AKADEMIA GÓRNICZO-HUTNICZA IM. STANISŁAWA STASZICA W KRAKOWIE

WYDZIAŁ INFORMATYKI, ELEKTRONIKI I TELEKOMUNIKACJI

KATEDRA INFORMATYKI

PRACA DYPLOMOWA MAGISTERSKA

MODELING SELECTED COMPUTATIONAL PROBLEMS AS SAT-CNF AND ANALYZING STRUCTURAL PROPERTIES OF OBTAINED FORMULAS

MODELOWANIE WYBRANYCH PROBLEMÓW OBLICZENIOWYCH PRZEZ FORMUŁY CNF I ANALIZA ICH WŁASNOŚCI STRUKTURALNYCH

Autor: Michał Mrowczyk

Kierunek studiów: Informatyka

Opiekun pracy: Prof. Dr. Piotr Faliszewski

Kraków, 2016

Uprzedzony o odpowiedzialności karnej na podstawie art. 115 ust. 1 i 2 ustawy z dnia 4 lutego 1994 r. o prawie autorskim i prawach pokrewnych (t.j. Dz.U. z 2006 r. Nr 90, poz. 631 z późn. zm.): „ Kto przywłaszcza sobie autorstwo albo wprowadza w błąd co do autorstwa całości lub części cudzego utworu albo artystycznego wykonania, podlega grzywnie, karze ograniczenia wolności albo pozbawienia wolności do lat 3. Tej samej karze podlega, kto rozpowszechnia bez podania nazwiska lub pseudonimu twórcy cudzy utwór w wersji oryginalnej albo w postaci opracowania, artystyczne wykonanie albo publicznie zniekształca taki utwór, artystyczne wykonanie, fonogram, wideogram lub nadanie.", a także uprzedzony o odpowiedzialności dyscyplinarnej na podstawie art. 211 ust. 1 ustawy z dnia 27 lipca 2005 r. Prawo o szkolnictwie wyższym (t.j. Dz. U. z 2012 r. poz. 572, z późn. zm.) „Za naruszenie przepisów obowiązujących w uczelni oraz za czyny uchybiające godności studenta student ponosi odpowiedzialność dyscyplinarną przed komisją dyscyplinarną albo przed sądem koleżeńskim samorządu studenckiego, zwanym dalej „sądem koleżeńskim", oświadczam, że niniejszą pracę dyplomową wykonałem(-am) osobiście i samodzielnie i że nie korzystałem(-am) ze źródeł innych niż wymienione w pracy.

Contents

Abstract 1

1. Introduction 3

2. Preeliminaries 5

2.1. The Boolean Satisfiability and CNF 5

2.2. The Integer Factorization Problem 8

2.3. The OWA-Winner Problem 8

2.4. Basic Notions and Definitions Used to Express Boolean Constraints . 10

3. Reducing Selected Computational Problems to SAT-CNF 11

3.1. Reducing Integer Factorization to SAT-CNF 11

3.2. Reducing OWA-Winner to SAT-CNF 17

4. Experimental Analysis of Structure of Obtained Formulas 23

4.1. Clauses to Variables Ratio 23

4.2. Running Time 27

Acknowledgments 29

A. Appendix 31

A.1. Overview 31

A.2. The next section 31

Bibliography 33

Nomenclature 35

Abstract

The boolean satisfiability was the first computational problem to be proven NP complete. The proof of this fact was established independently by Stephen Cook and Leonid Levin over 40 years ago. Since then numerous problems were shown to be NP complete. Nevertheless, boolean satisfiability (SAT) arguably still has remained the most fundamental NP complete problem out there. It is possible to convert all problems in NP to SAT by using polynomial time reductions. In this thesis I provide step by step description of reduction from OWA-Winner problem (to be precise it's decision version) to SAT-CNF. In order to do this I investigate known techniques of reducing Integer Factorization to SAT-CNF and encoding boolean cardinality constraints. Having reduced both Integer Factorization and OWA-Winner problems to SAT-CNF I consider experimental ways of exploring the structure of obtained boolean formulae instances.

