Available online at www.sciencedirect.com

ScienceDirect

Electronic Notes in Theoretical Computer Science 315 (2015) 17–30

Electronic Notes in Theoretical Computer Science

www.elsevier.com/locate/entcs

Using Binary Patterns for Counting Falsifying Assignments of Conjunctive Forms

Guillermo De Ita Luna[1]

Facultad de Ciencias de la Computación
Benemérita Universidad Autónoma de Puebla
Puebla, México

J. Raymundo Marcial-Romero[2],

Facultad de Ingeniería
Universidad Autónoma del Estado de México
Toluca, México

Pilar Pozos-Parra[3]

División Académica de Informática y Sistemas
Universidad Juárez Autónoma de Tabasco
Tabasco, México

José A. Hernández[4]

Facultad de Ingeniería
Universidad Autónoma del Estado de México
Toluca, México

Abstract

The representation of the set of falsifying assignments of clauses via binary patterns has been useful in the design of algorithms for solving #FAL (counting the number of falsifying assignments of conjunctive forms (CF)). Given as input a CF formula F expressed by m clauses defined over n variables, we present a deterministic algorithm for computing #FAL(F). Principally, our algorithm computes non-intersecting subsets of falsifying assignments of F until the space of falsifying assignments defined by F is covered. Due to #SAT(F) = 2^n-#FAL(F), results about #FAL can be established dually for #SAT. The time complexity of our proposals for computing #FAL(F) is established according to the number of clauses and the number of variables of F.

Keywords: #SAT, #FAL, Binary Patterns, Enumerative Combinatorics.

[1] Email:deita@cs.buap.mx

[2] Email:jrmarcialr@uaemex.mx

[3] Email:pilar.pozos@ujat.mx

[4] Email:xoseahernandez@uaemex.mx

http://dx.doi.org/10.1016/j.entcs.2015.06.003

1 Introduction

The problem of counting models for a Boolean formula (#SAT problem) can be reduced to several problems in approximate reasoning. For example, estimating the degree of belief in propositional theories, generating explanations to propositional queries, repairing inconsistent databases, Bayesian inference and truth maintenance systems [4,12,13,14]. The above problems come from several AI applications such as planning, expert systems, approximate reasoning, etc.

There are many real-life problems that can be abstracted as counting combinatorial objects on graphs. For instance, reliability network issues are often equivalent to connected component issues on graphs, e.g. the probability that a graph remains connected is given by the probabilities of failure over each edge, which is essentially the same as counting the number of ways that the edges could fail without losing connectivity [13].

The combinatorial problems that we address in this paper are the computation of the number of models and falsifying assignments for Boolean formulas in Conjunctive Forms (CF), denoted as #SAT and #FAL, respectively. Both problems (#SAT and #FAL) are classical $\#P$-complete problems even for the restricted cases of monotone and Horn formulas. We also show that string patterns can be used as a succinct representation of the set of falsifying assignments of conjunctive formulas.

Among the class of #P-complete problems, #SAT is considered a fundamental instance due to both, its application in deduction issues, and its relevance in establishing a boundary between efficient and intractable counting problems.

1.1 Literature Review

Given a 2-CF F with n variables and m clauses, it is common to analyze the computational complexity of the algorithms for solving #SAT regarding to n or m or any combination of both [14,15].

The standard strategy used to solve #SAT comes from variants of the classical Davis and Putnam (D&P) method, especially designed to solve the SAT problem. In this case, the main variant arises from reviewing the entire tree of search on the set of assignments of the formula; for example, the backtracking process has to be applied not only when a partial assignment falsifies a sub-formula, but also for assignments satisfying it. However, now a days, each variant from D&P have an exponential-time complexity [4,14].

In [6], some cases are presented where #SAT(F) is computed in polynomial time considering the graph-topological structure of the constrained graph of F. Additionally, in [5] a new way to measure the degree of difficulty for solving #SAT is presented. It is shown that there is a threshold, determined by the same number of models, where #SAT is computed in polynomial time.

Indeed, as $\#SAT(F) = 2^n - \#FAL(F)$, then $\#SAT(F)$ can also be computed based on the computation of $\#FAL(F)$. And it is the case that analogous results can be proved for $\#SAT(F)$ and $\#FAL(F)$.

One of the first works for computing $\#FAL$ was presented by Dubois [8]. Dubois' method begins analyzing the computation of $\#FAL(C_i \wedge C_j)$ for any two pair of clauses C_i and C_j, as:

$$\#FAL(C_i \wedge C_j) = \#FAL(C_i) + \#FAL(C_j) - \#FAL(C_i \cap C_j).$$

Given a 2-CF $F = \{C_1, \ldots, C_m\}$, the above formula for counting unsatisfying assignments can be extended, based on the inclusion-exclusion formula, as: $\#\text{FAL}(F) = |\bigcup_{i=1}^{m} A_i|$, where each A_i is the subset of assignments from the total formula falsifying C_i, then the following equation is inferred:

$$\#FAL = \sum_{i=1}^{m} |A_i| - \sum_{i<j} |A_i \cap A_j| + \sum_{i<j<k} |A_i \cap A_j \cap A_k| + \ldots + -1^{m-1} |\bigcap_{i=1}^{m} A_i| \quad (1)$$

However, this last inclusion-exclusion formula expresses an exponential number of operations on the input m (number of clauses). Several algorithms have been designed as finer or shorter versions of (1) for computing #FAL [1,2,8,3,10,11].

