Available online at [www.sciencedirect.com](http://www.sciencedirect.com/)

[Electronic Notes in Theoretical Computer Science 328 (2016) 31–45](http://dx.doi.org/10.1016/j.entcs.2016.11.004)

[www.elsevier.com/locate/entcs](http://www.elsevier.com/locate/entcs)

The Incremental Satisfiability Problem for a Two Conjunctive Normal Form

Guillermo De Ita Luna[1](#_bookmark0)

*Facultad de Ciencias de la Computaci´on Benem´erita Universidad Aut´onoma de Puebla Puebla, M´exico*

J. Raymundo Marcial-Romero[2](#_bookmark0) Jos´e A. Hern´andez[3](#_bookmark0)

*Facultad de Ingenier´ıa Universidad Aut´onoma del Estado de M´exico*

*Toluca, M´exico*

**Abstract**

We propose a novel method to review *K ▶ φ* when *K* and *φ* are both in Conjunctive Normal Forms (CF). We extend our method to solve the incremental satisfiablity problem (ISAT), and we present different cases where ISAT can be solved in polynomial time.

Especially, we present an algorithm for 2-ISAT. Our last algorithm allow us to establish an upper bound for the time-complexity of 2-ISAT, as well as to establish some tractable cases for the 2-ISAT problem.

*Keywords:* Satisfiability Problem, Incremental Satisfiability Problem, 2-SAT, Entail Propositional Problem, Efficient Satisfiability Instances.

# Introduction

The primary goal of complexity theory is to classify computational problems ac- cording to their inherent computational complexity. A central issue in determin- ing these frontiers has been the satisfiability problem (SAT) in the propositional calculus. The case 2-SAT, to determine the satisfiability of propositional two Con- junctive Normal Forms (2-CF), is an important tractable case of SAT (see e.g. [[2](#_bookmark10)] for polynomial-time algorithms for 2-SAT). And variations of the 2-SAT problem,

e.g. in the optimization and counting area, have been key for establishing frontiers between tractable and intractable problems.

1 [Email:deita@cs.buap.mx](mailto:deita@cs.buap.mx)

2 [Email:jrmarcialr@uaemex.mx](mailto:jrmarcialr@uaemex.mx)

3 [Email:xoseahernandez@uaemex.mx](mailto:xoseahernandez@uaemex.mx)

<http://dx.doi.org/10.1016/j.entcs.2016.11.004>

1571-0661/© 2016 The Author(s). Published by Elsevier B.V.

This is an open access article under the CC BY-NC-ND license (<http://creativecommons.org/licenses/by-nc-nd/4.0/>).

Since automatic reasoning is one of the purer forms of human, intellectual thought, the automation of such reasoning by means of computers is a basic and challenging scientific problem [[17](#_bookmark23)]. One of the fundamental problems in automatic reasoning is the propositional entail problem. This last problem is a relevant task in many other issues such as estimating the degree of belief, to review or update beliefs, the abductive explanation, logical diagnosis, and many other procedures in Artificial Intelligence (AI) applications.

It is known that logic entail problem is a hard challenge in automatic reasoning due to it is co-NP-Hard even in the propositional case [[12](#_bookmark19)]. However, some fragments of propositional logic allow efficient reasoning methods [[4](#_bookmark12)]. One of the most relevant cases of efficient reasoning is the fragment of Horn Formulas. We present a novel method to solve the entail problem between conjunctive forms and we show how to apply this method for solving the incremental satisfiability problem (ISAT) that consists in deciding if an initial knowledge Base *K* keeps its satisfiability anytime a conjuntion of new clauses is added.

Hooker [[10](#_bookmark18)] presented an algorithm for the ISAT problem, in which the main contribution was the speed-up for solving a single formula by solving a growing subset of its constraints. Whittemore et al. [[18](#_bookmark24)] defined the incremental satisfiability problem as the solving of each formula in a finite sequence of subformulas. Solvers, which use a variant of Whittemore’s approach, are ZCHAFF [[14](#_bookmark21)] and SMT-LIB [[1](#_bookmark9)]. E´en et al. [[7](#_bookmark15)] presented a simple interface for ISAT solvers which was first used by the solver MINISAT [[6](#_bookmark14)]. Wieringa [[19](#_bookmark25)] presented an incremental satisfiability solver and some of its applications. Finally, Nadel [[16](#_bookmark22)] presented a variation of ISAT problem under assumptions that are modeled as first decision variables; all inferred clauses that depend on some of the assumptions include their negation.

We present here an algorithm for solving the 2-ISAT problem, and we establish an upper bound for the time-complexity of 2-ISAT, as well as, we show some efficient cases for the ISAT and the 2-ISAT problems.

# Preliminaries

Let *X* = *{x*1*,..., xn}* be a set of *n Boolean variables*. A *literal* is either a variable

*xi* or a negated variable *xi*. As usual, for each *x ∈ X*, *x*0 = *x* and *x*1 = *x*.

A *clause* is a disjunction of different and non-complementary literals. Notice that we discard the case of tautological clauses. For *k ∈ IN* , a *k-clause* is a clause consisting of exactly *k* literals, and a (*≤ k*)-clause is a clause with at most *k* literals. A *phrase* is a conjunction of literals, a *k-phrase* is a phrase with exactly *k* literals. A *conjunctive normal form* (CNF, or CF) *F* is a conjunction of clauses. We say that *F* is a monotone positive CF if all of its variables appear in unnegated form. A *k*-CF is a CF containing only *k-clauses*. (*≤ 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. A *disjunctive normal form* (DF) is a disjunction of

phrases, and a *k*-DF is a DF containing only *k-phrases*.

A variable *x ∈ X appears* in a formula *F* if either *x* or *x* is an element of *F* .

We use *υ*(*X*) to represent the variables involved in the object *X*; where *X* can be a literal, a clause, or a CF. For instance, for the clause *c* = *{x*1*, x*2*}*, *υ*(*c*)= *{x*1*, x*2*}*. *Lit*(*F* ) is the set of literals involved in *F* , i.e. if *X* = *υ*(*F* ), then *Lit*(*F* )= *X ∪ X* =

*{x*1*, x*1*, ..., xn, xn}*. Also we used *¬Y* as the negation operator on the object *Y* . We denote *{*1*,* 2*, ..., n}* by **[***n*]], and the cardinality of a set *A* by *|A|*.

