Université de Lille

Faculté des Sciences et Technologies

# Complexity and Algorithms for counting problems

**Master Thesis**

**Oliver Irwin**

Computer Science MSc.

Masters in Computer Science

Département Informatique

Faculté des Sciences et Technologies

september 2022

*This report partially satisfies the requirements defined for the Research course, in the 2nd year of the Masters in Computer Science.*

**Candidate:** Oliver Irwin, No. 11500301, `oliver.irwin.etu@univ-lille.fr`

**Scientific Guidance:** Florent Capelli, `florent.capelli@inria.fr`

**Research Lab:** INRIA - Links

Département Informatique
Faculté des Sciences et Technologies
Campus Cité Scientifique, Bât. M3 extension, 59655 Villeneuve-d'Ascq

september 2022

## Acknowledgements

First and foremost, I would like to thank Florent Capelli for his guidance and scientific counsel during the work on this thesis. His help tremendously contributed to the success of this project.

Besides, I would like to thank Marc Tommasi and Charles Paperman for their help in starting this project.

I am also grateful to Tristan Coignion for his help and motivation boosts, and to Catherine Herpin and Jean-Carles Longuet for their proofreading of this document.

## Abstract

We take an interest in the theory behind the complexity of counting problems. We work on understanding the hardness that comes from counting, and the ways in which approximating can be used to give a fast and reliable answer to otherwise intractable exact approaches. Our study mainly focuses on the different randomised algorithms that were developed to give a good approximation of the #DNF problem, and the links from this problem to harder problems such as #NFA.

**Keywords**: approximate counting, boolean formulas, automata, randomised algorithms, DNF, NFA, FPRAS

## Résumé

On s'intéresse dans ce travail à la théorie des problèmes de comptage. On cherche à mieux comprendre les difficultés qui proviennent du fait de compter et les différentes manières à travers lesquelles des algorithmes d'approximation peuvent apporter des résultats fiables et rapides sur des problèmes où l'approche exacte est incalculable. L'étude que nous proposons se porte essentiellement sur différents algorithmes d'approximation qui ont été développés pour donner une bonne estimation pour le problème #DNF, ainsi que sur les liens de ce problème avec d'autres problèmes plus difficiles, comme #NFA.

**Mots-clés**: comptage approché, automates, formules booléennes, algorithmes randomisés, DNF, NFA, FPRAS

# Contents

## Introduction

Counting is an important factor in many computational problem. Sometimes it is not enough to know whether a problem has a solution, one has to find out how many solutions exist. Counting can seem to be a simple task, however it has been shown that counting variant of classical decision problems are usually harder[?][?]. In this work, we choose to focus our efforts on studying the complexity of such problems and the methods that have been developed to deal with them. An example of counting problem we have studied is #DNF, which is the problem of counting the number of satisfying assignments for a DNF formula. This problem has real-life applications, in diverse fields such as network reliability problems[?] or probabilistic databases[?], which makes it an interesting problem to study.

We start by developing the complexity theory associated with counting problems with the #Pcomplexity class, and the associated hardness and completeness. We then aim to show how exact counting methods can work with the example of simple problems such as #DFA, which is the problem of counting the number of words accepted by a given deterministic finite automaton. We then go to show that exact counting methods are often at a loss in due to the inherent complexity of counting, implying the necessity for approximation schemes to be devised. This is no easy feat either, as even the #P-completeness does not indicate the approximation difficulty for a given problem[?]. The main example we explain in this study is the #DNF counting problem, which is a #P-complete constrained counting problem where exact counting is untractable. We study the original method brought by Karp, Luby and Madras to construct the first fully polynomial-time randomised approximation scheme (FPRAS) for a #P-complete problem, allowing for reliable approximation of the correct result. Finally we overlook the work of Arenas *et al*[?] on how the complexity of counting the number of accepting words for a non-deterministic finite automaton is also a hard problem that admits a FPRAS.

# 1 Preliminaries

## 1.1 Normal forms and automata

### Normal forms

A boolean variable is a logical variable whose value belongs in $\{0, 1\}$. These variables are used in boolean formulas. If a formula $f$ only contains a single variable or its negation (meaning that with $x \in \{0, 1\}$, $f = x$ or $f = \neg x$)), then it is called a *literal*. A *clause* is another type of boolean formulas which is defined as a disjunction of literals. If a formula can be written as a conjunction of clauses, then the formula is in conjunctive normal form - or **CNF**.

Alternatively, a *cube* is a conjunction of literals, and if a formula can be written as a disjunction of cubes, then it is in disjunctive normal form - or **DNF**.

We define the *size* of a cube (or a clause) as the number of literals that it contains. The size of the formula is then defined as the sum of the sizes of the different cubes.

Let $f$ be a logical formula of $n$ variables. A *solution* to $f$ is a boolean assignment of the $n$ variables that satisfies the formula, meaning it returns **true**.

### Automata

A Non-Deterministic Finite Automaton (**NFA**) is a finite-state machine that is formally defined as a tuple $(Q, \Sigma, \delta, I, F)$. $Q$ is the (finite) set of states the machine can find itself in, $\Sigma$ is the input alphabet, or the set of all the recognised inputs, $\delta : Q \times \Sigma \to Q$ is the transition function that represents the different transitions available for each state. For a given state $s$ and an input $w \in \Sigma$, $\delta(s, w) = P$ where $P$ is the set of all available states from $s$ with $w$. Finally, $I$ and $F$ are respectively the set of initial and final states for the machine.

A Deterministic Finite Automaton (**DFA**) is a less general version of an NFA, where the initial state is unique and the transition function $\delta$ can only return one arrival state for a given start state $s$ and input $w$.