1. Introduction

Boolean Satisfiability problem (SAT) is a decision problem [1]where we are given a logical formula F over some variables and we ask if there is a satisfying assignment for it. Satisfying assignment simply means an assignment of truth values to the variables that evaluates to truth. *SAT* was the very first problem to be proven NP-complete [Coo71] and remains one of the most frequently studied problems in computational complexity theory. Although finding satisfying truth assignments or proving unsatisfiability seems to be hard in general, there are tools—solvers (PicoSAT, MiniSat, Glucose, Lingeling, etc.)—that can deal with really large instances in practice. Solving *SAT* is not only a theoretical challenge. There are a lot of practical applications that can be modeled using boolean functions. Examples of such problems in electronic design automation (EDA) include formal equivalence checking, model checking, formal verification of pipelined microprocessors [BGV99], automatic test pattern generation [Lar], routing of FPGAs [NSR02], planning [Kau], and scheduling [HZS] problems. In this thesis we consider *Integer Factorization* problem and we show the way of reducing it to *SAT*. The purpose of this is to obtain a set of similarly-structured[2] *SAT* instances and inspect their properties. We also consider *OWA-Winner* problem (an optimization problem in the election and voting theory) and the way of reducing this problem to *SAT*. We want to evaluate the performance of modern *SAT* solvers on this particular problem instances.

[1]A decision problem is a problem with a YES/NO answer. In formal languages theory, such a problem can be viewed as a formal language containing strings (problem instances) for which the answer is YES.

[2]By the way the reduction works, formulas generated for factorization problem of two distinct n-bit integers do have the same size and very similar structure. Yet these formulas may differ when it comes to the satisfiability.

2. Preeliminaries

We assume that the reader is familiar with basic notions regarding mathematics, logic and complexity theory. In this chapter we recall notions needed to understand the SAT problem and we establish our notation.

2.1. The Boolean Satisfiability and CNF

In this section we give a formal definition of concepts related to boolean satisfiability and conjunctive normal form. We define formally what we mean by a boolean formula

Definition 1. *Boolean formulas* F are defined recursively as follows. A formula is either:

1. a boolean variable (plain boolean variable is itself the simplest possible boolean formula)
2. another formula F_1 in parentheses
3. negation of another formula F_1
4. conjunction of two other formulas F_1 and F_2
5. disjunction of two other formulas F_1 and F_2
6. implication (F_1 implies F_2), where F_1 and F_2 are two formulas
7. equivalence of F_1 and F_2, where F_1 and F_2 are two formulas

The definition above states a formal grammar used to generate the language of valid boolean formulas. It is important to mention the precedence of operators (from highest to lowest):

1. () - parentheses have the highest priority
2. $\overline{x}$ - negation (of x)
3. $\wedge$ - conjunction
4. $\vee$ - disjunction
5. $\Rightarrow$ - implication
6. $\Leftrightarrow$ - equivalence

We introduce a concept of a truth assignment, which is simply an assignment of truth value to every variable in a boolean formula.

Definition 2. *Truth assignment* is a function ψ that assigns a truth value to every variable in a formula F (set of variables is denoted as $\text{vars}(F)$): $\psi : \text{vars}(F) \to \{\text{TRUE}, \text{FALSE}\}$

Having a truth assignment we replace all variables in a formula with their respective truth values. Then by using well known rules of logic, we simply an expression consisting of truth values and logical connectives (operators) to obtain a single truth value. This is known in logic as a valuation.

Definition 3. *(Valuation)* Let ψ be a truth assignment to variables of F. We define $\Psi : \{F | F \text{ is a boolean formula}\} \times \{\psi | \psi \text{ is a truth assignment to } F\} \to \{\text{TRUE}, \text{FALSE}\}$ (valuation of F under assignment ψ) in the following recursive way:

1. $\Psi(b, \psi) = \psi(b)$ (a valuation of a formula consisting of a single boolean variable is simply the truth value of this variable)
2. $\Psi((F_1), \psi) = \Psi(F_1, \psi)$ (parentheses do not affect valuation)
3. $\Psi(\overline{F_1}, \psi) = \begin{cases} \text{TRUE} & \text{if } \Psi(F_1, \psi) = \text{FALSE} \\ \text{FALSE} & \text{otherwise} \end{cases}$
4. $\Psi(F_1 \wedge F_2, \psi) = \begin{cases} \text{TRUE} & \text{if } \Psi(F_1, \psi) = \text{TRUE and } \Psi(F_2, \psi) = \text{TRUE} \\ \text{FALSE} & \text{otherwise} \end{cases}$
5. $\Psi(F_1 \vee F_2, \psi) = \begin{cases} \text{TRUE} & \text{if } \Psi(F_1, \psi) = \text{TRUE or } \Psi(F_2, \psi) = \text{TRUE} \\ \text{FALSE} & \text{otherwise} \end{cases}$
6. $\Psi(F_1 \Rightarrow F_2, \psi) = \Psi(\overline{F_1} \vee F_2)$
7. $\Psi(F_1 \Leftrightarrow F_2, \psi) = \Psi((F_1 \Rightarrow F_2) \wedge (F_2 \Rightarrow F_1), \psi)$

If there is an assignment (at least one) that valuates to truth, we call a formula satisfiable. More formal definition below.

