feel that understanding the difference between polynomial delay and strong polynomial delay is a challenge that complexity of enumeration must address to be more relevant for practitioners. We are especially interested in finding such algorithms for the problem of enumerating the models of DNF formulas. Indeed, one can easily solve it with delay linear in the size of the formula (see Theorem 4) but it appears that removing the dependency in the number of terms in the delay is not easy, even when allowing exponential

Class	Delay	Space
DNF	$O(\mathsf{size}(D))$ (Theorem 4)	Polynomial
($\star$) DNF	$O(n\sqrt{m})$ average delay (Theorem 13)	Polynomial
($\star$) k-DNF	$2^{O(k)}$ (Theorem 5)	Polynomial
($\star$) Monotone DNF	$O(n^3)$ (Theorem 14)	Exponential
($\star$) Monotone DNF	$O(\log(n)\log(nm))$ average delay (Theorem 16)	Polynomial

Table 1: Overview of the results. In this table, D is a DNF, n its number of models and m its number of terms. New contributions are annotated with ($\star$).

space. DNF formulas have a simple structure as it is easy to find a large number of solutions or to approximate their number [15]. Nevertheless, exactly counting the solutions of a DNF formula is a canonical #P-complete problem. Enumerating the models of a DNF formulas thus appears to be a reasonable candidate to separate the notions of strong polynomial delay and polynomial delay. Surprisingly, we manage to remove the dependency of the delay in the number of terms for k-DNF and monotone DNF and to improve it for the general case. Our results are summarized in Table 1.

There are few examples of strong polynomial delay in the literature. *Constant delay* algorithms naturally fall into this category and a whole line of research is dedicated to design such algorithm for enumerating models of first order queries for restricted classes of structures [21] (see also the survey [22]). However, while these algorithms are called constant delay because their delay does not depend on the database size, it often depends more than exponentially on the size of the solutions. Other examples naturally arise from logic such as the enumeration of assignments of MSO queries over trees or bounded width graphs [4, 7]. Recently, Amarilli et al. [1] presented an algorithm to enumerate the solutions of a restricted class of Boolean circuits known as structured d-DNNF used in knowledge compilation with delay linear in the number of variables and hence independent of the size of the circuit.

The paper is organized as follows. We first introduce basic notions on formulas and enumeration complexity then present tries and Gray code in Section 2. We show in Sec 3 how to adapt three classical methods to generate the models of a DNF, the best having a linear delay in the formula size. In Section 4, we give a backtrack search algorithm, using a good amortization, which is in constant delay for k-DNF. In Section 5, we give another backtrack search algorithm, whose average delay is better than linear. Finally, in Section 6, we provide a strong polynomial delay for monotone DNF formulas but with an exponential memory and we specialize and adapt the algorithm of the previous section to obtain a logarithmic delay for monotone DNF formulas. Proposition with omitted proofs due to space restrictions are labeled with ($\star$) and a proof is given in the appendix.

2 Definitions and notations

Terms and DNF-formulas. Let X be a set of variables and let n and be the size of X. We fix some arbitrary order on X and write $X = \{x_1, \ldots, x_n\}$. A *literal* ℓ is either a variable $x \in X$ or the negation of a variable $\neg x$ for some $x \in X$. A *term* C is a finite set of literals such that every two literals in C have a different variable. A *Disjunctive Normal Form formula*, DNF for short, is a finite set of terms. Given a literal ℓ, we denote its underlying variable by $\mathsf{var}(\ell)$. We extend this notation to terms by denoting $\mathsf{var}(C) := \bigcup_{\ell \in C} \mathsf{var}(\ell)$ for a term C and to DNF by denoting $\mathsf{var}(D) := \bigcup_{C \in D} \mathsf{var}(C)$ for a DNF D.

Given an assignment $\alpha : X \to \{0,1\}$, we naturally extend α to literals by defining $\alpha(\neg x) = 1 - \alpha(x)$. An assignment α satisfies a term C if for every $\ell \in C$, $\alpha(C) = 1$. A *model* α is an assignment that satisfies a DNF D, that is, there exists $C \in D$ such that α satisfies C. We write $\alpha \models D$ if α is a model of D. It is easy to see that given a term C, there exists a unique assignment of variables in $\mathsf{var}(C)$ satisfying C. We denote this assignment by $\mathbf{1}_C$.

Given a DNF D on variables X, we denote by $\mathsf{sat}(D) = \{\alpha \mid \alpha \models D\}$ the set of models of D. Let $Y \subseteq X$, $\tau : Y \to \{0,1\}$ and $\sigma : X \to \{0,1\}$, we say that σ is *compatible* with τ, denoted by $\sigma \simeq \tau$, if the restriction of σ to Y is equal to τ. We denote by $\mathsf{sat}(D,\tau) = \{\alpha \mid \alpha \models D, \sigma \simeq \tau\}$ the set of models of D compatible with τ. We also denote by $D[\tau]$ the DNF defined as follows: we remove every term C from D such that there exists a literal $\ell \in C$ such that $\tau(\ell) = 0$. For the remaining terms, we remove every literal whose variable is in Y. Observe that since we consider DNF to be sets of terms, by definition, $D[\tau]$ has no duplicated terms. It is clear that: $\mathsf{sat}(D,\tau) = \{\tau \cup \alpha \mid \alpha \models D[\tau]\}$. The *size* of a DNF D is denoted by $\|D\|$ and is equal to $\sum_{C \in D} |C|$.

Enumeration complexity. Let Σ be a finite alphabet and Σ^* be the set of finite words built on Σ. Let $A \subseteq \Sigma^* \times \Sigma^*$ be a binary predicate, we write $A(x)$ for the set of y such that $A(x,y)$ holds. The enumeration problem Enum·A is the function which associates $A(x)$ to x.