For example, Lozinskii [11] analyzed the set of terms in (1) in order to reduce the number of operations to be performed. He found that two clauses with opposite literals determine disjoint sets of falsifying assignments; therefore his proposal consists on working with subsets of clauses of F without opposite literals.

One of the main line, appearing during the 90's, was to approximate the computation of inclusion-exclusion formulas via the calculation of a limited number of its terms. For example, Linial and Khan [10,3] have shown that the size of the union in (1) can be approximated accurately when the sizes of only a certain part of the set of intersections are known. Assuming that the intersection sizes are known for all subfamilies containing at most k sets, if $k \geq \Omega(\sqrt{m})$, then the union size can be approximated with an additive error of $O(exp(-(\frac{k}{\sqrt{m}})))$.

The Bonferroni's inequalities generalize the inclusion-exclusion principle by showing that truncactions of the sum at odd (even) depths give upper (lower) bounds. For example, the inclusion-exclusion sum over subsets of size at most k yields an upper bound on the overall sum if k is odd, and a lower bound if k is even. Improvements to Bonferroni's inequalities, either in terms of generalization or reduced computation, is prevalent in the literature [1,2].

From those pioneer works, until now, there are not concise algorithms or methods establishing when #FAL can be computed in polynomial time based on the application of the inclusion-exclusion formula.

In [1], the terms in (1) are ordered via a tree exploration, where there is a node for each subset $S \subset 2^m$. The inclusion-exclusion formula is performed by a search on the tree, summing the contributions of each node. An improvement on this search comes from the observation that a node S with conflicting clauses contribute nothing to the sum, and no descendant of S will ever contribute to the overall sum. Therefore, the sum is optimized by pruning any subtree whose root has conflicting clauses (null intersections).

Bennett [1] looked for the identification of terms with same cardinality but different sign; therefore, their mutual contribution to the sum is zero, meaning that

both terms would be canceled. He also applied subsumption among clauses for pruning large subtrees. He has shown by an empiric analysis, that subsumption can greatly improves the average-case performance of the inclusion-exclusion based model counting.

In order to increase the pairs of clauses with opposite literals, Dubois introduced the reduction of independence between two clauses [8]. The efficiency of this process, as he suggested, depends on the order between consecutive set of clauses, because the independence reduction is not a commutative operation.

In this paper, we present a method for computing #FAL(F) in an incremental way with respect to the set of clauses in F. We order the clauses of the formula, taking advantages of the binary strings representing the falsifying assignments of a set of clauses. In order to accelerate the computation of #FAL(F), we reduce the cardinality of the set of clauses to work with by applying some reductions among clauses; e.g. the application of the independent reduction rule combined with subsumed clause rule.

Our proposal also suggests the possibility of applying a primal-dual procedure for computing #SAT(F), similar to Bonferroni's inequalities for the inclusion-exclusion formula or the primal-dual method for linear programming, where the procedure oscillates from the number of models to the number of falsifying assignments of a subset of clauses of F, according to the smallest value between them.

The paper is organized as follows. In Section 2 we give the basic notation as well as several definitions. In Section 3 we describe the binary pattern based approach for 2-CF cases, and we extend such approach for CF cases. Section 4 provides the algorithm associated with our proposal. In Section 5, we present the conclusions of the work.

2 Preliminaries

Let $X = \{x_1, \ldots, x_n\}$ be a set of n *Boolean variables*. A *literal* is either a variable x_i or a negated variable $\overline{x}_i$. As usual, for each $x_i \in X$, $x_i^0 = \overline{x_i}$ and $x_i^1 = x_i$.

A *clause* is a disjunction of different and non complementary literals (sometimes, we also consider a clause as a set of literals, e.g. $x_1 \vee x_2 = \{x_1, x_2\}$). Notice that we discard the case of tautological clauses. For $k \in I\!N$, a *k-clause* is a clause consisting of exactly k literals. A variable $x \in X$ *appears* in a clause c if either x or $\overline{x}$ is an element of c.

A *conjunctive form* (CF) F is a conjunction of non tautological clauses. We say that F is a monotone positive CF if all of its variables appear in an unnegated form. A k-CF is a CF containing only k-clauses. ($\leq k$)-CF denotes a CF containing clauses with at most k literals. A 2-CF formula F is said to be strict only if each clause of F consists of two literals. The *size* of a formula F is defined as the sum of the number of clauses and variables of F.

We use $\upsilon(X)$ to represent the variables involved in the object X, where X could be a literal, a clause or a CF. For instance, for the clause $c = \{\overline{x_1}, x_2\}$, $\upsilon(c) = \{x_1, x_2\}$. $Lit(F)$ is the set of literals involved in F, i.e. if $X = \upsilon(F)$,

then $Lit(F) = X \cup \overline{X} = \{x_1, \overline{x}_1, ..., x_n, \overline{x}_n\}$. We denote $\{1, 2, ..., n\}$ by $[\![n]\!]$ and the cardinality of a set A by $|A|$.