Definition 4. *Satisfiability* Let F be a boolean formula and Ψ be a valuation function. We call F satisfiable iff there exists a satisfying assignment ψ such that: $\Psi(F, \psi) = \text{TRUE}$. If a formula is not satisfiable then we call it unsatisfiable.

Example of the satisfiable boolean formula.

Example 1. Consider the following boolean formula: $F \equiv x_1 \wedge (\overline{x_1} \vee x_2)$. Formula F is clearly satisfiable because $\Psi(x_1 \wedge (\overline{x_1} \vee x_2), \{\psi(x_1) = \text{TRUE}, \psi(x_2) = \text{TRUE}\}) = \text{TRUE}$. In other words, assignment $x_1 = \text{TRUE}$ and $x_2 = \text{TRUE}$ is a satisfying assignment.

Example of the unsatisfiable boolean formula.

Example 2. Consider the following boolean formula: $F \equiv (x_1 \vee x_2) \wedge (x_1 \vee \overline{x_2}) \wedge (\overline{x_1} \vee x_2) \wedge (\overline{x_1} \vee \overline{x_2})$. It is easy to check that this formula is unsatisfiable because under all possible truth assignments it evaluates to $FALSE$

A boolean variable or it's negation is also called the *literal.* Both x and $\overline{x}$ are literals. A disjunction of literals is called the *clause.* For instance: $(x_1 \vee \overline{x_2})$ and $(x_1 \vee x_2 \vee x_3)$ are both clauses. We consider a special way in which we write boolean formulas as a conjunction of clauses

Definition 5. We say that a formula F is written in *Conjunctive Normal Form (CNF)* if F is a conjunction of clauses i.e. $F \equiv \bigwedge_{i=1}^{m} c_i$

We provide an example of the formula written in a *CNF.*

Example 3. $F \equiv (x_1 \vee x_2) \wedge (x_1 \vee \overline{x_2}) \wedge (\overline{x_1} \vee x_2) \wedge (\overline{x_1} \vee \overline{x_2})$ is a *CNF*. The set of clauses is $\{(x_1 \vee x_2), (x_1 \vee \overline{x_2}), (\overline{x_1} \vee x_2), (\overline{x_1} \vee \overline{x_2})\}$. The set of literals is $\{x_1, \overline{x_1}, x_2, \overline{x_2}\}$

We define the *SAT-CNF* problem as a special case of the *SAT* problem for which the input formulas are in a *CNF*.

Remark 1. Every boolean formula can be transformed into *CNF* efficiently. One way of doing it is to employ the so-called Tseytin transformation [Tse68]. The result of this transformation is the formula *equisatisfiable* to the original formula (satisfiable iff the original formula is satisfiable). A Tseytin transformation is summarized in the following steps:

1. Generate the parsing (derivation) tree for the boolean formula F based on boolean formulas grammar (Definition 1).
2. For every internal node in the generated tree, introduce a boolean variable b and add clause(s) assuring that it is logically equivalent to subformula derived from it's children. For instance consider the formula $F_1 \to F_2 \vee F_3$ (meaning: F_1 is the parent, F_2, $\vee$, F_3 are children of F_1 in the derivation tree). Recursively applying Tseytin transformation on F_1 introduces variables f_2 for F_2 and f_3 for F_3. When introducing variable f_3 to represent F_3, we have to add the following logical equivalence constraint: $f_1 \Leftrightarrow (f_2 \vee f_3)$ which can be written in *CNF* as: $(\overline{f_1} \vee f_2 \vee f_3) \wedge (\overline{f_2} \vee f_1) \wedge (\overline{f_3} \vee f_1)$
3. For the root node, we need to assure that variable representing it is set to TRUE. It is enough to add the single-element clause (r) to express this constraint.