The computational model is the random access machine model (RAM) with addition, subtraction and multiplication as its basic arithmetic operations and an operation $\texttt{Output}(i,j)$ which outputs the concatenation of the values of registers $R_i, R_{i+1}, \ldots, R_j$. We assume that all operations are in constant time except the arithmetic instructions which are in time linear in the size of their inputs. While this is arguably not a real-life scenario, this allows us to formally write algorithms with constant time delay while it is theoretically impossible if the time needed to write the output is taken into account. A RAM machine solves Enum·A if, on every input $x \in \Sigma^*$, it produces a sequence $y_1, \ldots, y_n$ such that $A(x) = \{y_1, \ldots, y_n\}$ and for all $i \neq j$, $y_i \neq y_j$. The space used by the machine at a given step is the sum of the number of bits required to store the integers in its registers.

The delay of a RAM machine which outputs the sequence $\{y_1, \ldots, y_n\}$ is the maximum over all $i \leq n$ of the time the machine uses between the generation of y_i and y_{i+1}. Note that we allow a *precomputation phase* before the machine starts to enumerate solutions, which can be in time polynomial in the size of the input.

Trie. A *trie* is a data structure used to represent a set of words on an alphabet M which supports efficient insertion and deletion. We refer the reader to [12, 14] for more details. We use them to either store a formula, that is the sets of its terms, or to represent a set of assignments of a formula.

The trie is a M-ary tree, each of its inner nodes is labeled by a letter of the alphabet. A leaf represents a word: the sequence of labels traversed when going from the root to the leaf. The trie represents the set of words represented by its leaves. The children of a node are given by a linked list.

A model α of the DNF is represented by the sequence of its values: $\alpha(x_1)\alpha(x_2)\ldots\alpha(x_n)$. The trie we use to store these models are binary trees, since the labels are in $\{0,1\}$. We represent a term by the sequence in ascending orders of its literals, with for all i, $x_i < \bar{x}_{i+1} < x_{i+1}$. The list of children of a node in the trie is maintained in this order.

We need to search, insert and suppress elements in a trie. Those three operations are in $O(n)$ for the two tries we have defined, where n is the number of variables.

Gray Code. Gray codes are an efficient way to enumerate the integers between 0 and $2^n - 1$ written in binary or equivalently the subsets of a set of size n. They enjoy two important properties: the Hamming distance of two elements in the Gray enumeration order is one and each new element is produced in constant time using only additional $O(\log(n)^2)$ space (see [13]). In other words, this enumeration algorithm has constant delay. We can generate all models of a term in constant delay, using Gray code and an additional array which contains the indexes of the free variables of the term.

Proposition 1 ($(\star)$)**.** *The models of a term C on variables X can be enumerated in constant delay.*

3 Classical enumeration algorithms

In this section we present three generic enumeration methods applied to the generation of the models of a DNF formula. The best one has a linear delay in the size of the instance, and we study in later sections several restrictions to obtain a delay polynomial in the size of a solution.

3.1 Union of terms

The first two methods are based on the fact that the models of a DNF formula are the union of the models of its terms. The only problem is to avoid repetitions of models. We first use a method to enumerate the union of sets of elements which preserves polynomial delay (see [23]). It relies on a priority rule between the sets to avoid repetitions, as recalled in the proof of the following proposition.

Proposition 2 ($(\star)$ Adapted from proposition 2.38 and 2.40 in [23])**.** *The models of a DNF formula D with m terms can be enumerated with delay $O(m\|D\|)$.*

By improving the way we test whether a model of a term is the model of any term of larger index, we can drop the delay to $O(m^2)$. In fact, by generating the solutions of each term in the same order, we can avoid completely the redundancy test and get a better delay. The following algorithm merges several ordered arrays which are generated dynamically by enumeration procedures.

Proposition 3 ($(\star)$ Adapted from proposition 2.41 in [23])**.** *The models of a DNF formula D with m terms and n variables can be enumerated with delay $O(mn)$.*

The average delay of the two previous algorithms is lower than their delays. Let r be the average number of times a solution is produced during the algorithm, we can replace m by r in the average delay of the previous algorithm. It is possible to prove that r is smaller than m by studying how terms share models, but the complexity gains are very small and we give an algorithm with a much better average delay in Section 5.

3.2 Flashlight method

We present a classical enumeration method called the *Backtrack Search* or sometimes the *Flashlight Method* used in many previous articles [20, 24] in particular to solve auto-reducible problems. We describe the method in the context of the generation of the models of D a DNF formula.

We define a tree T_D whose nodes are the assignments τ over variables $\{x_1, \dots x_k\}$ such that there is a model σ of D which is compatible with τ. The children of a node labeled by τ are the partial assignments τ' defined over $\{x_1, \dots x_{k+1}\}$ which are compatible with τ.

The leaves of T_D are the models we want to generate, therefore a depth first traversal visits all leaves and thus outputs all solutions. Since a path from the root of the tree is of size n, it is enough to be able to find the children of a node in polynomial time to obtain a polynomial delay. Hence the Flashlight Method has a polynomial delay if and only if the following extension problem is in P: given τ over $\{x_1, \dots x_k\}$ is there σ a model of D compatible with τ?

The extension problem for a DNF is very simple to solve: compute the formula $D[\tau]$ and decide whether it is satisfiable in time $O(\|D\|)$. This yields an enumeration algorithm with delay $O(n\|D\|)$. The delay can be improved by using the fact that we solve the extension problem several times on very similar instance as it has been done for other problems [18, 17] such as the enumeration of the models of a monotone CNF.

Proposition 4. *The models of a DNF formula D can be enumerated with delay $O(\|D\|)$.*

Proof. We use the previous algorithm which does a depth first search in T_D. When it visits the node τ, we need to decide whether $D[\tau]$ is satisfiable which is equivalent to testing whether it has a non falsified term. To speed-up the flashlight search, we use a data structure to decide quickly this problem and we need to guarantee that it can be updated fast enough when the tree is traversed.

For each term C, we store an integer f_C which represents how many literals of the term are falsified by the current partial assignment. A term C is valid when $f_C = 0$. For each literal l, we store the list of terms which contain the literal. We also store an integer vc for valid terms, which counts how many terms are not falsified by the current partial assignment.