An *assignment* s for F is a function $s : v(F) \to \{0, 1\}$. An *assignment* s can also be considered as a set of literals without a complementary pair of literals, e.g., if $l \in s$, then $\overline{l} \notin s$, in other words s turns l *true* and $\overline{l}$ *false* or viceversa. Let c be a clause and s an assignment, c is *satisfied* by s if and only if $c \cap s \neq \emptyset$. On the other hand, if for all $l \in c$, $\overline{l} \in s$, then s falsifies c. If $n = |v(F)|$, then there are 2^n possible assignments defined over $v(F)$. Let $S(F)$ be the set of 2^n assignments defined over $v(F)$.

Let F be a CF, F is *satisfied* by an assignment s if each clause in F is satisfied by s. F is *contradicted* by s if any clause in F is falsified by s. A model of F is an assignment for $v(F)$ that satisfies F. A falsifying assignment of F is an assignment for $v(F)$ that contradicts F. The SAT problem consists of determining whether F has a model. SAT(F) denotes the set of models of F, then SAT(F) $\subseteq S(F)$. The set FAL(F) $= S(F) \setminus SAT(F)$ consists of the assignments from $S(F)$ that falsify F.

The #SAT problem (or #SAT(F) problem) consists of counting the number of models of F defined over $v(F)$, while $\#FAL(F)$ denotes the number of falsifying assignments of F. Thus, #FAL(F) $= 2^n$- #SAT(F) and #2SAT denotes #SAT for 2-CF formulas.

A 2-CF F can be represented by an undirected graph, called the *constrained graph* of F, and determined as: $G_F = (V(F), E(F))$, where $V(F) = v(F)$ and $E(F) = \{\{v(x), v(y)\} : \{x, y\} \in F\}$. I.e. the vertices of G_F are the variables of F, and for each clause $\{x, y\}$ in F there is an edge $\{v(x), v(y)\} \in E(F)$.

The *neighborhood* for $x \in V(F)$ is $N(x) = \{y \in V(F) : \{x, y\} \in E(F)\}$ and its *closed neighborhood* is $N(x) \cup \{x\}$ denoted as $N[x]$. The degree of a variable x, denoted by $\delta(x)$, is $|N(x)|$, and the degree of F is $\Delta(F) = max\{\delta(x) : x \in V(F)\}$. The size of the neighborhood of x, $\delta(N(x))$, is $\delta(N(x)) = \sum_{y \in N(x)} \delta(y)$.

An algorithm to compute $\#SAT(F)$ considers the set of connected components of its constrained graph G_F. It has been proved that the set of connected components of a constrained graph can be determined in linear time with respect to the number of clauses in the formula.

Thus, $\#SAT(F) = \#SAT(G_F) = \prod_{i=1}^{k} \#SAT(G_i)$, where $\{G_1, \ldots, G_k\}$ is the set of connected components of G_F [12].

The set of connected components of G_F conforms a partition of F.

So, from now on, we will work with a formula F represented by just one connected component.

3 Computing #FAL via Binary Patterns

Let $F = \{C_1, C_2, \ldots, C_m\}$ be a strict 2-CF (each clause has length 2) and let $n = |v(F)|$. The size of F is $n + m$. Let k be a positive integer parameter such that $k < 2^n$. The values of k, where #SAT(F) $= k$ can be determined in polynomial time, is given by the following cases.

If $k = 0$ or $k = 1$, the *Transitive Closure* procedure presented in [9] can be applied

for determining if $\#\text{SAT}(F) = k$. Such procedure has a linear time complexity on the size of the 2-CF.

If k is upper bounded by a polynomial function on n, e.g. $k \leq p(n)$, then in [5], an exact algorithm was shown for determining when $\#\text{SAT}(F) = k$, and such algorithm has a polynomial time complexity on the size of F.

So, the hard cases to answer whether $\#\text{SAT}(F) = k$ are given when $k > p(n)$. In [7], several hard cases for solving $\#\text{SAT}(F)$ are identified. Such identification depends on the relation between its number of clauses (m) and the variables (n) of the instances F. For example, some of the proved cases were:
If $F = \{C_1, C_2, \ldots, C_m\}$ is a 2-CF, with $n = |v(F)|$, the hard cases to answer whether $\#\text{SAT}(F) = k$, are given when $m > n$ [7].

Lemma 3.1 *Let $F = \{C_1, \ldots, C_m\}$ be a 2-CF and $n = |v(F)|$, if F is not a tautology then $\#FAL(F) \geq 2^{n-2}$ or the number of falsifying assignments is at least 2^{n-2}.*

Proof. Let $C = (l_i, l_j)$ be a clause of F and s a falsifying assignment of C. As we assume that C is not a tautology then $v(l_i) \neq v(l_j)$, and as s falsifies C then $\bar{l}_i \in s$ and $\bar{l}_j \in s$. So two of the n positions in the assignment have fixed values, and there are $n-2$ different variables that can be assigned any truth value. That means that there are 2^{n-2} possible assignments that falsify C. Hence, from the 2^n assignments, 2^{n-2} are falsified by C. Thus, $\#SAT(F)$ is not bigger than $2^n - 2^{n-2}$. □

It is known that for any pair of clauses C_i and C_j, it holds that $\#FAL(C_i \cup C_j) = \#FAL(C_i) + \#FAL(C_j) - \#FAL(C_i \cap C_j)$ [8]. The following lemmas show when the number of models can be reduced.