2.2. The Integer Factorization Problem

In this section we provide a brief introduction regarding the *Integer Factorization* problem. Given the $n \in \mathbb{Z}$ we ask if there are $p \in \mathbb{Z}$ and $q \in \mathbb{Z}$ such that $n = pq$ and $1 < p, q < n$. If this is the case then we call p and q the nontrivial factors of n and n itself is called composite. If n has no nontrivial factors, we call it a prime (prime number). For all $n \geq 1$ there is always a prime p such that $n < p \leq 2n$. This fundamental fact is known as the Bertrand's postulate. One of the proofs of this fact was produced by Paul Erdős and is presented by Galvin [Gal15]. Because of Bertrand's postulate we can be sure that there is at least one prime among n-bit integers. It is obvious that for $n \geq 3$ there is also at least one composite among n-bit integers. This fact is of special importance to us because we consider boolean formulas generated for *Integer Factorization* of n-bit integers in the following chapters.

2.3. The OWA-Winner Problem

In this section we provide a brief introduction regarding the *OWA-Winner* problem.[1] The *OWA-Winner* problem was originally introduced by Skowron, Faliszewski and Lang [SFL14] and is related to voting and elections. The formal setting is presented below. Given a set of n agents $N = \{1, 2, ..., n\}$, and a set of m items $A = \{a_1, a_2, ..., a_m\}$, we want to select a size-K set W of items which in some sense are the most satisfying for the agents. In order to measure the level of satisfaction for each agent $i \in N$ and for each item $a_j \in A$, we introduce an intrinsic utility $u_{i,a_j} \geq 0$ that agent i derives from a_j. Total satisfaction (intrinsic utility) of agent i derived from set W is measured as an ordered weighted average of this agent's utilities for these items. A *weighted ordered average (OWA)* operator over K numbers can be defined through a vector $\alpha^{(K)} = \langle \alpha_1, \alpha_2, ..., \alpha_K \rangle$ of K nonnegative numbers in a following way. Let $\vec{x} = \langle x_1, x_2, ..., x_K \rangle$ be a vector consisting of K numbers and let $\vec{x}^{\downarrow} = \langle x_1^{\downarrow}, x_2^{\downarrow}, ..., x_K^{\downarrow} \rangle$ be the nonincreasing rearrangement of $\vec{x}$, that is, $x_i^{\downarrow} = x_{\sigma(i)}$, where σ is a permutation of $\{1, 2, ..., K\}$ such that $x_{\sigma(1)} \geq x_{\sigma(2)} \geq ... \geq x_{\sigma(K)}$ Then we define meaning of OWA operator in the following way:

$$\text{OWA}\alpha^{(K)}(\vec{x}) = \sum_{i=1}^{K} \alpha_i x_i^{\downarrow}$$

For simplicity we will write $\alpha^{(K)}(x_1, x_2, ..., x_K)$ instead of $OWA\alpha^{(K)}(x_1, x_2, ..., x_K)$. Having defined what ordered weighted operator is, we focus on formalizing the problem of computing "the most satisfying set of K items" as follows.

Definition 6. [SFL14] In the OWA-Winner problem we are given a set $N = [n]$ of agents, a set $A = \{a_1, ..., a_m\}$ of items, a collection of agent's utilities $(u_i, a_j)_{i \in [n], a_j \in A}$,

[1] OWA stands for Ordered Weighted Average

a positive integer $K(K \leq m)$, and a K-number OWA $\alpha^{(K)}$. The task is to compute a subset $W = \{w_1, ..., w_K\}$ of A such that $u_{ut}^{\alpha^{(K)}}(W) = \sum_{i=1}^{n} \alpha^{(K)}(u_{i,w_1}, ..., u_{i,w_K})$ is maximal.

The definition above can be translated into an integer linear program (ILP). One such translation is presented by Skowron et al. [SFL14] . In this thesis we reconsider this translation and provide corrections to minor errors present in the original.