At the beginning of the algorithm, all f_C are set to 0 and $vc = m$. Then visiting a child τ' of τ corresponds to choosing the value of some variable x_k. If x_k is set to 0 then for each term C containing x_k, f_C is incremented. The number of terms such that f_C is changed from 0 to 1 is subtracted to vc. If x_k is set to 1, then the same is done for $\bar{x_k}$. The fact that $D[\tau']$ is satisfiable is equivalent to $vc > 0$. Remark that going up the tree when backtracking works exactly as going down, but the variables f_c are decremented and vc is incremented instead.

As a consequence, the complexity of the algorithm over a path in the tree is $O(\|D\|)$ since for each term C, the variable f_C will be modified $|C|$ times. When the algorithm goes down the tree it may first set x_k to 0 and fails, but then it sets x_k to 1 and goes down. Hence the cost of going down to a leaf is at most twice the cost of following the path to the leaf. Since between two outputted solutions, the algorithm follows one path up and one down, the delay is in $O(\|D\|)$. □

This last algorithm has a delay linear in the size of the formula, however the size of the formula can be extremely large with regards to the size of a model. In the next sections we will try reduce or eliminate the dependency of the delay in the size of the input either for particular DNF formulas or by relaxing the notion of delay.

4 Enumerating models of k-DNF

A term C is a k-term if and only if $|C| \leq k$. A DNF is a k-DNF if all its terms are k-terms. In this section, we present an algorithm to enumerate the models of a k-DNF with a $2^{O(k)}$ delay. The idea is to select a k-term and use its 2^{n-k} models to amortize more costly operations. More precisely, we prove the following:

Theorem 5. *The models of a k-DNF with n variables can be enumerated with precomputation in $O(n)$ and $O(k^2 2^{5k})$ delay.*

To explain our algorithm, we need first to introduce notations. Let $H(p) = -p\log(p) - (1-p)\log(1-p)$ be the binary entropy. From the following classical inequality (see [11]):

$$\sum_{i=0}^{k} \binom{n}{i} \leq 2^{nH(k/n)},$$

we get for $k \leq n/2$:

Lemma 6. *The number of k-terms on n variables is at most $2^{nH(k/n)+k}$.*

Let D be a DNF-formula on variables X. Assume wlog that X is ordered with $<$. Given a term $C \in D$, we denote by $\mathbf{1}_C : \mathsf{var}(C) \to \{0,1\}$ the only model of C, that is, for every $\ell \in C$, $\mathbf{1}_C(x) = 1$ if $\ell = x$ and $\mathbf{1}_C(x) = 0$ if $\ell = \neg x$.

If $y \in \mathsf{var}(C)$, we denote by $\mathbf{0}^y_C : \{z \in \mathsf{var}(C) \mid z \leq y\} \to \{0,1\}$ the assignment defined by $\mathbf{0}^y_C(z) = \mathbf{1}_C(z)$ for $z < y$ and $z \in \mathsf{var}(C)$ and $\mathbf{0}^y_C(y) = 1 - \mathbf{1}_C(y)$.

For example, if $C = x_1 \wedge x_2 \wedge \neg x_3$, we have $\mathbf{1}_C = \{x_1 \mapsto 1, x_2 \mapsto 1, x_3 \mapsto 0\}$ and $\mathbf{0}^{x_1}_C = \{x_1 \mapsto 0\}$, $\mathbf{0}^{x_2}_C = \{x_1 \mapsto 1, x_2 \mapsto 0\}$ and $\mathbf{0}^{x_3}_C = \{x_1 \mapsto 1, x_2 \mapsto 1, x_3 \mapsto 1\}$.

These assignments naturally induce a partitioning of the model of a DNF:

Lemma 7. *Given a DNF D and a term $C \in D$, we have:*

$$\mathsf{sat}(D) = \mathsf{sat}(D, \mathbf{1}_C) \uplus \biguplus_{y \in \mathsf{var}(C)} \mathsf{sat}(D, \mathbf{0}^y_C).$$

Proof. The right-to-left inclusion is clear as $\mathsf{sat}(D,\tau) \subseteq \mathsf{sat}(D)$ for any τ. Moreover, these unions are clearly disjoint since for every $y, z \in \mathsf{var}(C)$, $y < z$, we have $\mathbf{0}^z_C(y) \neq \mathbf{0}^y_C(y)$ and $\mathbf{1}_C(y) \neq \mathbf{0}^y_C(y)$.

For the left-to-right inclusion, let $\tau \in \mathsf{sat}(D)$. If $\tau \simeq \mathbf{1}_C$, then $\tau \in \mathsf{sat}(D, \mathbf{1}_C)$. Otherwise, let y be the smallest variable of $\mathsf{var}(C)$ such that $\tau(y) \neq \mathbf{1}_C(y)$. Then we have $\tau \simeq \mathbf{0}^y_C$. □

Proof (of Theorem 5). Given a k-DNF D on variables X, we use Lemma 7 to enumerate $\mathsf{sat}(D)$. We denote by $X = \mathsf{var}(D)$, $N = |X|$ and $M = |D|$.

We start by picking a k-term $C \in D$. It is easy to see that for every $\tau : X \setminus \mathsf{var}(C) \to \{0,1\}$, we have $\tau \cup \mathbf{1}_C \models D$. Thus, enumerating $\mathsf{sat}(D, \mathbf{1}_C)$ can be done very efficiently with a $O(1)$ delay by simply enumerating $2^{X \setminus \mathsf{var}(C)}$ using a Gray code. Observe that the precomputation time boils down to choosing a term and outputting its first solution which can obviously be done in time $O(n)$ as stated in the theorem.

Between the output of two solutions of $\mathsf{sat}(D, \mathbf{1}_C)$, we spend some time to precompute $D[\mathbf{0}^y_C]$ for every $y \in \mathsf{var}(C)$. Since $|\mathsf{var}(C)| \leq k$, we have to compute at most k such DNF. Moreover, to compute each one of them, we have to inspect every term of D and every variable of these terms. Since each term has at most k variables, we need $O(kM)$ steps to precompute all terms in $D[\mathbf{0}^y_C]$ for one $y \in \mathsf{var}(C)$. We then remove all copies but one of the same terms. This can be done in time $O(kM^2)$ by comparing the terms pairwise. In the end, we need $O(k^2M^2)$ steps to precompute $D[\mathbf{0}^y_C]$ for every $y \in \mathsf{var}(C)$. Let A be a constant such that this precomputation can be done in time $A \cdot k^2M^2$ steps at most.