Lemma 3.2 *Let F be a 2-CF, $n = |v(F)|$. If $C_i \in F$ and $C_j \in F$, $i \neq j$ have non-complementary pairs of literals, but they share a literal (e.g. $C_i \cap C_j \neq \emptyset$), then there are exactly $2^{n-2} + 2^{n-3}$ assignments from $S(F)$ falsifying $C_i \cup C_j$.*

Proof. Since $C_i \cap C_j \neq \emptyset$ the elements A_i and A_j have a same value in the common literal (e.g. $A_i \cap A_j = *\ldots*0*\ldots*0*\ldots*0*\ldots*$) which represent 2^{n-3} assignments. That means that $2^{n-2} + 2^{n-2} - 2^{n-3} = 2^{n-1} - 2^{n-3}$ assignments from $S(F)$ are falsified. □

In our algorithmic proposal, we have used binary patterns in order to represent the set of falsifying assignments of any clause defined on n variables. Those patterns allow us to design procedures for computing $\#\text{FAL}(F)$.

Let $F = \{C_1, \ldots, C_m\}$ be a 2-CF and $n = |v(F)|$. Assume an enumeration over the variables of $v(F)$, e.g. $x_1, x_2, \ldots, x_n$. For each clause $C_i = \{x_j, x_k\}$, let A_i be a set of binary strings of length n such that the values at the j-th and k-th positions of each string, $1 \leq j < k \leq n$, represent the truth value of x_j and x_k that falsifies C_i. E.g., if $x_j \in C_i$ then the j-th element of A_i is set to 0. On the other hand, if $\overline{x}_j \in C_i$ then the j-th element of A_i is set to 1. The same argument applies to x_k. It is easy to show that if $C_i = \{x_j, x_k\}$, then x_j and x_k have the same values in each string of A_i in order to be a falsifying assignment of F. Those variables not

contained in the clause can take any truth value since the clause has been already falsified.

Example 3.3 Let $F = \{C_1, C_2\}$ be a 2-CF and $|v(F)| = 3$. If $C_1 = \{x_1, x_2\}$ and $C_2 = \{x_2, x_3\}$ then $A_1 = \{000, 001\}$ and $A_2 = \{000, 100\}$.

We will use the symbol $*$ to represent the variables that can take any truth value in the set A_i, e.g. if $F = \{C_1, \ldots, C_m\}$ is a 2-CF, $n = |v(F)|$, $C_1 = \{x_1, x_2\}$ and $C_2 = \{x_2, x_3\}$ then we will write $A_1 = \underbrace{00 * * \ldots *}_{n}$ and $A_2 = \underbrace{*00 * \ldots *}_{n}$. This abuse of notation will allow us to present a concise and clear representation in the rest of the paper, for considering the string A_i as a pattern formed from $\{0, 1, *\}$ symbols, and which gives a succinct notation for representing the set of falsifying assignments for any clause C_i.

We define a *falsifying string* as a binary pattern A_i which represents the set of falsifying assignments of C_i. Let $F = \{C_1, \ldots, C_m\}$ be a 2-CF, $n = |v(F)|$, we denote by $Fals\_String(C_i)$ the procedure which constructs the *falsifying string* (with n symbols) of C_i. Given a *falsifying string* A_i, the positions where 0 or 1 appear in A_i are called the fixed values of the string, while the positions where the symbol $*$ appears are called the free values of the string. We define the string representing the null clause as the *full string*: $\underbrace{* * \ldots *}_{n}$.

Lemma 3.4 *[7] Let F be a 2-CF, $n = |v(F)|$. If $C_i \in F$ and $C_j \in F$, $i \neq j$ contain a complementary pair of literals, that is $x_k \in C_i$ and $\overline{x}_k \in C_j$, the falsifying set of assignments A_i and A_j of C_i and C_j respectively, forms a disjoint set of falsifying assignments. Consequently, both clauses suppress exactly $2^{n-2} + 2^{n-2} = 2^{n-1}$ assignments from $S(F)$.*

We propose to generalize the previous definitions and results to any CF and then consider $\#SAT$ and $\#FAL$ problems for CF formulas in general, not only for the 2-CF case. We consider now any CF without restriction on the length of its clauses.

Definition 3.5 [8] Given two clauses C_i and C_j, if they have at least one complementary literal, it is said that they have the *independence* property. Otherwise, we say that both clauses are *dependent*.

Definition 3.5 can be written in terms of falsifying strings as follows:

Definition 3.6 Given two falsifying strings A and B both of the same length, if there is an i such that $A[i] = x$ and $B[i] = 1 - x$, $x \in \{0, 1\}$, it is said that they have the *independence* property. Otherwise, we say that both strings are *dependent*.

Definition 3.7 Let $F = \{C_1, C_2, \cdots C_m\}$ be a CF. F is called independent if each pair of clauses $C_i, C_j \in F, i \neq j$, has the independence property, otherwise F is called dependent.