Let $A = (Q, \Sigma, \delta, I, F)$ a finite automaton (deterministic or not) and $w = w_1 w_2 \dots w_n$ a word of length $n$ over alphabet $\Sigma$. $A$ is said to *accept* the word $w$ if there is a sequence of states $r = r_0, r_1, \dots, r_n$ where each state belongs to $Q$ such that :

1. $r_0 \in I$
2. $r_{i+1} \in \delta(r_i, w_{i+1})$ for $i \in [0 \dots n - 1]$
3. $r_n \in F$

In other words, if $w$ can be written by starting at an initial state of $A$ and following different transitions until a final, accepting, state, then it is recognised by $A$.

The set $\mathcal{L}$ of all recognised words for a given automaton is called the language recognised by said automaton: $\mathcal{L}(A) = \{w \mid w \in \Sigma^*, \text{accept}(A, w)\}$.

We define the size of an automaton by counting the number of states $n$ and the number of transitions $m$, and adding them together.

## 1.2 Counting problems

### Definition

Many algorithmical problems are solved by finding a *certificate* representing a solution to the problem. However, in many cases, knowing that a solution exists is not enough and one is interested in knowing the number of valid certificates. Formally, if $\Sigma$ is an input alphabet and $A$ a relation with $A \subseteq \Sigma^* \times \Sigma^*$. $A$ maps words $x \in \Sigma^*$ to outputs $y \in \Sigma^*$. An associated counting problem can be defined as trying to evaluate, for a given $x$, the value of $\#Y$, where $Y = \{y \mid (x, y) \in A\}$.

The complexity of counting problems is different than that of the associated decision problems. As it is complicated to compute the exact value for $\#Y$ for many problems, approximating it is often the best course of action. We will show here different examples of the greater difficulty of counting over deciding, before introducing the #P complexity class and approximation techniques for hard counting problems.

### Counting solutions to logical formulas

Let $F = c_1 \vee c_2 \vee \cdots \vee c_m$ be a DNF formula. Each cube $c_i$ is therefore written as the conjunction of $n$ boolean variables or their negation. If the length of the cubes is fixed to be a given integer $k$, then $F$ is said to be a $k$-DNF. The common decision problem is to check whether a boolean variable assignation is a solution to $F$. The associated counting problem is to count the number of satisfying assignments to $F$, also written $\#F$. This problem will be referred to as #DNF. Similarly, the problem of counting the number of variable assignments satisfying a formula in conjunctive normal form is known as #CNF or #SAT.

For instance, given $F := (x_1 \wedge \neg x_2) \vee (x_2 \wedge x_3)$. We want to count the number of assignments of $(x_1, x_2, x_3)$ that satisfy $F$. In this simple case, we can simply build the truth table (tab. **??**).

We can then simply count the number of satisfying assignments. In this case, $\#F = 4$. However, this is a very simple example, where there are only 3 variables, 2 cubes, and the width of the cubes is limited to 2. It only seems logical to grasp that the

| $x_1$ | $x_2$ | $x_3$ | $F$ |
|---|---|---|---|
| 0 | 0 | 0 | 0 |
| 0 | 0 | 1 | 0 |
| 0 | 1 | 0 | 0 |
| 0 | 1 | 1 | 1 |
| 1 | 0 | 0 | 1 |
| 1 | 0 | 1 | 1 |
| 1 | 1 | 0 | 0 |
| 1 | 1 | 1 | 1 |

Table 1: Truth table for $F$

truth table approach is clearly untractable for larger values of $n$. When this becomes the case, we must resort to approximating $\#F$.

## Counting solutions of a Finite Automaton

The decision problem often associated with finite automata is to determine if a given DFA (resp. NFA) accepts a word of size $n$. The counting counterpart to this problem is to find the number of words of size $n$ (written in unary) that the automaton accepts. In this case the problem is called #DFA (resp. #NFA).

Figure 1: Sample DFA

If one wants to count the number of words of size $n$ accepted by a DFA such as the one presented in Figure **??**, the method is relatively simple. For any given state $q$, we can count $k_{i,q}$, the number of words of size $i$ starting from the initial state up to $q$. Until we reach a length of $n$ and an accepting state, we can dynamically define $k_{j+1,q}$ as the sum of the $k_{j,q'}$ for all the states $q'$ such that there is a transition between $q$ and $q'$, that is that there exists $a$ such that $\delta(q, a) = q'$.

For the DFA in Figure **??**, the total number of recognised words of size 1 is 0, while the number of accepted words of length 2 is 3.

However, in case of a NFA, counting the paths leading to the accepting states is no longer an option. In Figure **??**, we can see that the word $w = 0 \cdot 1$ can be recognised by two separate paths $p_1 := s_0 \rightarrow s_1 \rightarrow s_3$ and $p_2 := s_0 \rightarrow s_2 \rightarrow s_3$. To get the exact number of words in the language of the automaton, we now have to find a way to only count a given word once.

Figure 2: Sample NFA

### Complexity of counting problems

It is not hard to convince oneself that in general, the counting counterparts to decision problems are harder in terms of computational complexity. If a decision problem in $\mathsf{P}$ had a simple counting counterpart, this would have consequences, notably implying that $\mathsf{P} = \mathsf{NP}$. This brings the need for an extended complexity theory for counting problems.

A well-known example of the extra complexity brought by counting solutions is the $\mathsf{SAT}$ problem. $\mathsf{SAT}$ asks whether a CNF formula has *a* solution or not. Its counting counterpart is named $\#\mathsf{SAT}$. Knowing the number of solutions to a CNF formula (that is, having a correct answer to $\#\mathsf{SAT}$) allows us to answer in constant time to the associated $\mathsf{SAT}$ problem. On the other hand, knowing that a CNF formula has *a* satisfying assignment does not give us any more information on the total number of such assignments. This goes to show how counting problems can be much more computationally difficult.