Assume that between the output of two solutions of $\mathsf{sat}(D, \mathbf{1}_C)$, we allow dA steps for this precomputation (the value of d will be fixed later depending on our needs). Since $|\mathsf{sat}(D, \mathbf{1}_C)| \geq 2^{N-k}$,

this gives us a total amount of $2^{N-k}dA$ steps for this precomputation. Thus, if $2^{N-k}dA > Ak^2M^2$, that is, if

$$2^{N-k}d > k^2M^2 \tag{1}$$

we have enough time to compute $D[\mathbf{0}^y_C]$ for every $y \in \mathsf{var}(C)$. If this is the case, then we do the precomputation, finish the enumeration of $\mathsf{sat}(D, \mathbf{1}_C)$ and then recursively start the enumeration of $\mathbf{0}^y_C \times D[\mathbf{0}^y_C]$ for each $y \in \mathsf{var}(C)$. The number n of variables of $D[\mathbf{0}^y_C]$ has of course decreased but we can still allow for dA steps of extra computation between the output of two solutions.

It is clear that as long as (1) is true, we can output solutions of D with delay dA. We may however have a problem if, at some point in the recursion, the DNF D' we are using to compute solutions of D may have too few variables making (1) false. We are going to pick d sufficiently large so it never happens.

Since we only consider k-DNF, we know by Lemma 6 that at each step of the recursion with n unassigned variables, the number m of terms verifies $m \leq M_0 := 2^{nH(k/n)+k}$. Our goal is thus to find d such that for every $n \geq 2k$:

$$2^{n-k}d \geq k^2M_0^2$$

By rearranging the terms so that k is in factor in the exponent and denoting $x = k/n$, we conclude that if the following holds then (1) is true when $k \leq n/2$:

$$d \geq k^2 2^{k((2H(x)-1)/x+3)} \tag{2}$$

It is easy to compute the maximum of the function $(2H(x) - 1)/x + 3$ over $[0, 1/2]$. We obtain a bound of 5 at $x = 1/2$. Hence for all $k \leq n/2$, if $d \geq k^2 2^{5k}$ then (1) is true at each recursive call. When $n < 2k$, we enumerate the models using the algorithm of Theorem 4. It gives a delay of $\|D\| \leq k2^{3k}$ since by applying Lemma 6 for $n = 2k$, we have that that the number of k-terms with $2k$ variables is at most $2^{2kH(1/2)+k} = 2^{3k}$. Thus, a k-DNF with less than $2k$ variables has size at most $k2^{3k}$. □

The $O(k^2 2^{5k})$ delay of the previous algorithm could be improved to $O(k^2 2^{3k})$ by using a more clever algorithm to detect duplicated terms when computing $D[\mathbf{0}^y_C]$. This operation can easily be done in time $O(k^2 M \log(M))$ by sorting the terms before eliminating duplicates and even in time $O(k^2 M)$ by using a trie with relevant pointers allowing to test if a term is already in the trie in time $O(k)$. Both solutions would make the presentation of the proof heavier so we chose to present this simplified version since the delay is also $2^{O(k)}$.

Theorem 5 gives an algorithm to enumerate the models of a k-DNF with delay $d = 2^{O(k)}$ and $O(n)$ precomputation. A pseudo code for this algorithm is given in the appendix (Algorithm 1). It has *constant delay* for constant k and it is in strong polynomial delay (polynomial in n the size of a solution) for terms of size $O(\log(n))$. We conjecture however that the models of a general DNF cannot be enumerated with strong polynomial delay.

We can use Algorithm 1 to significantly improve a result on the enumeration of models of first order formula with free second order variables. In [9], Theorem 10, it is proved that the models of a Σ_1 formula (a single block of existential quantifiers followed by a quantifier free formula) with free second order variables can be enumerated in polynomial delay using a method similar to Proposition 2. Moreover, this problem is shown to be equivalent to the enumeration of models of a k-DNF. As a corollary of Theorem 5, the enumeration of models of a Σ_1 formula can be done in constant delay. It is very surprising since it is the same complexity as for Σ_0 (quantifier free formulas), the first level of the hierarchy.

5 Average delay of enumerating the models of DNF

In this section, we analyze the average delay of the Flashlight method, using appropriate data structures, to show it is better than the delay. The idea is to amortize the cost of maintaining the formula $D[\tau]$ during the traversal of the tree over all models of $D[\tau]$ in the spirit of [25]. To do that we exhibit a relation between the number of models of a DNF and its number of terms. For any class of DNF for which it is possible to guarantee, for all partial assignments τ, that the number of models of $D[\tau]$ is large enough with regard to $|D[\tau]|$, we obtain a good average delay. We then improve the delay of the Flashlight method by using a different branching (in the spirit of Algorithm 1) which guarantees that the number of terms decreases sufficiently but not too much in some branch which allows us to amortize the cost of branching.

The following lemma shows that DNF whose terms share a model have many models. It applies directly to any monotone formula, since the all one assignment is always a model of a (positive) monotone term.

Lemma 8. *A DNF formula D with m distinct terms which have a common model has at least m models.*

Proof. Let α be a model of the m terms of the DNF formula D and x be a variable of D. We claim that x appears only negatively or positively in D. Indeed, assume w.l.o.g that that $\alpha(x) = 1$ and that $x \in C_1$ and $\neg x \in C_2$. Thus $\alpha \not\models C_2$ which contradicts the definition of α. We construct a monotone DNF D' by replacing every variable that appears only negatively in D by positive occurrences. There is an obvious one-to-one correspondence between the models of D' and the models of D. Now, for $C \in D'$, let m_C be the model of C defined as $m_C(x) = 1$ if and only if $x \in \mathsf{var}(C)$. Since D' is monotone, it is clear that $m_C = m_{C'}$ if and only if $C = C'$, which gives m distinct models for D' and thus for D. □

Using the previous lemma, we prove that a DNF has at most a number of terms which is quadratic in its number of models.