Let $F = \{C_1, C_2, \cdots C_m\}$ be a CF, $n = |\upsilon(F)|$. Let C be a clause in F and $x \in \upsilon(F) \setminus \upsilon(C)$ be any variable, we have that

$$C = (C \vee \overline{x}) \wedge (C \vee x) \tag{2}$$

Furthermore, this reduction preserves the number of falsifying assignments of C with respect to F, since $\#FAL(C) = 2^{n-|C|} = 2^{n-(|C|+1)} + 2^{n-(|C|+1)} = \#FAL((C \vee \overline{x}) \wedge (C \vee x))$, because $(C \vee \overline{x}) \wedge (C \vee x)$ are two independent clauses. In terms of falsifying strings, let $F = \{C_1, C_2, \ldots C_m\}$, $n = |\upsilon(F)|$, if A is the falsifying string of a clause C_j such that there is an index $i, A[i] = *$ then the falsifying string A can be replaced by two falsifying strings say A_1 and A_2 as follows:

(i) $A_1[j] = A[j]$ if $j \neq i$ and $A_1[j] = 1$ if $j = i$.

(ii) $A_2[j] = A[j]$ if $j \neq i$ and $A_2[j] = 0$ if $j = i$.

It is easy to show that the falsifying assignments represented by A with respect to F are equal to the sum of the falsifying assignments represented by A_1 and A_2 with respect to F.

Given a pair of dependent clauses C_1 and C_2. Let us assume that there are literals in C_1 which are not in C_2, let $x_1, x_2, ..., x_p$ be these literals. There exists a reduction to transform C_2 to be independent with C_1, we call at this transformation the *independent reduction*, and this works as follows: By (2) we can write: $C_1 \wedge C_2 = C_1 \wedge (C_2 \vee \neg x_1) \wedge (C_2 \vee x_1)$. Now C_1 and $(C_2 \vee \neg x_1)$ are independent. Applying (2) to $(C_2 \vee x_1)$:

$$C_1 \wedge C_2 = C_1 \wedge (C_2 \vee \neg x_1) \wedge (C_2 \vee x_1 \vee \neg x_2) \wedge (C_2 \vee x_1 \vee x_2)$$

The first three clauses are independent. Repeating the process of being independent between the last clause with the previous ones, until x_p is considered, we have that $C_1 \wedge C_2$ can be written as:

$C_1 \wedge (C_2 \vee \neg x_1) \wedge (C_2 \vee x_1 \vee \neg x_2) \wedge ... \wedge (C_2 \vee x_1 \vee x_2 \vee ... \vee \neg x_p) \wedge (C_2 \vee x_1 \vee x_2 \vee ... \vee x_p)$.

The last clause contains all literals of C_1, so it can be eliminated because it is subsumed by C_1, and then

$$C_1 \wedge C_2 = C_1 \wedge (C_2 \vee \neg x_1) \wedge (C_2 \vee x_1 \vee \neg x_2) \wedge ... \wedge (C_2 \vee x_1 \vee x_2 \vee ... \vee \neg x_p) \tag{3}$$

We obtain on the right hand side of (3) an independent set of $p+1$ clauses. Let us present the independent reduction transformation in terms of falsifying strings.

Given a pair of falsifying strings A and B. Let us assume that there are indices $x_1, x_2, ..., x_p$ such that $A[x_i] = 1$ or $A[x_i] = 0$ and $B[x_i] = *$, for all $i \in [\![p]\!]$. There exists a reduction to transform B to be independent with A, we call at this transformation the *independent reduction*, and this works by replacing the falsifying string B by p falsifying strings say $B_1, B_2, \ldots B_p$ as follows:

- $B_1[i] = B[i]$ if $i \neq x_1$ and $B_1[i] = 1 - A[i]$ if $i = x_1$.
- $B_2[i] = B[i]$ if $i \neq x_1$ and $i \neq x_2$, $B_2[i] = A[i]$ if $i = x_1$ and $B_1[i] = 1 - A[i]$ if $i = x_2$.
- ...

- $B_p[i] = B[i]$ if $i \neq x_j, j \in [[p]]$ and $B_p[i] = A[i]$ if $i = x_j, j \in [[p-1]]$, and $B_p[i] = 1 - A[i]$ if $i = x_p$.

We will denote the independent reduction between two clauses C_1 and C_2 or between their corresponding falsifying strings A and B as: $reduc(C_1, C_2)$ or $reduc(A, B)$, respectively. Notice that the independent reduction is not commutative, in the sense that $reduc(C_2, C_1)$ builds an independent set of $|Lit(C_2) - (Lit(C_1) \cap Lit(C_2))|$ clauses, while $reduc(C_1, C_2)$ builds an independent set of $|Lit(C_1) - (Lit(C_1) \cap Lit(C_1))|$ clauses. However, $reduc(C_1, C_2)$ and $reduc(C_2, C_1)$ are logically equivalent because they determine the same set of falsifying assignments. Furthermore, when $(Lit(C_1) - (Lit(C_1) \cap Lit(C_2))) = \emptyset$ then the set of falsifying assignments of C_2 is a subset of the set of falsifying assignments of C_1, that is, $FAL(C_2) \subseteq FAL(C_1)$, and then $reduc(C_1, C_2) = C_1$.