The $\#\mathsf{P}$ complexity class was introduced by Valiant[?] to represent the set of counting problems associated with $\mathsf{NP}$ decision problems. Let $f\colon \{0,1\}^* \to \mathbb{N}$. $f$ is in $\#\mathsf{P}$ if there exists a polynomial-time nondeterministic Turing machine $M$ such that for all $x \in \{0,1\}^*$, $f(x)$ is the number of accepting branches in $M$'s computation graph on $x$. Unlike $\mathsf{NP}$ and most complexity classes, $\#\mathsf{P}$ is not a set of problems but the set of all functions $f$ satisfying this condition.

The $\#\mathsf{P}$ class covers both very simple and very difficult counting problems, and even simple decision problems can lead to difficult counting counterparts. The $\mathsf{CYCLE}$ decision problem for instance. Given a directed graph $G$, $\mathsf{CYCLE}$ answers the question of whether $G$ has a simple cycle. This problem can be solved in linear time using a breadth-first search. However, counting the number of cycles in a directed graph is not as simple and it can be shown that $\#\mathsf{CYCLE}$ is not in $\mathsf{FP}$ unless $\mathsf{P} = \mathsf{NP}$[?].

### Problem reductions

In decision problems, reductions are ways to transform an instance of a problem into an instance of a different problem. In general, this is done to simplify the resolution of the problem. However, the reductions used for these problems cannot be transposed exactly to counting problems: new definitions are needed.

**Definition 1.** Let $f, g : \Sigma^* \to \mathbb{Z}$ be two functions. A *parcimonious reduction* from $f$ into $g$ exists if there is a function $e : \Sigma^* \to \Sigma^*$ that can be computed in polynomial time such that for any $x \in \Sigma^*, f(x) = (g \circ e)(x)$. This will be noted $f \leqslant^p_{parci} g$[?].

To extend this definition, we can also introduce the *counting reduction*, that allows for computations to be made on the output of the function $e$.

**Definition 2.** Let $f, g : \Sigma^* \to \mathbb{Z}$ be two functions. A *counting reduction* from $f$ into $g$ exists if there are two extra functions $e, s : \Sigma^* \to \Sigma^*$ that can be computed in polynomial time such that for any $x \in \Sigma^*, f(x) = s(x, (g \circ e)(x))$. This will be noted $f \leqslant_c^p g$[?].

**#P-completeness**

In a similar fashion than for decision problems, we can introduce the notion of *hardness* for the #P class. A problem $f$ is #P-hard if, for any $g \in \#\mathsf{P}, g \leqslant_c^p f$. If a problem $f$ is both in #P and #P-hard, then it is said to be #P-complete. Using parcimonious reduction instead of counting reduction would make a stronger requirement for #P-completeness.

Multiple problems have been found to be #P-complete. #SAT for example is a #P-complete problem, and stays so even if we use parcimonious reduction to define #P-completeness. Another example is #DNF, even though DNF itself is in P. #NFA is also known to be a #P-complete problem[?]. These results are shown in Section **??**.

## 1.3 Approximation schemes

As the problems of counting the number of solutions to a boolean formula, or the number of words accepted by a NFA are hard, one must consider the possibility of approximating the exact solution in order to give a fast answer. By fast, we mean in polynomial time in regards to the size of the input.

A **polynomial-time approximation scheme** (**PTAS**) is a type of approximation algorithm for counting and optimisation problems. For an instance $x$ of the problem and an $\varepsilon > 0$, the PTAS produces a result within a factor of $(1 \pm \varepsilon)$ of the true (optimal) solution. The major constraint is that the running time of the algorithm has to be polynomial in the problem size for every $\varepsilon$.

If $\mathcal{A}$ is a PTAS for a instance $x$ of a given problem, then it's result is a $(1 \pm \varepsilon)$-approximation of the optimal solution, that is :

$$(1 - \varepsilon) \cdot f(x) \leq \mathcal{A}(x) \leq (1 + \varepsilon) \cdot f(x)$$

Some problems do not admit a PTAS, however they may admit a randomised version with similar properties. For a counting problem $f$, we can define $\mathcal{A}$ as a **fully polynomial randomised approximation scheme** (**FPRAS**)[?] for $f$ if for every word $x \in \Sigma^*$ and error $\varepsilon > 0$, we have :

$$\mathbf{Pr}(|\mathcal{A}(x) - f(x)| \leq \varepsilon f(x)) \geq \frac{3}{4}$$

and the running time of $\mathcal{A}$ is polynomial in $|x|$ and $\varepsilon^{-1}$.

A FPRAS for a given problem is therefore an algorithm capable of providing a $(1 \pm \varepsilon)$-approximation of the correct solution with a high probability (greater than $\frac{3}{4}$).

### Generalisation to any probability

Once that we have defined an FPRAS as an algorithm that runs in polynomial time relative to $(n, \frac{1}{\varepsilon})$, we consider the possibility of reducing the probability of failure from $\frac{1}{4}$ to an arbitrary $\delta$.

The method used to do so is known as the *median of means* method[?]. Using the mean value could lead to wronged results as our algorithm is currently precise $\frac{3}{4}$ of the time, meaning some runs might return a value far away from the actual result.

We can show that by running an algorithm $A$ around $\log(\frac{1}{\delta})$ times, and by using the median output, we can get a result satisfying :

$$\mathbf{Pr}(|\mathcal{A}(x) - f(x)| \leq \varepsilon f(x)) \geq (1 - \delta)$$

This is now known as an $(\varepsilon, \delta)$-approximation : the result is within $(1 \pm \varepsilon)$ of the true result with a probability greater than $1 - \delta$.