An *assignment s* for a formula *F* is a function *s* : *υ*(*F* ) *→ {*0*,* 1*}*. An *assignment s* can also be considered as a set of literals without a complementary pair of literals, e.g., if *l ∈ s*, then *l /∈ s*, in other words *s* turns *l true* and *l false* or viceversa. Let *c* be a clause and *s* an assignment, *c* is *satisﬁed* by *s* if and only if *c ∩ s /*= *∅*. On the other hand, if for all *l ∈ c*, *l ∈ s*, then *s* falsifies *c*.

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 *υ*(*F* ) that satisfies *F* . A falsifying assignment of *F* is an assignment for *υ*(*F* ) that contradicts *F* . A DF *F* is satisfied by *s* if any phrase is satisfied by

*s*. *F* is contradicted by *s* if all of its phrases are contradicted by *s*.

If *n* = *|υ*(*F* )*|*, then there are 2*n* possible assignments defined over *υ*(*F* ). Let *S*(*F* ) be the set of 2*n* assignments defined over *υ*(*F* ). *s ▶ F* denotes that the assignment *s* is a model of *F* . *s /▶ F* denotes that *s* is a falsifying assignment of

*F* . If *F*1 *⊂ F* is a formula consisting of some clauses from *F* , and *υ*(*F*1) *⊂ υ*(*F* ), an assignment over *υ*(*F*1) is a *partial* assignment over *υ*(*F* ). If *s* has logical values determined for each variable in *F* then *s* is a *total assignment* of *F* .

The SAT problem consists on determining whether *F* has a model. SAT(*F* ) denotes the set of models of *F* , then SAT(*F* ) *⊆ S*(*F* ). The set FAL(*F* ) = *S*(*F* ) *\ SAT* (*F* ) consists of the assignments from *S*(*F* ) that falsify *F* .

Clearly, for any propositional formula *F* , *S*(*F* ) = *SAT* (*F* ) *∪ FAL*(*F* ). The #SAT problem (or #SAT(*F* )) consists of counting the number of models of *F* defined over *υ*(*F* ), while #*FAL*(*F* ) denotes the number of falsifying assignments of

*F* . If *n* = *|υ*(*F* )*|* then #FAL(*F* )= 2*n*- #SAT(*F* ). #2SAT denotes #SAT for 2-CF formulas.

A Knowledge Base (KB) is a set *K* of formulas. Given a KB *K* and a propo- sitional formula *φ*, we say that *K* implies *φ*, and we write *K ▶ φ*, if *φ* is satisfied for each model of *K*, i.e., if *SAT* (*K*) *⊆ SAT* (*φ*). This last problem is known as the *propositional entail* problem. The incremental satisfiability problem (ISAT) con- sists in deciding if an initial knowledge Base *K* keeps its satisfiability anytime a conjunction of new clauses *φ* is added.

# Computing falsifying assignments of CF’s

Assume a KB *K* in CF, *K* = V*m Ci*, where each *Ci* is a clause, *i ∈* [[*m*]]. For each

*i*=1

clause *Ci*, *i ∈* [[*m*]], the assignment *s*(*Ci*) = 1 contains at least one literal in *Ci*. It is easy to build *FAL*(*K*) since each clause *Ci ∈ K* determines a subset of falsifying assignments of *K*. The following lemma expresses how to form the falsifying set of assignments of a CF.

**Lemma 1** *Given a CF K* = V*m*

*i*=1

*Ci, it holds that*

*m*

*FAL*(*K*)= *{σ ∈ S*(*K*) *| FAL*(*Ci*) *⊆ σ}*

*i*=1

**Lemma 2** *If a CF K is satisﬁable, then ∀Kj ⊆ K, Kj is a satisﬁable CF.*

*Ci∈K* *i*

**Proof.** If *K* is satisfiable, then *FAL*(*K*) = *FAL*(*C* ) *⊂ S*(*K*). Clearly, if we discard some clauses from *K*, forming *Kj*, then *FAL*(*Kj*) = *C ∈K′ FAL*(*Ci*)

*⊆* *Ci∈K*

*i*

*FAL*(*Ci*) *⊂ S*(*F* ). Thus, *Kj* is satisfiable. *2*

**Corollary 3.1** *If a CF K is unsatisﬁable, then ∀ CF Kj such that K ⊆ Kj, Kj*

*remains unsatisﬁable.*

*Ci∈K* *i*

**Proof.** An unsatisfiable CF *K* holds that *FAL*(*K*) = *FAL*(*C* ) = *S*(*F* ). Then, if we add new clauses to *K* forming *Kj*, then *FAL*(*K*)= *C ∈K FAL*(*Ci*) *⊆*

*Ci∈K′*

*i*

*FAL*(*Ci*)= *S*(*F* ). Thus, *Kj* is also unsatisfiable. *2*

Now, let us consider the propositional entail problem: *K ▶ φ*, where *K* and *φ* are CF’s. The decision problem *K ▶ φ* is a classical Co-NP-Complete problem for CF’s in general, since this problem is logically equivalent to the tautology problem for any DF, which is a classic co-NP-complete problem.

On the other hand, *K ▶ φ* iff *SAT* (*K*) *⊆ SAT* (*φ*), and this last goal is equivalent to prove *FAL*(*φ*) *⊆ FAL*(*K*), due to basic properties on sets that are closed under complementation.

**Lemma 3** *FAL*(*φ*) *⊆ FAL*(*K*) *if and only if K ▶ φ.*

The best known case of an efficient method for the inference *K ▶ φ* between CF’s is when both *K* and *φ* are Horn formulas. In this case, the application of SLD-resolution leads to a linear-time process for deciding *K ▶ φ*. The application of SLD-resolution has been the mechanism most commonly used in the development of logic programming languages [[8](#_bookmark16)].

However, including 2-CF’s as extensions of a Horn formula and continue ap- plying SLDS-resolution method as the inference engine, gives an exponential-time complexity process on the number of Horn inferences to perform.