Independence reductions containing the smallest number of clauses can be formed by computing both $|Lit(C_1) - (Lit(C_1) \cap Lit(C_2))|$ and $|Lit(C_2) - (Lit(C_1) \cap Lit(C_2))|$ or if we are working with falsifying strings by computing the set of indices $\{i \mid A[i] = 1$ or $A[i] = 0$ and $B[i] = *\}$ and $\{i \mid B[i] = 1$ or $B[i] = 0$ and $A[i] = *\}$. The set with the smallest cardinality can be chosen to be reduced.

For formulas in CF, a pair of dependent clauses has either at least one common literal or not variables in common.

Let C_i, C_j be two dependent clauses with at least one common literal, in a CF F with n variables. Let A and B the falsifying strings of C_i and C_j. That A and B are dependents with at least one common literal means that the following conditions hold:

(i) if $A[i] = x$ then $B[i] \neq 1 - x$ for all $i \in [[n]]$,

(ii) there is a non-empty set of indices x_j such that $A[x_j] = B[x_j]$ and

(iii) there is a set of indices x_j such that ($A[x_j] = 0$ or $A[x_j] = 1$ and $B[x_j] = *$) or ($B[x_j] = 0$ or $B[x_j] = 1$ and $A[x_j] = *$).

Example 3.8 The following table shows two falsifying strings of two dependent clauses with a common literal

	x_1	x_2	x_3	$\cdots$	x_k	$\cdots$	x_{n-1}	x_n	Falsifying string	
C_1	1	*	*	$\cdots$	1	$\cdots$	*	*	1**...1...**	A
C_2	*	1	*	$\cdots$	1	$\cdots$	*	*	*1*...1...**	B

The application of $reduc(A, B)$ modifies the string B as follows

	x_1	x_2	x_3	$\cdots$	x_k	$\cdots$	x_{n-1}	x_n	Falsifying string	
C_1	1	*	*	$\cdots$	1	$\cdots$	*	*	1**...1...**	A
C_2	0	1	*	$\cdots$	1	$\cdots$	*	*	01*...1...**	B

Lemma 3.9 *Let C_i, C_j be two dependent clauses with no common literals, in a formula F with n variables. The number of falsifying assignments for this pair of clauses is:*

$$2^{n-|C_i|} + \sum_{i=1}^{|C_i|} 2^{n-|C_j|-i}$$

Proof. By applying $|C_i|$-times the independent reduction on C_i and C_j. □

Example 3.10 Let $C_i, C_j \in F$, where $C_i = (x_1 \vee x_2)$ and $C_j = (x_3 \vee x_4)$ are dependent clauses. By lemma 3.9, $\#FAL(C_i \wedge C_j) = 2^{n-2} + 2^{n-3} + 2^{n-4}$. The transformation to obtain independent clauses can be applied over C_j as $(x_1 \vee x_2) \wedge (x_3 \vee x_4) = (x_1 \vee x_2) \wedge (\overline{x_1} \vee x_3 \vee x_4) \wedge (x_1 \vee \overline{x_2} \vee x_3 \vee x_4)$.

Lemma 3.11 *For any independent formula $F = \{C_1, \ldots, C_m\}$ involving n variables, $\#FAL(F) = \sum_{i=1}^{m} 2^{n-|C_i|}$.*

Corollary 3.12 *If F is an independent k-CF then $\#FAL(F) = \sum_{i=1}^{m} 2^{n-k}$.*

F is a contradiction when $\#\mathrm{FAL}(F) = 2^n$, then all k-CF with at least 2^k independent clauses will be a contradiction.

Furthermore, let F be a CF, $n = |\upsilon(F)|$ and F is not necessarily an independent formula. Let C be an independent clause with each clause of F, based on the iterative application of lemma 3.4, it holds

$$\#FAL(F \wedge C) = \#FAL(F) + 2^{n-|C|} \tag{4}$$

Then, the independent reduction determines a procedure to compute $\#\mathrm{FAL}(F \wedge C)$ when $\#FAL(F)$ has already been computed. The procedure consists of applying the independent reduction on C and the clauses in F involving the variables $\upsilon(C)$ until we build a new set of clauses C' which will be independent with each clause $C_i \in F$. where $\upsilon(C) \cap \upsilon(C_i) \neq \emptyset$ and then $\#FAL(F \wedge C) = \#FAL(F \wedge C') = \#FAL(F) + \#FAL(C')$.

For example, let $F = \{\{\overline{x}_1, \overline{x}_2\}, \{\overline{x}_3, x_2\}, \{\overline{x}_4, x_3\}, \{x_4, x_5\}, \{x_5, x_6\}\}$. $\#\mathrm{FAL}(F)$ can be computed incrementally using the falsifying strings of the clauses of F. The matrix of Table 1 represents the falsifying string of each clause.

	x_1	x_2	x_3	x_4	x_5	x_6
C_1	1	1	*	*	*	*
C_2	*	0	1	*	*	*
C_3	*	*	0	1	*	*
C_4	*	*	*	0	0	*
C_5	*	*	*	*	0	0

Table 1
The falsifying strings for the clauses in F

Clauses C_1 and C_2 are independent so no reduction is needed between those clauses. The clause C_3 is not independent with C_1, even more, this pair of clauses does not have a common literal hence the independent reduction has to be applied either to C_1 or C_3. Applying it to C_3 we obtain the strings of Table 2.