Theorem 1. *[SFL14] OWA-Winner problem can be stated as a following integer linear program:*

$$
\begin{aligned}
&\text{maximize} \sum_{i=1}^{n}\sum_{j=1}^{m}\sum_{k=1}^{K} \alpha_k u_{i,a_j} x_{i,j,k} \\
&\text{subject to :} \\
&(a) : \sum_{i=1}^{m} y_i = K \\
&(b) : x_{i,j,k} \leq y_j && , i \in [n]; j \in [m]; k \in [K] \\
&(c) : \sum_{j=1}^{m} x_{i,j,k} = 1 && , i \in [n]; k \in [K] \\
&(d) : \sum_{k=1}^{K} x_{i,j,k} \leq 1 && , i \in [n]; j \in [m] \\
&(e) : \sum_{j=1}^{m} u_{i,a_j} x_{i,j,k} \geq \sum_{j=1}^{m} u_{i,a_j} x_{i,j,(k+1)} && , i \in [n]; k \in [K-1] \\
&(f) : x_{i,j,k} \in \{0,1\} && , i \in [n]; j \in [m]; k \in [K] \\
&(g) : y_j \in \{0,1\} && , j \in [m]
\end{aligned}
$$

$[n]$ is the set of agents, $A = \{a_1, ..., a_m\}$ is the set of items, $\alpha = \{\alpha_1, ..., \alpha_k\}$ is the *OWA* vector, $u_{i,a_j}$is the utility that the agent i derives from the item a_j.

Proof. The intended meaning of the variables in this *ILP* formulation is as follows:

$$
x_{i,j,k} = \begin{cases} 1 & \text{for agent i item } \mathrm{a_j} \text{ is the k}-\text{th most preferred from items in a solution} \\ 0 & \text{otherwise} \end{cases}
$$

$$
y_j = \begin{cases} 1 & \text{item j is taken in a solution} \\ 0 & \text{otherwise} \end{cases}
$$

By maximizing: $\sum_{i=1}^{n}\sum_{j=1}^{m}\sum_{k=1}^{K} \alpha_k u_{i,a_j} x_{i,j,k}$ we maximize the total sum of weighted utilities that agents derives from the items. This is consistent with the problem's statement. Below we clarify why conditions (a)-(g) are necessary in this *ILP* formulation:

(a) - This condition states that exactly K items are chosen in a solution.

(b) - If item a_j is not chosen in a solution, then there should be no agent i for whom this item appears on $k-$th position from items appearing in a solution. This constraint enforces that x and y are mutually consistent with each other.

(c) - For agent i, there is exactly one item on the $k-$th most preferred place from items appearing in a solution.

(d) - For agent i and item a_j, we require that agent i views item a_j on at most one position from the solution. Note that agent i may not view item a_j among his/her list of K most preferred items (but still item a_j might have been taken into solution).

(e) - For agent i, utility derived from item appearing on the $k-$th position in a solution is not smaller than the utility derived from the item appearing on the $(k+1)-$st position in the solution.

(f) - $x_{i,j,k}$ is a binary variable for $i \in [n]; j \in [m]; k \in [K]$

(g) - y_j is a binary variable for $j \in [m]$ □

The theorem proved above will be very useful when designing a *SAT-CNF* encoding of *OWA-Winner* problem.

2.4. Basic Notions and Definitions Used to Express Boolean Constraints

Below we introduce vocabulary used in the following chapters to describe various boolean constraints. Most of the terms should be familiar and self-explanatory. We start by defining the notion of a *boolean variable.*

Definition 7. *Boolean variable* x is a variable taking values from $\{0, 1\}$ (being either FALSE or TRUE)

Performing operations on individual boolean variables is quite cumbersome and sometimes we want to group a bunch of boolean variables into one collection. Formally we will call such collections *sequences.*

Definition 8. *Sequence (of boolean variables)* $\langle x_1, x_2, x_3, .., x_n\rangle$ is an ordered collection of boolean variables of fixed size. The length of a sequence is a number of boolean variables associated with a sequence. $\text{length}(\langle x_1, x_2, .., x_n\rangle) = n$

Sequences of length n can be used to represent n-bit integers. Each variable in a sequence is representing exactly one bit.

Remark 2. When using sequence $X = \langle x_1, x_2, .., x_n\rangle$ to represent integers, we use the convention that x_1 corresponds to the least significant bit and x_n corresponds to the most significant bit.

3. Reducing Selected Computational Problems to SAT-CNF

In this chapter we present the detailed description of how to reduce both *Integer Factoriazation* and *OWA-Winner* problems to *SAT-CNF.*

3.1. Reducing Integer Factorization to SAT-CNF

Since *Integer Factorization* problem belongs to the class *NP*,[1] there is a way to reduce it to *SAT-CNF* in polynomial time. Arguably, the most direct way of doing so is to encode multiplication circuit as a *SAT-CNF* formula. One of such encodings is available in the work of Srebrny [Sre04]. In the following subsections we present descriptions of various constraints used in this encoding. The main goal of each subsection is to establish either a *CNF* encoding for a given constraint or an algorithm producing such an encoding.