For example, let *K∪H* be a formula where *K* is a Horn formula and *H* is a 2-CF formula, let *φ* be a Horn formula, if we want to decide *K ∪ H ▶ φ*, then we could apply the distributive property on each monotone positive binary clause (*x∨y*) *∈ H* and the Horn part *K*, then *K ∧* (*x∨y*) *▶ φ* if and only if (*¬*(*K ∧* (*x∨y*)) *∨φ*) is valid, and it holds iff ((*¬K ∨* (*¬x∧ ¬y*)) *∨ φ*) *≡* (((*¬K ∨ ¬x*) *∧* (*¬K ∨ ¬y*)) *∨ φ*) *≡* ((*¬*(*K ∧ x*) *∧¬*(*K ∧y*)) *∨φ*) *≡* (*¬*(*K ∧x*) *∨φ*) *∧* (*¬*(*K ∧y*) *∨φ*) *≡* ((*K ∧x*) *▶ φ*) *∧* ((*K ∧y*) *▶ φ*). Thus, for each positive monotone binary clause, we duplicate the number of Horn inferences to perform. If we consider the existence of two monotone binary clauses in *H*, that is (*K ∧* (*x*1*, y*1) *∧* (*x*2*, y*2)) *▶ φ*, and we apply the above process distributing the literals of the monotone clauses in every process of inference, then we obtain four Horn inferences given as: ((*K ∧ x*1 *∧ x*2) *▶ φ*), ((*K ∧ x*1 *∧ y*2) *▶ φ*),

((*K ∧ y*1 *∧ x*2) *▶ φ*) and ((*K ∧ y*1 *∧ y*2) *▶ φ*).

If there are *m* positive monotone binary clauses (*xi ∨ yi*)*,* 1 *≤ i ≤ m* in *H*, we have under the above reduction a total of 2*m* Horn inferences, which leads to an exponential-time complexity process on the number of Horn inferences to perform. Despite of the refutation methods commonly used in the Horn inference,

we consider here another method to determine whether *K ▶ φ*. When *K* = V*m Ci*

*i*=1

and *φ* = V*k ϕi*, our method focuses on checking that *FAL*(*φ*) *⊆ FAL*(*K*) in order

*i*=1

to prove *K ▶ φ*.

Each set *FAL*(*Ci*) can be represented in a succinct way via a string *Ai* of length *n* = *|υ*(*K*)*|*. Given a clause *Ci* = (*xi*1 *∨ ... ∨ xik* ), the value at each position from *i*1-th to *ik*-th of the string *Ai* is fixed with the truth value falsifying each literal of *Ci*. E.g., if *xij ∈ Ci*, the *ij*-th element of *Ai* is set to 0. On the other hand, if *xij ∈ Ci*, then the *ij*-th element is set to 1.

The variables in *υ*(*K*) which do not appear in *Ci* are represented by the symbol ’\*’ meaning that they could take any logical value in the set *{*0*,* 1*}*. In this way, the string *Ai* of length *n* = *|υ*(*K*)*|* represents the set of assignments falsifying the clause *Ci*. E.g. if *K* = *{C*1*,..., Cm}* is a 2-CF, *C*1 = (*x*1 *∨ x*2) and *C*2 = (*x*2 *∨ x*3), the assignments of *FAL*(*C*1) can be represented by the string 00 *∗∗ ... ∗* and the assignments of *FAL*(*C*2) are represented by *∗*01 *∗ ... ∗*.

We call *falsifying string* to the string *Ai* representing the set of falsifying as- signments of a clause *Ci*. We denote by *Fal String*(*Ci*), the string (with *n* sym- bols), that is the *falsifying string* for the clause *Ci*. As *K* and *φ* are CF’s, the falsifying strings of their clauses allow us to denote *FAL*(*φ*) and *FAL*(*K*). If *K /▶ φ ≡ FAL*(*φ*) */⊂ FAL*(*K*) implies that there exists a set of assignments *S* such that *S ⊆ FAL*(*φ*) and *S /⊂ FAL*(*K*). A reviewing procedure for *K ▶ φ* consists on taking each falsifying string representing *FAL*(*φ*) and reviewing if it is a subset of *FAL*(*K*).

⎛

*∀i ∈* [[*k*]] : ⎝*FAL*(*ϕi*) *⊆*

*j*=1

*m*

⎞

*FAL*(*Cj*)⎠ where *K* =

*i*=1

*m*

*Ci,φ* =

*i*=1

*k*

*ϕi* (1)

# An exact algorithm for *K ▶ φ*, when *K* and *φ* are CF’s

We present a method for checking *K ▶ φ*, with *K* and *φ* CF’s. Applying the *independence property* introduced by Dubois [[5](#_bookmark13)], we have designed a procedure to compute *FAL*(*φ*) *− FAL*(*K*), with *K* and *φ* CF’s.

**Definition 1** *Given two clauses C*1 *and C*2*, if they have at least one complemen- tary literal, it is said that they have the* independence *property. Otherwise, we say that the clauses are* dependent*.*

Notice that falsifying strings for independent clauses have complementary values (0 and 1) in at least one of their fixed values.

**Definition 2** *Let F* = *{C*1*, C*2*, ··· , Cm} be a CF. F is called independent if each pair of clauses Ci, Cj ∈ F, i /*= *j, have the independence property, otherwise F is*

*called dependent.*

Let *F* = *{C*1*, C*2*, ··· , Cm}* be a CF, *n* = *|υ*(*F* )*|*. Let *Ci,i ∈* [[*m*]] be a clause in

*F* and *x ∈ υ*(*F* ) *\ υ*(*Ci*) be any variable, we have that

*Ci ≡* (*Ci ∨ ¬x*) *∧* (*Ci ∨ x*) (2)

**Definition 3** *Given a pair of dependent clauses C*1 *and C*2*, if Lit*(*C*1) *⊆ Lit*(*C*2)

*we say that C*2 *is subsumed by C*1*.*

If *C*1 subsumes *C*2 then *FAL*(*C*2) *⊆ FAL*(*C*1). On the other hand, if *C*2 is not subsumed by *C*1 and they are dependents, there is a set of indices *I* = *{*1*,..., p}⊆*

*{*1*,..., n}* such that for each *i ∈ I, xi ∈ C*1 but *xi /∈ C*2. There exists a reduction to transform *C*2 to become independent from *C*1, we call this transformation as the *independent reduction* between two clauses that works as follows: let *C*1 and *C*2 be two dependent clauses. Let *{x*1*, x*2*,..., xp}* = *Lit*(*C*1) *\ Lit*(*C*2). By ([2](#_bookmark2)) we can write: *C*1 *∧ C*2 *≡ C*1 *∧* (*C*2 *∨ ¬x*1) *∧* (*C*2 *∨ x*1). Now *C*1 and (*C*2 *∨ ¬x*1) are independent. Applying ([2](#_bookmark2)) to (*C*2 *∨ x*1):

*C*1 *∧ C*2 *≡ C*1 *∧* (*C*2 *∨ ¬x*1) *∧* (*C*2 *∨ x*1 *∨ ¬x*2) *∧* (*C*2 *∨ x*1 *∨ x*2)

The first three clauses are independent. Repeating the process of making the last clause independent with the previous ones, until *xp* is considered; we have that *C*1 *∧ C*2 can be written as:

*C*1 *∧*(*C*2 *∨¬x*1)*∧*(*C*2 *∨x*1 *∨¬x*2)*∧...∧*(*C*2 *∨x*1 *∨x*2 *∨...∨¬xp*)*∧*(*C*2 *∨x*1 *∨x*2 *∨...∨xp*).