Lemma 9. *A DNF formula with m non empty distinct terms has at least $m^{1/2}$ models.*

Proof. Let α be a model of the DNF D, we denote by r_α the number of terms which are compatible with α: $|\{C \models \alpha \mid C \in D\}|$. It corresponds to the number of redundancies of α, that is the number of terms which has it as model (and the number times it is generated in the algorithm of Proposition 3).

Now, let us denote by A the average of r_α over all models. By definition, we have

$$|\mathsf{sat}(D)| = \frac{\sum_{C \in D} |\mathsf{sat}(C)|}{A}.$$

Since A is an average, there is an α such that $r_\alpha \geq A$. It means that at least A distinct terms are compatible with α and then by Lemma 8 these terms have at least A distinct models. Hence we have $|\mathsf{sat}(D)| \geq A$. Now, multiplying both equations and observing the fact that $|\mathsf{sat}(C)| \geq 1$ for every $C \in D$, we obtain $|\mathsf{sat}(D)|^2 \geq \sum_{C \in D} |\mathsf{sat}(C)| \geq m$. □

In the previous proof, the inequalities are not tight and we may certainly improve the result. In particular, we can avoid to bound $\sum_{C \in D} |\mathsf{sat}(C)|$ by m. We then get a lower bound on the number of models of $m^{1/2+c/\log(n)}$ for some constant c. It improves the delay of the next algorithm only for

superpolynomial m and require some additional computations, hence we chose to state Lemma 9 with the simpler $\sqrt{m}$ bound.

The quality of the algorithms presented in this section is directly connected to this relation between the number of terms and the number of solutions of a DNF. Any improvement would have direct consequences on the complexity of enumerating the models of a DNF.

Question 10. *What is the best k such that we can guarantee that a DNF with m terms has at least m^k solutions? We are not aware of any results concerning this fact. We only know that $1/2 \leq k \leq \log_3(2)$. The lower bound is Lemma 9 and the upper bound easily follows from the DNF having all terms of size at most n on n variables has 2^n models and 3^n terms.*

To improve the average delay of the flashlight method, we need to use an adapted data structure. In particular we need that, when considering some inner node $D[\tau]$ of T_D, the cost to process it is in $O(|D[\tau]|)$ and not in $O(m)$. In particular, we need to guarantee that there are no redundancy of terms in the structure representing $D[\tau]$ and that we can maintain it efficiently, that is why we use a trie.

Theorem 11. *The models of a DNF can be enumerated with average delay $O(n^2\sqrt{m})$ and polynomial space.*

Proof. We maintain the formula $D[\tau]$ when we traverse T_D using the trie containing its terms as explained in Section 2. On a node τ we can decide quickly whether $D[\tau]$ has a model and we can maintain $D[\tau]$ efficiently without redundancy of terms. In the flashlight search, we will fix the variables following their order $x_1, \ldots, x_k$. Hence, visiting a child τ' of τ corresponds to setting the value of some variable x_k with all variables x_i with $i < k$ already fixed. If we set x_k to 0 then we need to remove the subtree under the root of the trie, with first node x_k. Then we remove the subtree under the root of the trie, with first node $\bar{x}_k$, and insert back all elements in this subtree into the trie without the first node $\bar{x}_k$. The complexity of the latter is in $O(|D[\tau]|n)$ since the number of terms is bounded by $|D[\tau]|$ (no term appear several times in the trie). To set x_k to 1, we do the same operation where we exchange the roles of x_k and $\bar{x}_k$. To be able to go up in the tree during the flashlight search, we must restore the trie to its previous state. To do that in time $O(|D[\tau]|n)$, it is enough to store the list of elements which have been removed or added in the trie when going down the same edge and to reverse the operations. The additional memory used during the algorithm is bounded by $O(mn^2)$.

We now compute the average delay, that is the total time of the algorithm divided by the number of outputted models. To do that we distribute the time spent on each inner node $D[\tau]$ to the models in the leaves of the subtree rooted at $D[\tau]$. As we have explained, the time spent on the node $D[\tau]$ (to do the branching on some variable x_k) is in $O(|D[\tau]|n)$. By Lemma 9, we have that $D[\tau]$ has at least $\sqrt{|D[\tau]|}$ models. The time spent in $D[\tau]$ is distributed uniformly over all leaves of the tree rooted at $D[\tau]$. Hence each leaf receive at most $n\sqrt{|D[\tau]|} = n|D[\tau]|/\sqrt{|D[\tau]|}$.

A leaf receives a cost of at most $n\sqrt{m}$ for each of its n ancestors, hence the time per leaf (or the average delay) is bounded by $O(n^2\sqrt{m})$. □

We now improve the way we choose the next variable on which we branch in order to improve the delay. The idea is to branch on variables so that the recursion tree kept as balanced as possible. We use the following lemma:

Lemma 12 ($(\star)$)**.** *Let D be a DNF formula with m terms. Let m_0 and m_1 be the number of terms of $D[x \to 0]$ and $D[x \to 1]$ respectively. Then $m_0 + m_1 \geq m/2$.*

Theorem 13. *The models of a DNF can be enumerated with average delay $O(n\sqrt{m})$ and polynomial space.*

Proof. Instead of branching on a variable x_1 in the flashlight search, the branching will have the structure of a potentially large comb, similar to the one used in Algorithm 1. We define a sequence of partial assignments α_i, α'_i with $0 \leq i \leq n$. The assignments α_i and α'_i are defined over the variables $\{x_1, \ldots, x_i\}$ and they are compatible with α_i. The assignment α_0 is the empty assignment, α_i is the extension of α_{i-1} by $x_i \to \epsilon(x_i)$ and α'_i is the extension by $x_i \to 1 - \epsilon(x_i)$. We write $m_i = |D[\alpha_i]|$| and $m'_i = |D[\alpha'_i]|$, these numbers are the number of distinct terms in $D[\alpha_i]$ and $D[\alpha'_i]$ respectively. Let us now define the assignment ϵ, given $D[\alpha_{i-1}]$, $\epsilon(x_i)$ is chosen to maximize m_i.