Encoding Equality of Sequences X and Y ($\mathbf{X = Y}$)

To represent equality between sequences X and Y it suffices to encode 'variable-wise' equality. Given two sequences X and Y

$$X = \langle x_1, x_2, .., x_n \rangle$$
$$Y = \langle y_1, y_2, .., y_n \rangle$$

We define the equality of sequences in the following way

$$X = Y \iff (x_1, x_2, .., x_n) = (y_1, y_2, .., y_n)$$

This equality constraint can be written as a conjunction of equivalences

$$\bigwedge_{i=1}^{n} (x_i \Leftrightarrow y_i)$$

[1] It is easy to verify that given n and numbers p and q if $n = pq$

Finally, we replace equivalences with logically equivalent conjunctions of disjunctions to obtain

$$\bigwedge_{i=1}^{n}((\overline{x_i} \vee y_i) \wedge (x_i \vee \overline{y_i}))$$

(in a conjunctive normal form)

Encoding Inequality Between a Sequence X and a Constant I ($\mathbf{X \neq I}$)

This type of constraint is especially useful when we want to enforce that some sequence X is **not** equal given integer I. For example, we may wish that our factor X (represented by the sequence) is not equal 1. For this to hold we need to encode $X \neq 1$ constraint as a SAT-CNF formula (set of clauses). Given a sequence X and an integer (constant) I, which is represented as a sequence of bits

$$X = \langle x_1, x_2, .., x_n \rangle$$
$$I = \langle i_1, i_2, .., i_n \rangle$$

We define the inequality $X \neq I$ in the following way

$$X \neq I \iff (x_1, x_2, .., x_n) \neq (i_1, i_2, .., i_n)$$

Let us introduce a sequence $Y = \langle y_1, y_2, .., y_n \rangle$, where $y_i = x_i$ if the i-th bit of I is 0 (If the i-th bit of I is 1 then $y_i = \overline{x_i}$). The inequality $X \neq I$ holds when the following clause is satisfiable

$$\bigvee_{i=1}^{n} y_i$$

Example 4. Let $I = 13$ and $X = \langle x_1, x_2, x_3, x_4 \rangle$ Constraint $X \neq I$ can be encoded as $(\overline{x_1} \vee x_2 \vee \overline{x_3} \vee \overline{x_4})$

Encoding Shift Equality Constraint ($\mathbf{Y = 2^i X}$)

We are given two sequences X and Y

$$X = \langle x_1, x_2, .., x_n \rangle$$
$$Y = \langle y_1, y_2, .., y_n \rangle$$

The shift equality constraint is basically stating that after shifting X by i positions to the left we obtain Y. The shift by i positions to the left can be defined in the following way

$$2^i X = \langle \underbrace{0, .., 0}_{i}, x_1, .., x_{n-i} \rangle$$

By definition of the equality between sequences we have

$$Y = 2^i X \iff (y_1, y_2, .., y_n) = (\underbrace{0, .., 0}_{i}, x_1, .., x_{n-i})$$

This constraint is encoded in the following way

$$(\bigwedge_{j=1}^{i} \overline{y_j}) \wedge \bigwedge_{j=i+1}^{n} (y_j \Leftrightarrow x_{j-i})$$

Finally, we replace equivalences with logically equivalent conjunctions of disjunctions to obtain

$$(\bigwedge_{j=1}^{i} \overline{y_j}) \wedge \bigwedge_{j=i+1}^{n} ((y_j \vee \overline{x_{j-i}}) \wedge (\overline{y_j} \vee x_{j-i}))$$

(in a conjunctive normal form)

Encoding Left Variable-Wise Multiplication (bX = Y)

We are given two sequences X, Y and a boolean variable b

$$X = \langle x_1, x_2, .., x_n \rangle$$
$$Y = \langle y_1, y_2, .., y_n \rangle$$

We want to encode the following equality

$$bX = Y \iff (b \wedge x_1, b \wedge x_2, .., b \wedge x_n) = (y_1, y_2, .., y_n)$$

The condition $bX = Y$ is encoded in the following way

$$\bigwedge_{i=1}^{n} ((b \wedge x_i) \Leftrightarrow y_i)$$

Finally, we rewrite the formula above as

$$\bigwedge_{i=1}^{n} ((b \vee \overline{y_i}) \wedge (x_i \vee \overline{y_i}) \wedge (y_i \vee \overline{b} \vee \overline{x_i}))$$