The last clause contains all literals of *C*1, so it is subsumed by *C*1, and then

*C*1 *∧ C*2 *≡ C*1 *∧* (*C*2 *∨ ¬x*1) *∧* (*C*2 *∨ x*1 *∨ ¬x*2) *∧ ... ∧* (*C*2 *∨ x*1 *∨ x*2 *∨ ... ∨ ¬xp*) (3)

We obtain on the right hand side of ([3](#_bookmark3)) an independent set of *p* + 1 clauses which we denote as *indep reduction*(*C*1*, C*2). We use the independent reduction between two clauses *C* and *ϕ* (or between their respective falsifying strings) to define:

⎧ *ϕ* if *ϕ* and C are independent

⎪⎪⎨

*Ind*(*C, ϕ*)= *∅* if Lit(*C*) *\* Lit(*ϕ*)= *∅*

⎪⎪⎩ *indep reduction*(*C, ϕ*) *− C* Otherwise

It is straightforward to redefine the operator *Ind* in terms of the falsifying strings representing *FAL*(*C*) and *FAL*(*ϕ*). The operation *Ind*(*C, ϕ*) forms a conjunction of clauses whose falsifying assignments are exactly *FAL*(*ϕ*) *− FAL*(*C*).

**Theorem 1** *If ϕ and C are two clauses, then FAL*(*Ind*(*C, ϕ*)) = *FAL*(*ϕ*) *−*

*FAL*(*C*)

**Proof.** If *Ind*(*C, ϕ*) = *∅* then *FAL*(*ϕ*) *⊆ FAL*(*C*), so *FAL*(*ϕ*) *\ FAL*(*C*) =

*∅*. Now, we assume that *Ind*(*C, ϕ*) */*= *∅*. Let *s* be an assignment such that

*s ∈ FAL*(*Ind*(*C, ϕ*)). We will show that *s ∈ FAL*(*ϕ*) and *s ∈/ FAL*(*C*). If

*s ∈ FAL*(*Ind*(*C, ϕ*)) then *s* falsifies *ϕ* because each clause in *Ind*(*C, ϕ*) has the form (*ϕ ∨ R*), where *R* is a disjunctive set of literals (possibly *R* is empty). If *s* falsifies (*ϕ ∨ R*) then *s* has to falsify *ϕ* and thus *s ∈ FAL*(*ϕ*). On the other hand, each clause (*ϕ∨R*) *∈ Ind*(*C, ϕ*) is independent to *C* by construction of the operator *Ind*; therefore, *FAL*(*C*) *∩ FAL*(*Ind*(*C, ϕ*)) = *∅*. Furthermore, *s ∈/ FAL*(*C*). *2 2*

Let *K* = V*m Cj* be a CF and *ϕ* be a clause. If we apply the *Ind* operator

*j*=1

between each *Cj ∈ K* and *ϕ*, we get as a result a set *S* such that *S ⊆ FAL*(*ϕ*) and

*S /⊂ FAL*(*K*).

In order to generate a mimimun set of independent clauses as a result of *Ind*(*K, ϕ*), it is crucial to sort the clauses *Cj ∈ K* according to the length *|Sj|* = *|Lit*(*Cj*) *\ Lit*(*ϕ*)*|* in ascending order, because the number of literals in *Cj*, different to the literals in *ϕ*, determines the number of independent clauses to be generated.

The operator *Ind* applied on the clause *ϕ* and on each one of the clauses *Cj ∈ K*, allow us to build the space *FAL*(*ϕ*) *− FAL*(*K*). Thus, the following recurrence is defined as: *A*1 = *ϕ*, *Aj*+1 = *Ind*(*Cj, Aj*))*,j* = 1*,..., m*. The algorithm ([1](#_bookmark4)) performs the computation of this last recurrence, while it checks if any *Aj*+1 is empty, in whose case *K ▶ ϕ* will hold.

In order to perform *Ind*(*Cj, Aj*), the remaining clauses in *Cl ∈ K, l* = *j* + 1*,..., m*, those which are not reducted independently with *Aj*, are sorted again in ascendent order according to the number of common literals with the literals represented by *Aj*. This process can be extended to each *ϕi ∈ φ, i* = 1*,..., k*, as:

*Ai,*1 = *ϕi*

*Ai,j*+1 = *Ind*(*Cj, Ai,j*)*,j* = 1*,..., m,* and *i* = 1*,...,k*

being so constructed clauses *Ai,m*+1 such that *k* (*Ai,m*+1)= *FAL*(*φ*) *− FAL*(*K*).

*i*=1

These strings *Ai,j,i* = 1*,..., k, j* = 1*,...,m* form a matrix of strings, as it is illustrated in Table 1. Notice that if *Ai,j* = *∅* then *Ai,l* = *∅*, for *l* = *j* + 1*,..., m*.