The algorithm is still considered an FPRAS as it runs in polynomial time relative to $\big(n, \varepsilon^{-1}, \log(\delta^{-1})\big)$.

### Importance of FPRAS

Approximation schemes allow for a quick, and hopefully good, estimation of the number of solutions for a given problem. However, the presence of an FPRAS is even more important. Jerrum, Valiant and Vazirani showed in 1986 that every #P-complete problem must admit an FPRAS or is practically impossible to approximate[?].

The existence of a consistent polynomial-time approximation method for a #P-complete problem can therefore be used to build an FPRAS for that problem.

### Monte Carlo algorithms

A Monte Carlo algorithm is a randomised method to answer a problem. The output of such an algorithm can be incorrect, with a usually small probability. The running time of these methods is usually bounded in a number of executions[?].

The idea behind this method is the following. We want to estimate the size of a set $S \subseteq U$. We call $U$ the *universe* for the problem, which is the set of possible solutions and we assume we have a function $f\colon U \to \{0,1\}$ such that $S = \{x \in U | f(x) = 1\}$ and that we know how to efficiently compute $f$.

A *trial* $X$ in a Monte Carlo algorithm is usually defined as such :

1. Sample uniformly an element $x$ from universe $U$
2. Return $f(x)$.

This kind of trial is run $N$ times, that is, we compute $X_1, \dots, X_N$ values in $\{0,1\}$. It turns out that if $N$ is large enough, $\frac{\sum_i X_i}{N}$ is a good approximation of $r := \frac{|S|}{|U|}$ with high probability.

We estimate the value of $N$ as follows: we consider the outcome of a trial to be a random variable $X_i$ and we denote by $X = \frac{\sum_{i=1}^n X_i}{N}$ to be the random variable describing the output of the algorithm. We want to pick $N$ such that the probability that $X$ is not a good $(1 \pm \epsilon)$-approximation of $r$ to be small, that is, we want $\mathbf{Pr}(|X - r| > \epsilon) < 1/4$ for a given $\epsilon$. To do that, we can upper bound this probability using Chebyshev's inequality that says: $\mathbf{Pr}(|X - \mathbb{E}[X]| > \mathsf{st}[X]y) \leq \frac{1}{y^2}$ where $\mathbb{E}[X]$ is the expected value of $X$ (which is $r$ in our case) and $\mathsf{st}[X]$ is the standard deviation of $X$ (which is $\frac{r(r-1)}{N}$ in our case). By setting $y = \frac{\epsilon}{\mathsf{st}[X]}$, we can verify that taking $N = \frac{4(1-r)}{r\epsilon^2}$ ensures that $\mathbf{Pr}(|X - r| > \epsilon) < 1/4$.

Hence for the Monte Carlo methods to give a FPRAS, two things have to be ensured: first, one needs to be able to sample elements uniformly from the universe and second, one needs to have $1/r = |U|/|S|$ to be of polynomial size.

# 2 Observations and known difficulties

## 2.1 Complexity of #DNF

The problem of determining whether a DNF formula has a satisfying assignment or not is trivial. However, counting the number of such satisfying assignments is a lot more complicated. In fact, #DNF is even a #P-complete problem. To prove this, we can reduce #SAT to #DNF.

Let $f$ be a CNF formula and $g = \neg f$. By distributing the negation and using de Morgan's laws, we can rewrite $g$ into a DNF formula $g'$ that has the same number of literals as $f$ as they are just negated. This implies we keep the same formula size. The fact that we have a negation implies that any satisfying assignment for $f$ does not satisfy $g'$ (and vice versa). This in turn implies that if we know the number of satisfying assignments to $g'$, we can compute the solution to $\#f$, as $\#f = 2^n - \#g'$. This means solving #DNF allows for a solution to #SAT, which is a #P-complete problem. As we have reduced the problem using a counting reduction, we can therefore say that #DNF is a #P-complete problem too.

## 2.2 Complexity of #NFA

For a clause (or a cube) $C$, we define $var(C)$ as the set of boolean variables used to construct the clause. We can then define $L_C^n$ as the set of all the words $\tau(x_1)\dots\tau(x_n)$ where $\tau$ satisfies $C$.

**Lemma 3.** *Let $C$ be a cube such that $var(C) \subseteq \{x_1, \dots, x_n\}$. Then there exists a NFA $A$ defined on an alphabet $\{0,1\}$ and of size of $O(|C|)$, such that for any word $w$, $w \in \mathcal{L}(A)$ if and only if :*

- $|w| = n$
- *the assignment $\tau$, defined as: $\forall i \in [0 \dots n], \tau(x_i) = w_i$, is in $L_C$*

*Proof.* The proof of Lemma **??** is quite straightforward. We start by building a NFA $A$ with $n+1$ states, named $q_0, \dots, q_n$. The transition function is defined as follows:

- if the literal $x_i$ appears in $C$, then $\delta(q_{i-1}, 1) = q_i$
- if the literal $\neg x_i$ appears in $C$, then $\delta(q_{i-1}, 0) = q_i$
- if $x_i$ is not in the cube, then $\delta(q_{i-1}, 0) = \delta(q_{i-1}, 1) = q_i$

This NFA is therefore simply a path of length $n+1$ and with no loops.

We now proceed to show that the language accepted by automaton $A$ is equal to $L_C$. For this, we start by showing that every word accepted by $A$ has size $n$ exactly.

After reading the first letter, if the automaton has not failed, it has followed the transition from $q_0$ to $q_1$. Moreover, suppose the automaton is in state $q_{i-1}$ (with $1 < i < n$). Reading another letter without failing would mean that the automaton would end in state $q_i$. This implies that when the NFA reads $i$ letters without failing, it ends up at state $q_i$ for any $i < n$. In order for the word to be fully recognised, it has to reach $q_n$, the only accepting state, which, due to the fact that the automaton has no loops, means having read *exactly* $n$ letters without failing.