Encoding Addition ($\mathbf{X + Y = Z}$)

We are given three sequences X, Y and Z

$$X = \langle x_1, x_2, .., x_n \rangle$$
$$Y = \langle y_1, y_2, .., y_n \rangle$$
$$Z = \langle z_1, z_2, .., z_n \rangle$$

In order to encode the addition of two sequences ($X + Y = Z$), we need to introduce additional sequence C representing carry bits

$$C = \langle c_0, c_1, .., c_n \rangle$$

Please note that C has length of $n + 1$. Addition can be depicted as follows

$$\begin{array}{cccccc}
c_n & c_{n-1} & \dots & c_1 & c_0 \\
 & x_{n-1} & \dots & x_1 & x_0 \\
+ & y_{n-1} & \dots & y_1 & y_0 \\
\hline
 & z_{n-1} & \dots & z_1 & z_0
\end{array}$$

For the whole addition to be valid, we require that c_0 and c_n are both 0 (FALSE). c_{i+1} is 1 (TRUE) if at least two of $\{x_i, y_i, c_i\}$ are 1. Otherwise c_{i+1} is 0. Value of z_i is 1 if either exactly one of $\{x_i, y_i, c_i\}$ is 1 or exactly three of $\{x_i, y_i, c_i\}$ are 1. Otherwise z_i is 0. To encode such addition, we need to translate all these requirements to *CNF*. One of such translations is presented below.

$X + Y = Z$ (with carry C):

$$\begin{aligned}
&(\overline{c_0}) \land (\overline{c_n}) \\
&\land \bigwedge_{i=1}^{n} ((\overline{c_i} \lor x_{i-1} \lor c_{i-1}) \land (\overline{c_i} \lor x_{i-1} \lor y_{i-1}) \land (\overline{c_i} \lor y_{i-1} \lor c_{i-1}) \\
&\land (c_i \lor \overline{x_{i-1}} \lor \overline{c_{i-1}}) \land (c_i \lor \overline{x_{i-1}} \lor \overline{y_{i-1}}) \land (c_i \lor \overline{y_{i-1}} \lor \overline{c_{i-1}})) \\
&\land \bigwedge_{i=0}^{n-1} ((z_i \lor y_i \lor x_i \lor \overline{c_i}) \land (z_i \lor y_i \lor \overline{x_i} \lor c_i) \land (z_i \lor \overline{y_i} \lor x_i \lor c_i) \land (z_i \lor \overline{y_i} \lor \overline{x_i} \lor \overline{c_i}) \\
&\land (\overline{z_i} \lor y_i \lor x_i \lor c_i) \land (\overline{z_i} \lor y_i \lor \overline{x_i} \lor \overline{c_i}) \land (\overline{z_i} \lor \overline{y_i} \lor x_i \lor \overline{c_i}) \land (\overline{z_i} \lor \overline{y_i} \lor \overline{x_i} \lor c_i))
\end{aligned}$$

Encoding Multiplication ($\mathbf{PQ = N}$)

Consider two k-bit integers p and q, which can be expressed in the binary form

$$p = (p_k p_{k-1} \dots p_2 p_1)_2$$
$$q = (q_k q_{k-1} \dots q_2 q_1)_2$$

Formula for computing the product of two numbers, p and q can be expressed as

$$pq = q_1p + q_22p + q_32^2p + ... + q_k2^{k-1}p$$

We extend the notion of multiplication to sequences. Consider two sequences P and Q

$$P = \langle p_1, p_2, .., p_n\rangle$$
$$Q = \langle q_1, q_2, .., q_n\rangle$$

The multiplication of sequences is defined as

$$PQ = q_1P + q_22P + q_32^2P + ... + q_k2^{k-1}P$$

Careful reader can note that the formula above is basically a **sum of shift multiplications** for which we have already shown appropriate encodings. We need a lot of additional variables (and sequences) to construct *CNF* encoding of $PQ = N$. Let ln mean $length(N)$ and let lq mean $length(Q)$. Below is a summary of additional sequences used to construct *CNF* encoding of $PQ = N$:

- S is an array of lq sequences of length ln (i.e. $S = [S_0, S_1, .., S_{lq-1}]$ and $length(S_i) = ln$)