Example: let *K* be an initial KB, *K* = *{*(*x*1*, x*2)*,* (*x*1*, x*7)*,* (*¬x*1*, x*7)*,* (*¬x*2*, x*3)*,* (*x*3*,*

*¬x*4)*,* (*x*4*, ¬x*5)*,* (*x*5*, ¬x*6)*,* (*x*6*, x*7)*}* and *φ* = *{*(*¬x*3*, x*6)*,* (*x*2*, ¬x*6*, x*7)*,* (*x*1*, x*4*, x*5)*}*. In each cell of the Table 1, the result of *Ind*(*Cj, Ai,j*) is shown, until determin- ing if *K ▶ ϕi,i* = 1*,...,* 3.

|  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- |
| zzz *K*  *ϕ*1 zz | \*\*01\*\*\* | \*10\*\*\*\* | \*\*\*\*01\* | \*\*\*\*\*00 | 1\*\*\*\*\*0 | 0\*\*\*\*\*0 | 00\*\*\*\*\* | \*\*\*01\*\* |
| z\*\*1\*\*0\* | \*\*1\*\*0\* | \*\*1\*\*0\* | \*\*1\*\*0\* | \*\*1\*\*01 | \*\*1\*\*01 | \*\*1\*\*01 | *K b ϕ*1 |  |
| zz *K*  *ϕ*2 zz | \*\*\*\*\*00 | 00\*\*\*\*\* | 0\*\*\*\*\*0 | 1\*\*\*\*\*0 | \*\*\*\*01\* | \*10\*\*\*\* | \*\*01\*\*\* | \*\*\*01\*\* |
| z\*0\*\*\*10 | \*0\*\*\*10 | 10\*\*\*10 | 10\*\*\*10 | *K ▶ ϕ*2 |  |  |  |  |
| zz *K*  *ϕ*3 zz | \*\*\*01\*\* | \*\*01\*\*\* | 1\*\*\*\*\*0 | 00\*\*\*\*\* | 0\*\*\*\*\*0 | \*10\*\*\*\* | \*\*\*\*01\* | \*\*\*\*\*00 |
| 0\*\*00\*\* | 0\*\*00\*\* | 0\*\*00\*\* | 0\*\*00\*\* | 01\*00\*\* | 01\*00\*1 | 01100\*1 | 0110001 | 0110001 |

Table 1 Computing *Ind*(*K, φ*)

**Algorithm 1** Procedure InferenceCFCF(*K, ϕ*)

Input: *K*: A knowledge base in CF, *ϕ*: a clause that is a new knowledge Output: True or False according to *K ▶ ϕ* or *K /▶ ϕ*

Push(*ϕ, Stack*); *Fs* = *∅*;

**for all** *Cj ∈ K* **do while** (*Stack /*= *∅*) **do**

*ϕ* = *Pop*(*Stack*); *{*test next *ϕ* that has been previously computed*}*

*Fs* = *Fs − ϕ*; *A* = *Ind*(*Cj, ϕ*);

**if** (*A /*= *∅*) **then**

*Fs* = *Fs ∪ A*; *{*Only if there are new clause to be aggregated*}*

## end if end while

**if** (*Fs* = *∅*) **then**

Returns(true);

## end if

*Stack* = *Fs*; *{*new set of clauses to be considered in the next iteration*}*

## end for

Returns(False) *{K b ϕ}*

# The Transitive Closure of a 2-CF

The fact that in a 2-CF formula a clause is equivalent to a pair of implications can be straightforward established as follows: if *{x, y} ∈ F* then *{x, y}* is equivalent to both *x → y* and *y → x*. The arrow *→* has the usual meaning of implication in classical logic. By abuse of notation, the arrow *→* will be also used to denote a relation between literals as established in definition [4](#_bookmark5).

**Definition 4** *Let F be a 2-CF and L its set of literals. The relation →R⊂ L × L*

*is deﬁned as follows: x →R y if and only if x → y.*

**Definition 5** *Let F be a 2-CF, a partial assignment s of F is a feasible model for*

*F, if s does not falsify any clause in F.*

In principle, the relation above is too general to work with so it will be taken the transitive closure of *→R*, denoted by ”*⇒*”, instead. The new relation *⇒* can always be constructed inductively from *→R*. For any feasible model *s* of *F* where *x* and *y* occur in *F* ; if *x ⇒ y* and *x* is true in *s* then it is straightforward to show that *y* is true in *s*. Under these circumstances, it is said that *y* is *forced* to be true by *x*. Let *T* (*x*) be the set of literals forced to be true by *x*, that is

*T* (*x*)= *{x}∪ {y* : *x ⇒ y}* (4)

It is clear that, if *x* is a literal occurring in a formula *F* , and if *x*¯ *∈ T* (*x*) then *x* cannot be set to true in any model of *F* . Analogously, if *x ∈ T* (*x*¯) then *x* cannot be set to false in any model of *F* .

Given formulas *X* and *Y* , it is said that *X ≡ Y* (or *X* is equivalent to *Y* ) whenever *X ↔ Y* in classical logic. It is straightforward to show that *x → y ≡ y →*

*x*. Hence, if *y ∈ T* (*x*) then *x ∈ T* (*y*).

For any literal *x* in a 2-CF *F* , *T* (*x*) can be classified as consistent or inconsistent.

Formally,

**Definition 6** *Let F be a 2-CF, for any literal x ∈ F, it is said that T* (*x*) *is incon- sistent if x ∈ T* (*x*) *or ⊥∈ T* (*x*)*, otherwise T* (*x*) *is said to be consistent.*

Unit clauses in 2-CF can be expressed as implications, that is, if *F* has unit clauses *{u}* then *u ≡ u ∨⊥* hence *⊥ ∈ T* (*u*). As a consequence, in formulas with unit clause *{u}* follows that *T* (*u*) is inconsistent. Let *F* be a 2-CF with *n* variables and *m* clauses, it has been shown that for any literal *x ∈ F* , *T* (*x*) and *T* (*x*) are computed in polynomial time over *|F |*, in fact, for all *l ∈ Lit*(*F* ), *T* (*l*) is computed with time complexity *O*(*n · m*) [[9](#_bookmark17)].

For any literal *x* in a 2-CF, the sets *T* (*x*) and *T* (*x*) allow to determine which variables have a fixed logical values in every model of *F* , that is to say, the variables that are true in every model of *F* and the variables that are false in every model of

*F* . The properties of the sets *T* (*x*) and *T* (*x*) will be established as a lemma.

**Lemma 5.1** *Let F be a 2-CF and x a variable in F.*

1. *If T* (*x*) *is inconsistent and T* (*x*) *is consistent then x is true in every model of*

*F.*

1. *If T* (*x*) *is inconsistent and T* (*x*) *is consistent then x is true in every model of*

*F.*

1. *If both T* (*x*) *and T* (*x*) *are inconsistent then F does not have models and F is unsatisiﬁable.*
2. *If both T* (*x*) *and T* (*x*) *are consistent then x does not have a ﬁxed valued in each model of F.*

## Proof.

1. Suppose *x* is false in a model of *F* , so *x* should be true in that model of *F* , however, *T* (*x*) is inconsistent so *x ⇒ x* and *x* cannot be true in the model of *F* contradicting the assumption. Hence, any model of *F* has to assign false to *x* and true to *x*. The other cases are proved similarly.