The second step of the proof is to show that $L(A) \subseteq L_C$. Let $w$ be a word accepted by this automaton and $\tau_w$ the boolean assignment built such that for any $i$, $\tau_w(x_i) = w_i$, which is well defined from what preceeds. As we have showed that any word $w$ accepted by $A$ has to be of length $n$, $\tau_w$ is well defined for all possible values of $i$. Let $l$ be literal of $C$ and $x_i$ be its underlying variable. The fact that $w$ is accepted by the NFA implies that the transition from $q_i$ to $q_{i+1}$ is taken. From the way the automaton was built, this means that if $x_i$ appears positively (resp. negatively) in $C$, then $w_i = 1$ (resp. $w_i = 0$). In both cases, $\tau_w(l) = 1$, meaning the assignment $\tau_w$ satisfies every literal in $C$ and therefore satisfies $C$ as a whole. This shows that the words recognised by this automaton are satisfying assignments to the original cube.

For the last step of the proof, we show the counterpoint, which is that $L_C \subseteq L(A)$.

Let $\tau$ be a boolean assignment satisfying cube $C$. We show by induction on $i$ that, after reading $\tau(x_1) \dots \tau(x_i)$, automaton $A$ reaches state $q_i$. Suppose the automaton to be in state $q_0$. We read $\tau(x_1)$ and are confronted with one of three cases. The first possibility is that literal $x_1$ appears in cube $C$. In this case, we know from the automaton's definition that $\delta(q_0, 1) = q_1$. The fact that $\tau$ satisfies $C$ means that $\tau(x_1) = 1$, and therefore the automaton reaches $q_1$. The second possibility is that $\neg x_1$ appears in $C$, which implies $\delta(q_0, 0) = q_1$. Again, from the fact that $\tau$ satisfies $C$, we have $\tau(x_1) = 0$ and therefore $A$ reaches the next state. The last possibility is that the literal does not appear in $C$, in which case no matter what is read, the automaton can reach state $q_1$. We use this as the base case for our proof. Using the induction hypothesis, we assume that this stays true for $i > 1$. The way automaton $A$ is built means that by reading $\tau(x_{i+1})$ from state $q_i$, the automaton ends up in state $q_{i+1}$. If $x_{i+1}$ is not in the cube, the construction still allows for the passage to the next state as $\delta(q_i, 0) = \delta(q_i, 1) = q_{i+1}$. As we have a base case and induction, this proof is complete. This means that by applying this induction hypothesis for $i = n$, we observe that reading the sequence $\tau(x_1) \dots \tau(x_n)$ reaches the final and accepting state $q_n$. This means that the word built by this sequence of letters is accepted and therefore $L_C \subseteq L(A)$.

We have therefore shown that we have $L_C \subseteq L(A) \subseteq L_C \implies L_C = L(A)$. As the language recognised by the automaton is composed solely of satisfying assignments to $C$, and any such satisfying assignment is recognised by the automaton, the two sets are equal.

Finally, as this automaton is built using $n$ transitions between $n+1$ states, its size is in $O(|C|)$. □

Figure 3: Example of a NFA built from a cube

**Lemma 4** (Extension to DNF formulas)**.** *Let $D = C_1 \vee \ldots \vee C_m$ be a DNF formula such that for any $i \in [1; m]$, $var(C_i) \subseteq \{x_1, \ldots, x_n\}$. Then there exists a NFA $A$, defined on an alphabet $\{0,1\}$ and of size of $O(|D|)$, such that for any word $w$, $w \in \mathcal{L}(A)$ if and only if :*

- $|w| = n$
- *the assignment $\tau$, defined as: $\forall i \in [0 \ldots n], \tau(x_i) = w_i$, satisfies $D$*

*Proof.* For a DNF formula to be satisfied, at least one of the cubes must be satisfied. Lemma **??** tells us that for any given cube over $n$ variables, there exists a NFA that fits this description.

Consider a NFA $A$ consisting of the union of $m$ NFAs $A_1, \ldots, A_m$ such that for any given $i \in [1; m]$, $A_i$ is the NFA built to recognise the satisfying assignments of the cube $C_i$. $A$ has $m$ initial states and $m$ final accepting states, each corresponding to different assignments in $D$.

The satisfying assignments for a union of cubes is the union of the satisfying assignments for the cubes themselves, and the language recognised by a union of NFAs is the union of the languages recognised by the individual automata. Using Lemma **??**, we can see that the language recognised by $A$ consists of all of the satisfying assignments to the $C_i$ cubes, and therefore to $D$.

The size of $D$ is at most $nm$ ($n$ variables used in $m$ cubes), and the size of the resulting NFA is $(2n+1) \cdot m$, which is indeed of the same order. Moreover, as the NFAs do not interact with each other, the length of the accepted words is still $n$. □

Lemma **??** implies that any DNF formula can be written in the form of a NFA with a conservation of the number of solutions. This means we can construct a reduction from #DNF to #NFA. We know that #DNF is a #P-complete problem. #NFA is a problem is #P, and as a #P-complete problem can be reduced to it, this means it also is #P-complete.

# 3 Sampling and counting in DNF formulas

The #DNF counting problem is a emblematic problem as it is the first #P-complete problem where it has been shown that a FPRAS can be built to solve it. The first randomised algorithm to provide polynomial-time and reliable approximation for #DNF was based on the Monte-Carlo approach. This algorithm was published in 1989[?] and is still used today, even if different methods have since been published using alternative search methods.