By Lemma 12, we have that $m_i \geq m_{i-1}/4$ that is the number of terms is divided by at most 4 at each step. Since $m_0 = m$ and $m_n \leq 1$, then there is a i_0 such that $m_{i_0} \in [m/8, m/2]$.

We use the same method as in Theorem 11 to maintain the trie representing $D[\tau]$. During the successive construction of $D[\alpha_i]$ and $D[\alpha'_i]$, each term of D will be removed or inserted twice at most in the trie. Hence the total cost of this branching and of the computation of the m_i and m'_i is in $O(mn)$, that is as fast as a binary branching. We assign this cost to the solutions in the branch α_{i_0}. By construction, $D[\alpha_{i_0}]$ has more than $m/4$ terms, hence it has by Lemma 9 at least $\sqrt{m/8}$ models. Therefore the cost charged to the leaves is $n\sqrt{m}$.

Now let us analyze how much a leaf is charged. Remark that in the path from the root to the a leaf, not all nodes charge the leaf. By construction the nodes which charge the leaf have less than half the terms of the previous node which has charged the leaf. Hence the complexity charged to a node is bounded by $\sum_i n\sqrt{m/2^i} = n\sqrt{m}\sum_i 1/2^{i/2}$. Since $\sum_i 1/2^{i/2}$ is bounded by a constant, the cost of a leaf and thus the average delay is $O(n\sqrt{m})$. □

6 Enumerating models of monotone DNF

When the underlying formula is monotone, that is it does not contain any negated literal, we can enumerate the models with a delay polynomial in the number of variables only. However, our current techniques need an exponential memory to work.

Theorem 14. *There is an algorithm that given a monotone DNF with n variables and m terms, enumerate the models of D with preprocessing $O(nm^2)$ and delay $O(n^3)$. The space needed for this algorithm is linear in the number of solutions of D.*

Proof. We start by removing from D every term C' such that there exists $C \in D$ with $C \subseteq C'$. Observe that it does not change the models of C since $C' \Rightarrow C$. This preprocessing phase takes $O(nm^2)$ since comparing two terms may be done in time $O(n)$. Let D' be the resulting minimized monotone DNF.

The algorithm then work as follows: arbitrarily order the terms of $D' = \{C_1, \ldots, C_p\}$ and its variables $X = \{x_1, \ldots, x_n\}$. We initialize a trie T that will contain the solutions that we have already enumerated, that is, in the following algorithm, each time we output a solution, we store it in T, so we can check in time $O(n)$ if a solution has already been enumerated.

We now enumerate the solutions as follows: we start by enumerating all solutions of C_1. Then, we proceed by induction: once we have enumerated all solutions of $D'_i = C_1 \vee \cdots \vee C_i$, we enumerate all solutions of C_{i+1} that are not solutions of D'_i until $i = p$. Once we are done, we have, by induction, enumerated all models of D.

We claim that we can do this with delay $O(n^3)$ using a classical reverse search method. Let $Y = (y_1, \ldots, y_m)$ be the variables that are not in $\mathsf{var}(C_i)$, ordered following the natural order we have on X. The solutions of C_i are in one-to-one correspondence with 2^Y as follows: given $S \subseteq Y$, we have a solution m_S defined as $m_S(x) = 1$ if $x \in \mathsf{var}(C_i) \cup S$ and $m(x) = 0$ otherwise. We explore the solution of C_i by following a tree A whose nodes are labeled with m_S for every $S \subseteq Y$. The root of the tree is label-led by $m_\emptyset$ and for every S, the unique predecessor of node m_S is $m_{S'}$ with $S' = S \setminus \max(S)$. In other words, given S, the successors of m_S are $m_{S \cup \{x_k\}}$ for $k > \max(S)$.

We enumerate the solution by following the structure of A. We start from the root of A, that is, we enumerate $m_\emptyset$. We claim that $m_\emptyset$ has not yet been enumerated. Indeed, assume toward a contradiction that $m_\emptyset \models C_j$ for $j < i$. Then since D' is monotone, it means that $C_j \subseteq C_i$ which is absurd since D' is minimized. Thus, the first model we enumerate is guaranteed to be fresh.

Now we follow the structure of A by depth first search. When we visit a node of A label-led with m_S, we start by checking in the trie if m_S has already been enumerated. It can be done in $O(n)$. If m_S has not been enumerated, then we output it and keep on going down in the tree A. If m_S has been enumerated then we can discard the whole subtree of A rooted in m_S. Indeed, by definition, every solution $m_{S'}$ in this subtree verifies $S' \supseteq S$. Thus, if m_S has been enumerated then $m_S \models D'_{i-1}$ and since D'_{i-1} is monotone, $m_{S'} \models D'_{i-1}$, so $m_{S'}$ has already been enumerated.

Thus, if m_S has already been enumerated, we backtrack in the tree to the next node in A that has not yet been explored and that is not in the discarded subtree. This can be done in $O(1)$ if we maintain in each visited node a pointer to the first ancestor that has an unexplored child.

We claim that we will visit at most n^2 nodes of A before finding a new solution or realizing that we have enumerated all solutions of D'_i and should proceed with C_{i+1}. Indeed, by the previous argument, we only have to check the maximal unexplored nodes of A since if a solution has already been enumerated, so have all its descendants. Now, each node of A has at most n children and A is of depth n. Hence, during the depth-first search, we will have at most n^2 such maximal nodes (at most n per level). Since checking if a solution is already in the trie can be done in $O(n)$, our algorithm will have delay at most $O(n^3)$. □

By using a better data structure to store the solutions and another organization of the models of a term we should bring the delay down to $O(n^2)$ or better, but we leave that for further researchs. Moreover, observe that, using the reduction in the proof of Lemma 8 to the monotone case, the algorithm described in Theorem 14 also works if every variable appears only positively or negatively in all terms of the formula.