*2*

From properties (1) and (2) of lemma [5.1](#_bookmark6) we formulate the following definition

**Definition 7** *A base for the set of models of a 2-CF F, denoted as S*(*F* )*, is a partial assignment s of F which consists of the variables with a ﬁxed truth value.*

We denote by *Transitive Closure(F)* to the procedure which computes the sets *T* (*x*) and *T* (*x*¯) for each *x ∈ υ*(*F* ). The transitive procedure applied on a 2-CF *F* allows to build bases for the set of models of *F* . If a base *S*(*F* ) is such that

*|S*(*F* )*|* = *|υ*(*F* )*|* then each variable of *F* has a fixed truth value in every model of *F* , so there is just one model. Similarly, if #SAT(*F* )= 0, *Transitive Closure*(*F* ) finds at least a variable *x ∈ υ*(*F* ) such that *T* (*x*) and *T* (*x*) are inconsistent. So, when #*SAT* (*F* ) *≤* 1 such value is computed in polynomial time by *Transitive Closure*(*F* ).

**Definition 8** *Let F be a 2-CF and x a literal of F. The reduction of F by x, also called forcing x and denoted by F* [*x*]*, is the formula generated from F by the following two rules*

1. removing from *F* the clauses containing *x* (subsumption rule),
2. removing *x* from the remaining clauses (unit resolution rule).

A reduction is also sometimes called a *unit reduction*. The reduction by a set of literals can be inductively established as follows: let *s* = *{l*1*, l*2*,..., lk}* be a partial assignment of *υ*(*F* ). The reduction of *F* by *s* is defined by successively applying definition [8](#_bookmark7) for *li*, *i* = 1*,..., k*. That is reduction of *F* by *l*1 gives the formula *F* [*l*1], following a reduction of *F* [*l*1] by *l*2, giving as a result the formula *F* [*l*1*, l*2] and so on. The process continues until *F* [*s*] = *F* [*l*1*, ..., lk*] is reached. In case that *s* = *∅* then *F* [*s*]= *F* .

**Example 5.2** Let *F* = *{{x*1*, x*2*}, {x*1*, x*2*}, {x*1*, x*3*}, {x*1*, x*3*}, {x*2*, x*4*}, {x*2*, x*4*}, {x*2*,*

*x*5*}, {x*3*, x*5*}}*. If *s* = *{x*2*, x*3*}*, *F* [*x*2]= *{{x*1*}, {x*1*, x*3*}, {x*1*, x*3*}, {x*4*}, {x*4*}, {x*3*,*

*x*5*}}*, and *F* [*s*]= *{{x*1*}, {x*1*}, {x*1*}, {x*4*}, {x*4*}, {x*5*}}*.

Let *F* be a 2-CF formula and *s* a partial assignment of *F* . If a pair of contradic- tory unitary clauses is obtained while *F* [*s*] is being computed then #*SAT* (*F* [*s*]) = 0. Because under no circumstances, a pair of complementary unit clauses can be set to true at the same time. Thus, *F* [*s*] does not have models.

Furthermore, during the computation of *F* [*s*] new unitary clauses can be gen- erated. Thus, the partial assignment *s* is extended by adding the unitary clauses found, that is, *s* = *s ∪ {u}* where *{u}* is a unitary clause. So, *F* [*s*] can be again reduced using the new unitary clauses. The above mentioned iterative process is generalized, and we call to this iterative process *U nit Propagation*(*F, s*). For sim- plicity, we will abbreviate *U nit Propagation*(*F, s*) as *UP* (*F, s*), where *F* is a CF and *s* is the set of literals belonging to unit clauses of *F* .

# Incremental Satisfiability Problem

The incremental satisfiability problem (ISAT) involves checking whether satisfia- bility is maintained when new clauses are added to an initial satisfiable knowledge base *K*. ISAT is considered as a generalization of SAT since it allows changes of the input formula over the time, and also, it can be considered as a prototypical Dynamic Constraint Satisfaction Problem (DCSP) [[13](#_bookmark20)].

Different methods have been applied to solve ISAT, among them, branch and bounds procedures as variants of the classical Davis-Putnam-Loveland (DPL) method, denoted as IDPL methods. In IDPL procedures, when adding new clauses, those procedures maintain the search tree generated previously for the set of clauses *K*. IDPL performs substantially faster than DPL for a large set of SAT problems [[10](#_bookmark18)].

As a generalization of SAT, ISAT has been considered as an NP Problem, al- though until now, we have not seen complexity theory studies about the complexity- time differences between SAT and ISAT. For example, it is known that 2-SAT is

in the complexity class P, however it is not known the computational complexity of 2-ISAT. It is clear that a set of changes over a satisfiable KB *K* in 2-CF could change *K* into a general CF, in whose case, it turns in a general CF *KJ*, *K ⊂ KJ*, and where the SAT problem on *KJ* is a classic NP-complete problem.

Rather than solving related formulas separately, modern solvers attempt to solve them *incrementally* since many practical applications require solving a sequence of related SAT formulas [[3](#_bookmark11),[6](#_bookmark14)]. For example, in [[16](#_bookmark22)] Clause-Sharing *CS* marks all the conflict clauses that depend on assumptions and discards them before the next incremental step. Consequently, the generated proof obligations are solved by an incremental SAT-based SMT solver. We present in this section, an study about the threshold for the 2-ISAT problem that could be relevant to understand the border between P and NP complexity classes.

Assuming an initial KB *K*, and a new CF *φ* to be added, both are satisfiables CF’s, let us consider some cases where ISAT can be determined directly.

* 1. If *K* and *φ* are 2-CF’s then (*K ∧ φ*) is a 2-CF that is the input of ISAT, and in this case, 2-ISAT is solvable in linear-time by appying the well known algorithms for 2-SAT [[9](#_bookmark17),[2](#_bookmark10)]
  2. If *φ* consists of exactly one clause and we have the satisfiability tree of *K*, we only have to review which satisfiable branches of the tree falsify *φ*, and this can be done in linear time on the number of satisfiable branches of the tree.
  3. For monotone formulas, ISAT keeps satisfiable formulas. If each variable main- tains an unique sign in both *K* and *φ* then (*K ∧ φ*) is always satisfiable.