## 3.1 Sampling in a DNF formula

In order for a counting algorithm based on a Monte-Carlo approach to be reliable, the random sampling of the elements has to be uniform. That is, that every element of the universe has to have an equal probability of being sampled. The basic Monte-Carlo approach could be summarised in 3 steps : sample an element, check if it is a solution to the current problem and repeat enough to be able to take a proper mean.

When counting the number of solutions to a DNF formula, sampling is not as easy as just choosing a random boolean assignment and checking whether it satisfies a cube in the formula. The main problem is that the solutions of a given DNF formula can be small in regards to the universe, which is $\{0, 1\}^n$. A very naive version of a Monte Carlo method could be to sample a random element and check if it satisfies the formula, but this could converge very slowly, meaning a greater number of iterations for the same precision.

For a DNF formula $F = C_1 \vee \ldots \vee C_m$, we call $D_i$ the set of satisfying assignments for cube $C_i$. With this notation, finding the cardinality of $D = \cup_i D_i$ is the same as finding the number of satisfying assignments to $F$, without counting assignments multiple times each. The universe $U$ is defined as the direct sum of the $D_i$.

In 1989, Karp, Luby and Madras[?] proposed a method to uniformly sample a boolean assignment belonging to the universe. The idea is to first choose a cube, then choose a assignment satisfying that cube. This idea is inspired by methods used to solve the *coverage of sets* problem[1]. Two assumptions are made in order for this method to work. First, it must be easy to compute the size of $D_i$ for any value of $i$. Second, it should be possible to uniformly sample an element from any $D_i$.

The second assumption implies that the probability of sampling any given element $s \in D_i$ is $1/|D_i|$. Also, the fact that we have build the universe as the direct sum of the $D_i$ means that we know $|U| = \sum_{i=1}^{m} D_i$. This means we are able to choose a cube $C_i$ with probability $|D_i|/|U|$. This means that the probability, when sampling

[1]the *coverage of sets* problem aims to provide a correct estimation of the cardinality of the union of multiple sets

$X$ that it is equal to a given assignment $s$ from the universe $U$ is :

$$\mathbf{Pr}[X = s] = \frac{1}{|D_i|} \cdot \frac{|D_i|}{|U|} = \frac{1}{|U|} \tag{1}$$

Equation **??** goes to show that the probability of sampling any given assignment is uniform on the universe.

## 3.2 Monte-Carlo method for #DNF

The method Karp *et al* proposed in [**?**] uses the method above to describe part of a Monte-Carlo trial.

As a quick example of how these methods are used, let $F = C_1 \vee C_2 \vee C_3$, where $C_1 = X_1 \wedge X_2$, $X_2 = \neg X_1$ and $C_3 = \neg X_3$. We can build the following matrix :

| 1 | 2 | 3 | $x_1$ | $x_2$ | $x_3$ |
|---|---|---|---|---|---|
|  | ■ | □ | 0 | 0 | 0 |
|  | ■ |  | 0 | 0 | 1 |
|  | ■ | □ | 0 | 1 | 0 |
|  | ■ |  | 0 | 1 | 1 |
|  |  | ■ | 1 | 0 | 0 |
| ■ |  | □ | 1 | 1 | 0 |
| ■ |  |  | 1 | 1 | 1 |

Table 2: Example matrix for a simple DNF

In this matrix, a box in cell $(s, i)$ implies that $(s, i)$ belongs to $U$. Let $f$ be the function that for a couple $(s, i)$ returns 1 if $i$ is the smallest index such that $s \in D_i$, and 0 otherwise. A black box means that $f(s, i) = 1$ and an empty box means that $f(s, i) = 0$.

The proposed algorithm starts with a preprocessing phase, where one computes the cardinality of each $D_i$: $\forall i, |D_i| = 2^{n-k}$, where $k$ is the number of literals of cube $C_i$, and the cardinality of the universe $U$, which is defined as $|U| = \sum_1^m D_i$.

For the rest of the algorithm, a *trial* is defined as follows :

1. Sample a random element from the universe with probability $1/|U|$ using the method described above
2. Let $f = 1$ if $i$ is the smallest index such that $s \in D_i$ and 0 otherwise
3. Estimate $Y = f \cdot |U|$

This trial is run $N = m \cdot 4\ln(2/\delta)/\varepsilon^2$ times, with $\delta$ and $\varepsilon$ the estimation parameters. The final output to the algorithm is $\tilde{Y} = \sum_{i=1}^{N} Y_i/N$.

The time complexity for this algorithm is in polynomial-time in regards to the size of the problem and the estimation parameters as it has been computed by Karp, Luby and Madras to be in $O(nm^2 \cdot \ln(1/\delta)/\varepsilon^2)$.

## 3.3 Implications of the KLM method

To be a FPRAS for #DNF, the KLM method must give a answer that we know is close to the true count with a probability higher than 3/4. As defined, one trial of this algorithm consists of taking uniformly at random an element of the universe, computing the result of a function, then estimating the size of the set of satisfying assignments. As this trial is based on the random sampling of an assignment, the important factor in the precision of the algorithm is the number of trials. Karp, Luby and Madras present the following *Zero-One Estimator Theorem* to prove $N$ trials is sufficient for $\varepsilon, \delta$ precision.
**Theorem 5** (Zero-One Estimator Theorem)**.** *Let $\mu = |G|/|U|$ and $\varepsilon \leq 2$. Then if $N \geq (1/\mu) \cdot 4\ln(2/\delta)/\varepsilon^2$, the Monte-Carlo algorithm described above is a $\varepsilon, \delta$ approximation.*

This theorem is proven in [**?**], but is more general than just a piece of the KLM method. It is a deeper version of what we have explained in section **??**.