Now, the four clauses are independent each other. Consider now C_4 whose falsifying string is $***00*$. Again it is not independent with C_1, applying the

	x_1	x_2	x_3	x_4	x_5	x_6
C_1	1	1	*	*	*	*
C_2	*	0	1	*	*	*
C_{3-1_1}	0	*	0	1	*	*
C_{3-1_2}	1	0	0	1	*	*

Table 2
The falsifying strings after applying the independent reduction between C_1 and C_3.

	x_1	x_2	x_3	x_4	x_5	x_6
C_1	1	1	*	*	*	*
C_2	*	0	1	*	*	*
C_{3-1_1}	0	*	0	1	*	*
C_{3-1_2}	1	0	0	1	*	*
C_{4-1_1}	0	*	*	0	0	*
C_{4-1_2}	1	0	*	0	0	*

Table 3
The falsifying strings after applying the independent reduction between C_1 and C_4.

independent reduction rule we obtain the strings of Table 3.

Both C_{4-1_1} and C_{4-1_2} are independent with C_1. The new clauses have to be checked to be independent with the rest of the clauses. It can be noticed that the new clause C_{4-1_1} is not independent with the clause C_2 so the procedure is repeated until each clause in F is independent with each other. The independent reduction application gives the results shown in Table 4.

From lemma 3.11, $\#\text{FAL}(F) = \Sigma_{i=1}^{10} 2^{n-|C_i|} = 2^4 + 2^4 + 2^3 + 2^2 + 2^2 + 2^0 + 2^0 + 2^0 + 2^0 + 2^0 = 53$. Then, $\#\text{SAT}(F) = 2^n - \#FAL(F) = 64 - 53 = 11$.

3.1 Reducing the sizes of CF's

The size reduction of CF's formulas is a relevant task in order to build efficient algorithms with the aim of check $\#\text{FAL}(F)$. In order to reduce the number of clauses while applying the independent reduction principle, the following rule can be applied.

Subsumed clause Rule.

Given two clauses C_i and C_j of a CF F, if $\text{Lit}(C_i) \subseteq \text{Lit}(C_j)$, then C_j is subsumed by C_i, and then C_j can be deleted from F.

Namely, all satisfying assignments of C_j are satisfying assignments of C_i: $\text{SAT}(C_j) \subseteq \text{SAT}(C_i)$. Thus, it is enough to keep C_i (the clause which subsumes) in the CF. In

	x_1	x_2	x_3	x_4	x_5	x_6
C_1	1	1	*	*	*	*
C_2	*	0	1	*	*	*
C_3	0	*	0	1	*	*
C_4	1	0	0	1	*	*
C_5	0	1	*	0	0	*
C_6	0	0	0	0	0	1
C_7	1	0	0	0	0	1
C_8	0	1	1	1	0	0
C_9	0	0	0	0	0	0
C_{10}	1	0	0	*	0	0

Table 4
The falsifying strings after applying the independent reduction between C_1 and C_4.

terms of falsifying strings.

Definition 3.13 Let A and B be two falsifying strings of length n. It is said that B *subsumes* A if there is a set of indices $I = \{i_1, \ldots, i_k\} \subset \{1, \ldots, n\}$ such that the following conditions hold:

- $\forall i \in I, (A[i] = 0$ or $A[i] = 1)$ and $B[i] = *$
- $\forall j \notin I$, $j \in [\![n]\!]$, $A[j] = B[j]$.

The subsumed clause rule requests that each falsifying string will be compared with the rest in the Table of falsifying strings in order to find a subsumed clause, if it exists. That implies the order of $O(n \cdot m^2)$ basic operations in the worst case.

4 An Incremental Computation for #FAL

Given $F(n, m)$ a CF, Algorithm 1 computes $\#SAT(F)$ based on the computation of $\#FAL(F)$. We start with the first clause $C_1 \in F$, and continue adding the following clause making it independent with the previous processed clauses, until all the original clauses in F are processed. Furthermore, if we know the value for $\#SAT(F)$ then we can also determine if F is (or not) satisifiable.

The procedure $Simplify(T_i)$ works on a set of falsifying strings T_i, looking for pair of strings representing subsumed clauses in order to delete the string corresponding to the subsumed clause. The procedure $Simplify(T_i)$ keeps or reduces the cardinality of the set of strings T_i. As $Simplify(T_i)$ involves to compare each falsifying string A with the following strings in T_i, it has a time complexity of $O(n \cdot |T_i|^2)$.

Algorithm 1 Procedure `#Falsifying`(F)

```
Input: A CF F with m clauses and n variables
Output: (True /False )= (SAT(F) = ∅ or not)
Simplify(F); {delete all subsumed clause}
m = |F|; {count the new number of clauses}
for all C_i ∈ F do
    A_i = Fals_String(C_i);
end for
T_1 = A_1; {Begin the table of falsifying strings for F}
for all i = 2 to m do
    T_i = reduc(A_i, T_{i-1}); {Apply independent reduction}
    Simplify(T_i); {delete all subsumed clause}
    if (T_i has a full string) then
        Returns 0 (F 'is unsatisfiable'); {Fals(F) cover all the assignments}
    end if
end for
Ct = Count_Falsifyings(T_m); {Count number of falsifying assignments}
if ((2^n − Ct) > 0) then
    Returns Ct (F 'is satisfiable');
else
    Returns 0 (F 'is unsatisfiable');
end if
```

The procedure $Count\_Falsifyings(T_m)$ counts the number of falsifying assignments of an independent set of clauses represented by the set of falsifying strings T_m. This computation is done in time $O(|T_m|)$.