Let us consider now that *K* is a 2-CF and *φ* is a general CF, both of them different from the previous cases. We present Algorithm [2](#_bookmark8) which takes as inputs a satisfiable 2-CF formula *F* and a satisfiable CF formula *φ* and it determines whether (*K ∧ φ*) is satisfiable.

By the results presented in Section [4](#_bookmark1), each *ϕi ∈ φ* such that *K ▶ ϕi* is removed from *φ*, so we assume that *φ* = (*φ − ϕi*). It means, we will consider only the clauses in *φ* which decrease effectively the set of models of *K*. Assume that the computation of both *T* (*x*) and *Bi* = *FAL*(*ϕi*) have been computed for each *x ∈ Lit*(*K*) and each *ϕi ∈ φ*, respectively.

Algorithm ([2](#_bookmark8)) proposed for reviewing the satisfactibility of (*K ∧ φ*) is based on the following properties:

1. Given the partial assignments *A*1, *A*2 which they are part of any model (if there exists) of *K*. Those partial assignments may be extended in a way that they do not falsify any *ϕi ∈ φ*, which is verified by *Ind*(*Aj, ϕi*)*,j* = 1*,* 2. If it is possible, then a model for (*K ∧ φ*) is built.
2. Otherwise, *Ind*(*Aj, ϕi*) = *∅,j* = 1*,* 2 and any model of *K* will be part of any falsifying assignment of *φ*. Thus, (*K ∧ φ*) is unsatisfiable.

The first property establishes that a partial assignment *s* which is part of any model of *SAT* (*K*) has been built and also it is not part of any falsifying string of *ϕi, ∀ϕi ∈ φ*. Thus, *s* satisfies (*K ∧ φ*). The second property establishes the

**Algorithm 2** Procedure that determines whether (*K ∧ φ*) is satisfiable. Inputs a 2-CF formula *F* and a CF *φ*

1: **for** each *x ∈ Lit*(*F* ) and *ϕi ∈ φ* **do** 2: computes *T* (*x*) and *Bi* = *FAL*(*ϕi*) 3: **end for**

4: Add *⊥* to each consistent *T* (*x*) only if *Bi ⊆ T* (*x*) for some *Bi {*That means, makes inconsistent any partial assignment which falsifies some *ϕi ∈ φ}*

5: *A* = *∅*

6: **for** each inconsistent *T* (*x*) **do**

7: *A* = *A ∪ T* (*¬x*)*{*That means the maximum partial assignment satisfying *K}*

8: **end for**

9: **if** *A* is inconsistent or there is an *x ∈ Lit*(*K*), such that *T* (*x*) and *T* (*¬x*) are inconsistent **then**

10: (*K ∧ φ*) is unsatisfiable

11: **else**

12: *φ* = *UP* (*φ, A*)*{*That means, any satisfying clause for the partial assignment

*A* is deleted from *φ*.*}*

13: **end if**

14: **if** *φ* = *∅* **then**

15: (*K ∧ φ*) is satisfiable

16: **else**

17: **let** (*l ∈ Lit*(*K*) such that both *T* (*l*) and *T* (*¬l*) are consistent)

18: *A*1 = *A ∪ T* (*l*)

19: *A*2 = *A ∪ T* (*¬l*)

20: **for** *ϕi ∈ φ* **do**

21: *Aj* = *Ind*(*A*1*, ϕi*)

22: *Ajj* = *Ind*(*A*2*, ϕi*)

23: **end for**

24: **end if**

25: **if** *Aj* = *∅* and *Ajj* = *∅* **then** 26: (*K ∧ φ*) is unsatisfiable 27: **else**

28: (*K ∧ φ*) is satisfiable

29: **end if**

unsatisfiability of (*K ∧ φ*).

In algorithm ([2](#_bookmark8)), when the operation *Aj* = *Ind*(*A*1*, ϕi*) is reached (step 21), a new literal *x* will be joined to a superstring of *A*1. In fact, we consider to join *T* (*x*) instead of *x*. For example, if *A*1 = *∗*01 *∗∗∗* 1*∗* and *ϕ*1 = *∗*0 *∗* 010 *∗ ∗*, *Ind*(*A*1*, ϕi*) gives asa result *∗*011 *∗ ∗*1*∗*, *∗*0100 *∗* 1*∗* and *∗*010101*∗*. However, in this case it means to build the following three parcial assignments: *A*1 *∪T* (*x*4), *A*1 *∪T* (*¬x*4) *∪T* (*¬x*5), and *A*1 *∪ T* (*¬x*4) *∪ T* (*x*5) *∪ T* (*¬x*6). If any of those strings is inconsistent then such string is substituted by *∅*.

Let us analyze the growth on the number of possible partial assignments of the operation: *Ind*(*A*1*, ϕi*), *∀ϕi ∈ φ*. Firstly, the number of partial assignments for

a fixed *ϕi* is *|S|* = *|υ*(*ϕi*) *− υ*(*A*1)*|*. Moreover, each partial assignment *si ∈ S* is independent to the other assignments in *S*, because they are different in at least one literal, and each of them hold *|si*+1*|≥ |si|* + 1*, ∀si ∈ S*.

Although *Ind*(*A, ϕi*) is computed in linear time on the size of both strings, the computational complexity of the algorithm ([2](#_bookmark8)) depends on the number of strings generated by *Ind*(*A, ϕi*)*, ∀ϕi ∈ φ*.

In some cases, *Ind*(*A, ϕi*) may generate empty sets. However, in the worst case, the time complexity depends on the cardinality of the sets *Li* = *{x*1*,..., xp}*

= *lit*(*ϕi*) *− lit*(*A*)*,i* = 1*,..., k*.