This means that the number of trials can be adjusted in order to attain the 3/4 probability threshold. Moreover, the total execution time is show to be polynomial, both in the size of the problem and in $\log(1/\delta), 1/\varepsilon$. This means that the method developped by KLM is a FPRAS for #DNF.

## 3.4 Improvements to the Monte-Carlo method

Many different FPRASs for #DNF have been proposed over the years, most of which were based on the same Monte-Carlo technique than the Karp-Luby-Madras (KLM) method. A second approach was later used, based on hashing techniques in different algorithms, but these generally had a worse time complexity than the Monte-Carlo. In their 2019 article[**?**], Meel, Shrotri and Vardi devise a new hashing-based method that removes the time-complexity difference with the Monte-Carlo algorithms. They also present the first empirical study of the runtime behaviour for different FPRASs on #DNF.

### New search technique and improvements

In this section we mainly explain the different methods used by Meel *et al.* to improve on the KLM Monte-Carlo method. We will not delve deep into implementation details, but instead focus on explaining the main idea.

For hashing-based counting to work properly, the solution space has to be partitioned into cells of similar sizes. The greatest difficulty for these algorithms is to find the correct number of hash constraints that allow for the number of solutions in any given cell to be not too large and not too small. This search for the correct number of constraints is done by enumerating all of the solutions in a given cell. If this number is greater than a set threshold, then the number of constraints is increased. The study[?] uses 2-universal hash functions to guarantee a low variance on the number of solutions in the cells.

The added complexity of hashing-based counting in regards to Monte-Carlo techniques has been found to be linked to the search function, whose role is to enumerate the solutions in cells. The calls to the search function were agnostic to each other, leading to redundancy in the enumeration[?]. This is why a new search function was introduced, leading to the removal of the polylog time-complexity difference with the KLM counter. This new search functions keeps track of the count of all solutions currently enumerated. The complexity has been proven to be the same as the KLM counter, however the authors show their method is practically faster.

### Empirical study

The empirical study explained in [?] is based on the generation of a large quantity of random DNF formulas with different cube widths. The analysis is based on the different parameters in play, including the size of the formulas, the number of different variables or the tuning of the FPRAS' probability parameters.

The results of the study show that the Monte-Carlo based algorithms scale better according to the $\delta$ parameter. This implies that these algorithms will perform better than the hashing-based algorithms when the probability threshold - the *confidence* - for the approximation is increased. The conclusion of this first proper empirical study is that even if the implementation of choice for most cases is the regular KLM counter, the extended Monte-Carlo version can be better when we have no information about the formula or if the solution density is known to be low.

## 4 Approximating and sampling for NFA

In this section, we revisit the recent algorithm by Arenas, Croquevielle, Jayaram and Riveros [?] to approximate the number of words of size $n$ accepted by an NFA. In this paper, the authors give an FPRAS for the #NFA problem, a problem that we have shown to be #P-complete in Section **??**. The paper is very dense and technical but our goal here is to highlight the main ideas of the paper and show its relations with the KLM algorithm.

To explain the algorithm, we fix in this section an NFA $A = (Q, \{0,1\}, \delta, I, F)$ on alphabet $\{0,1\}$ and $n \in \mathbb{N}$. Our goal is to compute a $(1 \pm \epsilon)$-approximation of the size of $L_n(A) := L(A) \cap \{0,1\}^n$, the set of words of length $n$ accepted by $A$. For a state $q$ and a number $i \leq n$, we denote by $L_n^q$ the set of words $w$ of length $i$ for which there is a path from an initial state of $A$ to $q$.

### 4.1 Main idea

The idea of [?] is to dynamically compute a *good approximation* $k_i^q$ of $|L_i^q|$. These intermediate approximations are used to construct Las Vegas uniform samplers for $L_i^q$ that are themselves used in a generalization of the KLM algorithm to compute $k_{i+1}^q$.

The algorithm itself relies on the following observation: $L_{i+1}^q = \bigcup_{(q',0,q)\in\delta} L_i^{q'} \uplus \bigcup_{(q',1,q)\in\delta} L_i^{q'}$ which is just another way of saying that a path of length $i+1$ in $A$ that ends in $q$ is formed from a path of length $i$ that ends in a state $q'$ and takes one last transition to $q$ that is labelled by either 1 or 0. Getting a good approximation $k_{i+1}^q$ of $|L_{i+1}^q|$ can hence be seen as the procedure able to construct a good approximation of $|\bigcup_{(q',0,q)\in\delta} L_i^{q'}|$ and of $|\bigcup_{(q',1,q)\in\delta} L_i^{q'}|$.

In essence, the #NFA problem boils down to be able to approximate, given $S \subseteq Q$, the size of $\bigcup_{q\in S} L_i^q$. To directly adapt the trial phase of KLM presented earlier in Section **??** to the NFA case, one would ideally need to be able to efficiently perform the following three procedures for every $i$ and $q$:

(i) decide whether a given word $w$ is in $L_i^q$,

(ii) compute $|L_i^q|$ and

(iii) uniformly sample in $L_i^q$.

It is not hard to see that **??** can be done in time polynomial in the size of $A$ by maintaining the subset of sets that can be reached by reading $w$ from an initial state. However, from the #P-completeness of #NFA, it is unlikely that **??** and **??** can be efficiently performed. Hence the idea of [**?**], though it is not exactly phrased in this way, is to show that KLM can be adapted when we relax **??** and **??**.

## 4.2 Estimating from sketches

The first ingredient of the algorithm is to generalize the KLM algorithm to estimate, for a given $Q' = \{q_1, \ldots, q_p\} \subseteq Q$, the size of $L_i^{Q'} := \bigcup_{q \in Q'} L_i^q$. For now, we assume that we are able to sample uniformly in $L_i^q$ (see next section for a presentation of how the authors manage to do it). Assume also that good approximations $k_i^q$ of $|L_i^q|$ have been computed.