Furthermore, the function $Count\_Falsifyings$ can be performed into the body of the main loop of the procedure $\#Falsifying(F)$ in order to detect any minimum unsatisfiable subset of clauses of F.

The total time complexity of our proposal is polynomial on n and on the size of the falsifying table T_i. Then, we have to compute the growing size of the set T_i when each new falsifying string A_j is processed into the loop of $\#Falsifying(F)$. Of course, a non-tight upper bound for $|T_m|$ is #FAL(F) itself, because at most each falsifying string can have n fixed values and there are at most #FAL(F) falsifying strings in T_m. However, the representation of each $FAL(C_i)$, $C_i \in F$ by a falsifying string guarantees that it is not needed to express in an exhaustive way the set $FAL(C_i)$.

For formulas F, whose number of falsifying assignments is bounded by a polynomial on the size of F, our algorithm computes #FAL(F) in a polynomial time-complexity on the size of F; therefore, the value #SAT$(F) = 2^n -$ #FAL(F) too. Furthermore, *Count_Falsifyings* is an effective procedure when subset of falsifying assignments of F are represented by a reduced number of fixed variables into the set of variables of $v(F)$, in such a way that the codification of such subset is effective via falsifying strings.

Our proposal suggests the possibility to use a primal-dual method for computing #SAT(F), where the computation of #SAT(F) oscillates from the number of models to the number of falsifyng assignments of subsets of clauses of F, according to the smaller value between them.

5 Conclusions

Given as input a CF F expressed by m clauses defined on n variables, we have shown a deterministic algorithm for computing $\#FAL(F)$. Our method is incremental on

the set of clauses of F. We use a short notation to represent the set of falsifying assignments of clauses using string patterns. And we show how to manipulate such strings to form the set of falsifying assignments of a conjunctive formula.

We begin computing excluded subsets of falsifying assignments of F until the whole space of falsifying assignments defined by F is covered. Due to $\#SAT(F) = 2^n$-$\#FAL(F)$, results about #FAL can be established dually for #SAT. Some reductions in our procedure are used, as subsumed clauses and the independent reduction involving dependent clauses, in order to accelerate the computation of $\#FAL(F)$.

References

[1] Bennett H., Results on Extensions of the Satisfiability Problem, Thesis of Master degree, Computer Sc. Dept., University of Colorado, (2012).

[2] K. Dohmen, Improved Bonferroni Inequalities via Abstract Tubes: Inequalities and identities of the Inclusion-Exclusion Type, Lecture Notes in Mathematics, Springer-Verlag, (2003).

[3] Kahn J., Linial N., Samorodnitsky A., inclusion-exclusion: Exact and Approximate, *Combinatorica*, Vol. 16 , No. 4, (1996), pp. 465–477.

[4] Dahllöf V., Jonsson P., Wahlström M., Counting models for 2SAT and 3SAT formulae, *Theoretical Computer Sciences*, 332(1-3), (2005), pp. 265-291.

[5] De Ita G., Marcial-Romero R., Hernández J.A., A threshold for a Polynomial Solution of #2SAT, *Fundamenta Informaticae*, 113:1(2011), pp. 63-77.

[6] De Ita G., Bello P., Contreras M., New Polynomial Classes for #2SAT Established Via Graph-Topological Structure, *Engineering Letters*, Vol. 15;2, Int. Assoc. of Engineers, www.engineeringletters.com, (2007), pp. 250-258.

[7] De Ita G., Marcial-Romero J.R., Computing #2SAT and #2UNSAT by Binary Patterns, *LNCS, Vol. 7329 - 4th Mexican Conf. on Pattern Recognition*, (2012), pp. 273-282.

[8] Dubois Olivier, Counting the number of solutions for instances of satisfiability, *Theor. Comp. Sc.* 81, (1991), 49-64.

[9] Gusfield D., Pitt L., A bounded approximation for the minimum cost 2-Sat problem, *Algorithmica* 8, (1992), 103-117.

[10] Linial N., Nisan N., Approximate inclusion-exclusion. *Combinatorica*, Vol. 10, No. 4, (1990), pp. 349-365.

[11] Lozinskii E., Counting propositional models, *Inf. Proc. Letters* 41, (1992), pp. 327-332.

[12] Roth D., On the hardness of approximate reasoning, *Artificial Intelligence 82*, (1996), pp. 273-302.

[13] Vadhan Salil P., The Complexity of Counting in Sparse, Regular, and Planar Graphs, *SIAM Journal on Computing*, Vol. 31, No.2, (2001), pp. 398-427.

[14] Wahlström M., A Tighter Bound for Counting Max-Weight Solutions to 2SAT Instances, *LNCS, Vol. 5018 - Parameterized and Exact Computation: Third Int. Workshop*, (2008), pp. 202-213.

[15] Zhou Junping, Yin Minghao, Zhou C., New Worts-Case Upper Bound for #2-SAT and #3-SAT with the Number of Clauses as the Parameter, Proc. of the AAAI, (2010), pp. 217-222.