We now consider algorithms with a good average delay for enumerating the models of a monotone DNF formula. This allows to obtain a better delay that the previous theorem, while only using a polynomial space. First, recall that monotone DNF formulas have at least as many models as terms because of Lemma 8. Hence, using the algorithm of Theorem 13 on a monotone formula, we obtain the following theorem.

Theorem 15. *The models of a monotone DNF can be enumerated with average delay $O(n)$ and polynomial space.*

To improve the bound on the average delay using a similar algorithm, we should either guarantee a better relationship between the number of terms and solution or we should reduce the complexity of maintaining the trie during the algorithm. Note that the formula with all positive terms has 2^n models but also 2^n terms. If we further assume that no term are redundant, that is there are no C_1, C_2 such that $C_1 \subseteq C_2$, then the formula with all terms of size $n-1$ has $n+1$ models and n

terms. Even when m is large, the relationship is almost linear: the formula with all terms of size $n/2$ has $2^n/2$ models and $O(2^n/\sqrt{n})$ terms. Hence, to improve the average delay, it seems hard to rely on a better bound on the number of solutions. However, the cost to deal with the trie can be reduced, as long as m is not too large as shown in the next theorem.

Theorem 16. *The models of a monotone DNF can be enumerated with average delay* $O(\log(n)(\log(m)+\log(n))$ *and polynomial space.*

Proof. First remark that when a term contains $n-k$ variables, then it has 2^k models. Hence, in the algorithm of Theorem 13, when we compute the cost distributed over all solutions of $D[\tau]$, we can make the following adjustment. We consider two cases. First say that $D[\tau]$ has n_τ variables and assume there is a term with $n_\tau - k$ variables such that $k > \log(|D[\tau]|) + 2\log(n)$. Then $D[\tau]$ has at least 2^k models which is larger than $|D[\tau]|n^2$. As a consequence, the complexity which is charged to the leaves is only $1/n$ and the total cost charged to a leaf by branching steps of this king is bounded by 1.

Now assume that for some partial assignment τ, all terms in $D[\tau]$ contain more than $n_\tau - k$ literals, with $k = \log(|D[\tau]|) + 2\log(n)$. Then we change the encoding of the formula: each term of D is represented by its complementary. Since the formula is monotone, we need only to store

The complementary terms are stored in a trie where the children of a node are organized in an AVL tree instead of an ordered list. All operation can be done on this trie in time $O(\log(n) * k)$, where k is the size of the object to insert. The algorithm works as the one in Theorem 13 on this new data structure with the following difference in branching. When x_i is set to 1, we remove the subtree under the roof with first node x_i and add back all elements without x_i. When x_i is set to 0, we keep only the subtree under the roof with first node x_i and remove the rest.

The cost of a step is now bounded by $O(|D[\tau]|k\log(n))$ and the cost charged to the leaf is bounded by $k\log(n)$, that is by $(\log(m) + 2\log(n))\log(n)$ which proves the proposition. □

A better data structure can be used in the previous theorem: a trie where the children of a node are stored in an array of size n. Then the operations are in time $O(k)$ if we accept that we have an arbitrary supply of initialized memory. The average delay then drop to $O(\log(n) + \log(m))$ and we conjecture that there is a good data structure allowing such delay without needing initialized memory.

The result on monotone DNFs can easily be transferred to a problem studied in [17]: the generation of all unions of given subsets. An instance $\{s_1, \ldots, s_m\}$ is a set of m subsets of $[1, n]$ and we want to generate all distinct unions of these s_i. The delay of the algorithm in [17] is $O(nm)$, using a flashlight search algorithm similar to the algorithm of Theorem 4. Among the enumeration problems captured by the framework of saturation by set operators [17] it is the only one not proved to be in strong polynomial delay and it is also proved to be at least as hard to enumerate as the models of a monotone DNF.

If there are m distinct subsets in the instance, these subsets are also solutions then we have an equivalent of Lemma 8. Moreover, if we restrict a set of subsets by fixing an element, we have an inequality similar to Lemma 12. Hence we obtain an algorithm to enumerate union of sets with an average delay $O(n)$ using the branching scheme of Theorem 13. However, the algorithm of Theorem 14 do not seem usable for generating the union of sets, leaving the question of obtaining an algorithm with strong polynomial delay open.

References

[1] Antoine Amarilli, Pierre Bourhis, Louis Jachiet, and Stefan Mengel. A circuit-based approach to efficient enumeration. In *44th International Colloquium on Automata, Languages, and Programming, ICALP 2017, July 10-14, 2017, Warsaw, Poland*, pages 111:1–111:15, 2017.

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

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

[4] Guillaume Bagan. Mso queries on tree decomposable structures are computable with linear delay. In *International Workshop on Computer Science Logic*, pages 167–181. Springer, 2006.

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

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

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

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

[9] Arnaud Durand and Yann Strozecki. Enumeration complexity of logical query problems with second-order variables. In *LIPIcs-Leibniz International Proceedings in Informatics*, volume 12. Schloss Dagstuhl-Leibniz-Zentrum fuer Informatik, 2011.

[10] Christophe Costa Florêncio, Jonny Daenen, Jan Ramon, Jan Van den Bussche, and Dries Van Dyck. Naive infinite enumeration of context-free languages in incremental polynomial time. *J. UCS*, 21(7):891–911, 2015.

[11] Jörg Flum and Martin Grohe. *Parameterized complexity theory*. Springer Science & Business Media, 2006.

[12] Edward Fredkin. Trie memory. *Communications of the ACM*, 3(9):490–499, 1960.

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

[14] Donald Ervin Knuth. *The art of computer programming*, volume 3. Pearson Education, 1997.

[15] Michael Luby and Boban Veličković. On deterministic approximation of dnf. *Algorithmica*, 16(4-5):415–433, 1996.

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

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

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

[19] Gara Pruesse and Frank Ruskey. Generating linear extensions fast. *SIAM Journal on Computing*, 23(2):373–386, 1994.