The idea is to uniformly sample for each $q \in Q'$ a set $S(q) \subseteq L_i^q$ which is called a *sketch*. Then a step similar to KLM is applied: we construct $T(q_\ell)$ by discarding from $S(q_\ell)$ every word $w$ such that $w \in S(q_{\ell'})$ for some $\ell' < \ell$. Then the size of $L_i^{Q'}$ is estimated as follows:

$$\sum_{j=1}^{p} k_i^{q_j} \frac{|T(q_j)|}{|S(q_j)|}.$$

This algorithm, though different in the way it is presented, is akin to the Monte Carlo method presented in KLM. Indeed, their estimation can be rephased as follows: given sets $A_1, \ldots, A_p$ such that:

- one is provided with $a_1, \ldots, a_p$ such that $a_i$ is a good approximation of $A_i$,
- one has a procedure to uniformly sample in $A_i$ for every $i \leq p$,

one can estimate the size of $\bigcup_{i=1}^{p} A_i$ by running $p$ different Monte Carlo algorithms.

The estimate given by the authors boils down to computing an estimate $\tilde{r_i}$ of the ratio $\frac{|A_i \setminus \bigcup_{j<i} A_j|}{|A_i|}$ with a Monte Carlo method using the uniform sampler for $A_i$. Finally they estimating $|A_i \setminus \bigcup_{j<i} A_j|$ as $a_i\tilde{r_i}$ which they use to estimate $|\bigcup_{i \leq p} A_i| = \sum_{i=1}^{p} |A_i \setminus \bigcup_{j<i} A_j|$ with $\sum_{i=1}^{p} a_i\tilde{r_i}$.

## 4.3 Las Vegas Sampling in NFA

The last ingredient of the proof of [**?**] is a Las Vegas sampling algorithm for NFA. It is built inductively for every length of word using the fact that one has good approximation of $|L_i^q|$. They actually construct an algorithm having the following property: given a set $A$, a *controlled Las Vegas uniform sampler* for $A$ is an algorithm that given $\alpha \in [0, 1]$ such that $\alpha < |A|^{-1}$, returns a value $\tilde{a} \in A \cup \{\mathbf{fail}\}$ such that for every $a \in A$, $\mathbf{Pr}(a = \tilde{a}) = \alpha$. In other word, it is a Las Vegas algorithm since provided it does not fail, then its output is uniform, but we can also control its probability of failing. We say that $\alpha$ is the *controlled failure* the algorithm. The key insight of the paper can be summarized in the following lemma:

**Lemma 6.** *Let $A_1$, $A_2$ be two sets and $a_1, a_2$ be such that:*

- *$A_1 \cap A_2 = \emptyset$,*
- *$A_1$ and $A_2$ have both a controlled Las Vegas uniform sampler,*
- *$a_1$ and $a_2$ are $(1 \pm \epsilon)$-approximation of $|A_1|$ and $|A_2|$ respectively.*

*Then $A_1 \uplus A_2$ admits a controlled Las Vegas uniform sampler.*

*Proof.* The idea of the algorithm is to compensate the uncertainty on $|A_1|$ and $|A_2|$ by controlling the output of the samplers. Given $\alpha < \frac{1}{|A_1|+|A_2|}$, we sample from $A_1 \cup A_2$ with controlled failure $\alpha$ as follows:

- Choose $i = 1$ with probability $\frac{a_1}{a_1+a_2}$ and $i = 2$ otherwise.
- Sample in $A_i$ with controlled failure $\alpha\frac{a_1+a_2}{a_i}$.

We now prove that this algorithm samples with controlled failure $\alpha$. Let $a \in A_1$. The probability that the algorithm outputs $a$ is the probability that we picked $i = 1$ times the probability that the sampler of $A_1$ outputs $a_1$, that is $\frac{a_1}{a_1+a_2} \times \alpha\frac{a_1+a_2}{a_1} = \alpha$. $\square$

The way Lemma **??** is written here is not enough to completely get the desired bounds in order to have an FPRAS for NFA but it contains the main idea on how the sampling algorithm works. The lemma is applied in the following scenario: in order to make the estimation of Section **??** work for sets $L^q_{i+1}$, one needs to be able to sample words in sets of the form $L^{Q'}_{i+1}$ for some $Q' \subseteq Q$. To to so, one can assume that we already have computed approximation on the size of $L^Q_i$ and that we have controlled Las Vegas uniform sampler for these sets. One get a controlled Las Vegas uniform sampler for $L^{Q'}_{i+1}$ by observing that $L^{Q'}_{i+1} = L^{Q_0}_i \cdot 0 \uplus L^{Q_1}_i \cdot 1$ where $Q_0$ (resp. $Q_1$) is the set of states from which there is a transition labeled with 0 (resp. 1) to a state of $Q'$.

To finally get an FPRAS for NFA, the authors combine the ideas presented above with careful calculation on the precision of the needed approximation to have a good final approximation with high probability.

## Conclusion

In this work, we have seen the importance of being able to reliably approximate the results of complicated problems when finding the exact solution is untractable. We have demonstrated the complexity behind counting problems, especially #DNF and #NFA. The implementation of a fully polynomial-time randomised algorithm for #DNF from Karp, Luby and Madras[?] and the subsequent efforts to improve it are the cornerstones of building reliable and fast approximation schemes for hard counting problems.

We also have spent a great deal of time during this work studying the way that the methods introduced by KLM can be used in the context of harder problems, notably by Arenas *et al.* on #NFA[?]. Their contribution to developing a FPRAS for #NFA is an interesting avenue for the applications of counting solutions to real-world problems, and we look forward to later attempt to rewrite and generalise their method.