In order to improve the time complexity of our procedure, it is convenient to sort the clauses *ϕi ∈ φ* according to the cardinality of the sets *Li*, *i* = 1*,...,k* from the smallest to the biggest and removing the clauses that are independent with *A*. Once the clauses are sorted in *φ* with respect to their cardinalities *Li*, the operation *Ind*(*Ind*(*....Ind*(*Ind*(*A, ϕ*1)*, ϕ*2)*,..., ϕk*) is applied, determining so the succession: *S*0 = *v*(*A*)

*S*1 = *v*(*ϕ*1) *− v*(*A*)

*S*2 = *v*(*ϕ*2) *−* (*v*(*ϕ*1) *∪ v*(*A*))

*...*

*Sk* = *v*(*ϕk*) *−* (*v*(*ϕk−*1) *∪ ... ∪ v*(*ϕ*1) *∪ v*(*A*))

Then, the number of new clauses in the worst case is given by:

*k*

*| Ind*(*A, φ*) *|≤* *| Si |*=*| S*1 *|∗| S*2 *| ∗ ... ∗| Sik | .* (5)

*i*=1

It is clear that the number of strings in *|Ind*(*A, φ*)*|* is not bigger than the number of assignments in *SAT* (*K*) *− FAL*(*φ*) since the falsifying assignments of *ϕi ∈ φ* are removed from the partial assignment denoted by the string *A*. That means *|S*1*| ∗*

*|S*2*| ∗ ... ∗ |Sk|∈ O*(*|SAT* (*K*) *− FAL*(*φ*)*|*). From this last upper bound, we can infer some tractable cases for ISAT, as the following theorem establishes.

**Theorem 6.1** *Let K be a 2-CF formula and φ a CF formula. The following holds:*

1. *If |SAT* (*K*)*| ≤ poly*(*n*) *then ISAT is solved in polynomial time. In fact, we can have the set of models S explicitly and each model s ∈ S can be checked: φ*[*s*]*.*
2. *If |SAT* (*K*) *− FAL*(*φ*)*|≤ poly*(*n*) *then the number of strings in |Ind*(*A, φ*)*| is upper bounded by |SAT* (*K*) *− FAL*(*φ*)*|≤ poly*(*n*)*.*
3. *When φ*[*T* (*x*)] *is false for all consistent T* (*x*)*, algorithm (*[*2*](#_bookmark8)*) ﬁnds the unsatisﬁ- ability of* (*K ∧φ*) *in polynomial time on the set of literals of K and the number of clauses of φ. Consequently, this determines a tractable case for the 2-ISAT problem.*

# Conclusions

We proposed a novel method to review *K ▶ φ* when *K* and *φ* are both in Conjunctive Normal Forms. This initial method is extended to consider the incremental satis- fiability (ISAT) problem. We have shown different cases where the ISAT problem can be solved in polynomial time.

Especially, we have designed an algorithm for solving the 2-ISAT problem that allows us to detemine an upper bound for the time-complexity of this problem. Furthermore, we have established some tractable cases for the 2-ISAT problem.

# References

1. Barrett, C., Stump, A., Tinelli C., The SMT-LIB standard version 2.0, 2010. Available from: [http://www.smtlib.org.](http://www.smtlib.org/)
2. Buresh-Oppenheim J., Mitchell D., *Minimum 2CNF resolution refutations in polynomial time*, Proc. SAT’07 - 10th int. Conf. on Theory and applications of satisfiability testing, (2007), pp.300-313.
3. Cabodi G., Lavagno L., Murciano M., Kondratyev A., Watanabe Y., *Speeding-up heuristic allocation, scheduling and binding with SAT-based abstraction/refinement techniques*, ACM Trans. Design Autom. Electr. Syst., 15(2), (2010).
4. Creignou N., Papini O., Pichler R., Woltran S., *Belief Revision within fragments of propositional logic*, DBAI Tech. Report DBAI-TR-2012-75, (2012).
5. Dubois O., Counting the number of solutions for instances of satisfiability, *Theor. Comp. Sc.* 81, (1991), pp.49-64.
6. E´en N., S orensson K., *An Extensible SAT-solver*, In Enrico Giunchiglia and Armando Tacchella, editors, Selected Revised Papers of 6th International Conference on Theory and Applicationsof Satisfiability Testing (SAT), Santa Margherita Ligure, Italy, LNCS Vol. 2919, (2003), pp. 502-518.
7. E´en N., S orensson K., *Temporal induction by incremental SAT solving*, In Procs. of First Int. Workshop on Bounded Model Checking (BMC), volume 89 of Electronic Notes in Theoretical Computer Science, (2003), pp. 543-560.
8. Gallier J., *Logic for Computer Science: Foundations of Automatic Theorem Proving (chapter 9)*, (2003), online revision (free to download).
9. Gusfield, D., Pitt, L., A Bounded Approximation for the Minimum Cost 2-Sat Problem, *Algorithmica* *8*, (1992), pp. 103-117.
10. Hooker J.N., Solving the incremental satisfiability problem. *Journal of Logic Programming 15*, (1993), pp.177-186.
11. Katsuno, H. & Mendelzon, A. O., *On the difference between updating a knowledge base and revising it*, KR’91 Cambridge, MA. USA, (1991), pp. 387–394.
12. Khardon R., Roth D.: Reasoning with Models, Artificial Intelligence 87, No.1, (1996), pp. 187-213.
13. Gutierrez J., Mali A., *Local search for incremental Satisfiability*, Proc. Int. Conf. on AI (IC-AI’02), Las Vegas, (2002), pp. 986-991.
14. Mahajan Y S., Zhaohui F., Sharad M. *ZChaff2004: An Efficient SAT Solver*, In Holger H. Hoos and David G. Mitchell, editors, Revised Selected Papers of 7th Int. Conf. on Theory and Applications of Satisfiability Testing (SAT), Vancouver, Canada, LNCS Vol. 3542, (2004), pp. 360-375.
15. Marchi J., Bittencourt G., Perrusssel L., Prime forms and minimal change in propositional belief bases,

*Ann. Math. Artif. Intelligence* 59(1),(2010), pp.1–45.

1. Nadel A., Ryvchin V., Strichman O., *Ultimately Incremental SAT*, Proc. SAT 2014, LNCS Vol. 8561, (2014), pp. 206218.
2. Shankar N., *Mathamathematics, Machines, and Godels Proof*, Cambridge Tracks in Theoretical Computer Science No. 38, Cambridge University Press, (1997).
3. Whittemore J., Joonyoung K., Sakallah K.A. *SATIRE: A New Incremental Satisfiability Engine*, In Proceedings of the 38th Design Automation Conference (DAC)- ACM, Las Vegas - USA, (2001), pp. 542-545.
4. Wieringa S., *Incremental Satisfbiability Solving and its Applications*, PhD thesis, Department of Computer Science and Engineering, Aalto University Pub., (2014).