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

[21] Nicole Schweikardt, Luc Segoufin, and Alexandre Vigny. Enumeration for fo queries over nowhere dense graphs. In *Proceedings of the 37th ACM SIGMOD-SIGACT-SIGAI Symposium on Principles of Database Systems*, SIGMOD/PODS '18, pages 151–163, New York, NY, USA, 2018. ACM. URL: `http://doi.acm.org/10.1145/3196959.3196971`, `doi:10.1145/3196959.3196971`.

[22] Luc Segoufin. Enumerating with constant delay the answers to a query. In *Proceedings of the 16th International Conference on Database Theory*, pages 10–20. ACM, 2013.

[23] Yann Strozecki. *Enumeration complexity and matroid decomposition*. PhD thesis, Paris 7, 2010.

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

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

[26] Robert Alan Wright, Bruce Richmond, Andrew Odlyzko, and Brendan D McKay. Constant time generation of free trees. *SIAM Journal on Computing*, 15(2):540–548, 1986.

A Proof of Proposition 1

Proof. The models of C on variables X are exactly the assignments of the form $\mathbf{1}_C \cup \tau$ for any $\tau : X \setminus \mathsf{var}(C) \to \{0,1\}$. Enumerating the models of C intuitively boils down to enumerating all assignments on variables $X \setminus \mathsf{var}(C)$. We use a Gray code to do this in constant time.

More precisely, we represent a model of C in n registers $R_1, \dots, R_n$ where R_i holds the value of x_i in this model. We initialize the registers such that $R_i = 1$ if $x_i \in C$ and $R_i = 0$ otherwise, that is, if $\neg x_i \in C$ or $x_i \notin \mathsf{var}(C)$.

Now let $k = |X \setminus \mathsf{var}(C)|$ and $\sigma : \{1, \dots, k\} \to \{1, \dots, n\}$ be such that $X \setminus \mathsf{var}(C) = \{x_{\sigma(1)}, \dots, x_{\sigma(k)}\}$ with $\sigma(1) < \cdots < \sigma(k)$. We start by storing the values of σ in an array of size k. We then execute $\texttt{Output}(1, n)$ which outputs the first model of C. After that, we run a Gray code enumeration on the subsets of a set of size k. For each new element generated, the Gray code algorithm flips a bit at some position i. We thus switch the bit of $R_{\sigma(i)}$ and call $\texttt{Output}(1, n)$. This generates all solutions of C since all possible values of the variables not in C are set, while the other

have the only allowed value by C. The delay between two solutions is constant since we only look the value of $\sigma(i)$ in an array and switch the value of a register between two outputs. □

B Proof of Proposition 2

Proof. As explained in Proposition 1, a term $C \in D$, $\mathsf{sat}(C)$ can be enumerated in constant delay using Gray code enumeration. The terms of D are indexed from C_1 to C_m in an arbitrary order. During the algorithm, the state of the enumeration by Gray Code for each term is maintained so that we can query in constant time the next model of a given term. At each step, the algorithm does a loop from C_1 to C_m. For a term C_i the next model is generated and the algorithm tests whether it is a model of some C_j with $j > i$. If not, it is outputted. If a term has no more models we skip it.

By this method, we guarantee that each model is outputted when generated by the term of largest index it satisfies, hence all models are generated and without repetitions. Moreover, at each step of the algorithm, the model given by the last term which has still models will be outputted, therefore the delay is bounded by the time to execute one step of the algorithm.

The cost of the generating new models at each step is bounded by $O(m)$ since each solution is produced in $O(1)$. The cost of testing whether a model satisfies some term of larger index is bounded by $\|D\|$ and it is done at most m times before outputting a solution which implies that the delay is $O(m\|D\|)$. □

C Proof of Proposition 3

Proof. As in the previous algorithm, we run a simple enumeration algorithm on each term and we maintain their states so that we can easily query the next model of a term. We chose to enumerate the models of the terms in lexicographic ascending order (for some arbitrary order of the variables), which can be done with delay $O(n)$ for each term.

The first model of each term which has not yet been outputted is stored in a trie; if all models of a term have been outputted then nothing is stored for this term. Moreover, for each model in the trie, we maintain the list of terms from which it has been generated.

At each step of the algorithm, the smallest model α is found in the trie, then outputted and removed from the trie all in time $O(n)$. Then we use the list of terms which had α as a model, to generate for each of them their next model and add it to the trie in time $O(mn)$ since the insertion can be done in time $O(n)$ and the number of new models is bounded by $O(m)$. The delay of this algorithm is thus bounded by $O(mn)$. By induction, we prove that at each step the smallest non outputted model is outputted, which implies that all models are outputted without repetitions. □

D Pseudocode for Theorem 5

E Proof of Lemma 12

Proof. Denote by n_0 the number of terms of D which contain $\bar{x}$, n_1 the number which contain x and n' the number which do not contain variable x. The terms of $D[x \to 0]$ are the terms of D which contains the literal $\bar{x}$ or which does not contain the variable x. Remark that fixing the variable x

Algorithm 1: Enumerates the models of k-DNF with delay $2^{O(k)}$.

```
Data: A k-DNF-formula D
begin
    if D = ∅ then
        return ∅ ;
    if D = {C} then
        Enumerates the models of C ;
    else
        d ← Ak²2^{kα} ;
        Pick C ∈ D ;
        Every d steps of computation in the next block, output a new model of D[1_C] ;
        begin
            for y ∈ var(C) do
                D^y ← D[0^y_C] ;
        for y ∈ var(C) do
            Recursively enumerates 0^y_C × sat(D^y);
```

to 0 cannot make two terms equal except when one contains $\bar{x}$ and the other does not contain the variable x. Hence $m_0 \geq \max(n_0, n')$. Similarly, $m_1 \geq \max(n_1, n')$.

Since $\max(a, b) \geq (a + b)/2$, we have $m_0 + m_1 \geq (n_0 + n')/2 + (n_1 + n')/2 \geq (n_0 + n_1 + n')/2$. By definition $m = n_0 + n_1 + n'$, then we have proved the lemma. □