- C is an array of $lq - 1$ sequences of length $ln + 1$

- M is an array of lq sequences of length ln

- R is an array of lq sequences of length ln

Instead of writing the encoding down using explicit *CNF* formula, we take the approach of providing an algorithm (in form of a pseudocode) representing the steps necessary to generate such an encoding. We start with an empty formula ϵ (no clauses) and then we proceed by adding clauses derived from the *CNF* encodings of various constraints. Algorithm 3.1, in each step we extend the output formula by the clauses derived from the *CNF* encoding of a particular constraint. $CNF(c)$ simply denotes the *CNF* encoding of a constraint c e.g. $CNF(A = B + C)$ means the CNF encoding of an addition $A = B + C$. Last two for loops are there to fix some variables in P and Q in order to explicitely decrease the search space.

Algorithm 3.1 Generating *CNF* for $PQ = N$

$f \leftarrow \epsilon$
$f \leftarrow CNF(S_0 = P) \wedge f$
for $i = 1$ **to** $lq - 1$ **do**
  $f \leftarrow CNF(S_i = 2S_{i-1}) \wedge f$
end for
for $i = 0$ **to** $lq - 1$ **do**
  $f \leftarrow CNF(M_i = Q_i S_i) \wedge f$
end for
$f \leftarrow CNF(R_0 = M_0) \wedge f$
for $i = 1$ **to** $lq - 1$ **do**
  $f \leftarrow CNF(R_{i-1} + M_i = R_i) \wedge f$ // carry=C_{i-1}
end for
$f \leftarrow CNF(R_{lq-1} = N) \wedge f$
for each pair $(i, j) \in [0, 1, .., ln - 1] \times [0, 1, .., lq - 1]$ **do**
  if $i + j \geq ln$ **then**
    $f \leftarrow (\bar{P_i} \vee \bar{Q_j}) \wedge f$ // to ensure that multiplication result does not have more bits than N
  end if
end for
for $i = 0$ **to** $lq - 1$ **do**
  if $i > \frac{lq-1}{2}$ **then**
    $f \leftarrow (\bar{Q_i}) \wedge f$ // Limiting number of significant bits in Q
  end if
end for
return f

Encoding Nontriviality $(\mathbf{P \neq 1, Q \neq 1})$

The final step needed to reduce *Integer Factorization* to *SAT-CNF* is to enforce that both P and Q represent nontrivial factors, i.e., that $1 < P < N, 1 < Q < N$

There are multiple ways to do it, but the most straightforward is to demand:

$$Q \neq 1$$

Up to this point, we have shown all steps necessary to convert arbitrary *Integer Factorization* problem instance to boolean formula in *CN*F. If the formula created in such fashion turns out to be unsatisfiable then we can be sure that there are no nontrivial factors for the original *Integer Factorization* problem instance. On the other hand, if there is a satisfying assignment, then we can recover factors by looking at the part of the satisfying assignment that corresponds to P and Q

3.2. Reducing OWA-Winner to SAT-CNF

In this section we develop a machinery needed to reduce the *OWA-Winner*[2] problem to *SAT-CNF*. To do this, we consider the *ILP* formulation of *OWA-Winner* presented in Chapter 1

Encoding Inequality between Sequences ($\mathbf{X \leq Y}$)

We are given two sequences X and Y

$$X = \langle x_1, x_2, .., x_n \rangle$$
$$Y = \langle y_1, y_2, .., y_n \rangle$$

We want to define a way of comparing X and Y. The most natural way of doing so is to adopt the definition we use when compare binary expansions of integers. If we pretend that X and Y are binary expansions of integers, we can write the following definitions for $X \leq Y$

$$X \leq Y \iff (x_n < y_n) \vee (x_n = y_n \wedge (x_{n-1} < y_{n-1} \vee ...(x_1 = y_1 \vee (x_1 < y_1))))$$

If the most significant digit of X (x_n) is smaller than the most significant digit of Y (y_n), then we know for sure that $X \leq Y$. If $x_n = y_n$ we keep comparing x_{n-1} against y_{n-1} and so on.

Below (Algorithm 3.2) we provide an algorithm which constructs a boolean formula encoding for $X \leq Y$. We simply initialize the formula f and in the for loop we keep extending f.

Algorithm 3.2 Encoding $X \leq Y$

1: $f \leftarrow (\bar{x_1} \wedge y_1) \vee ((\bar{x_1} \vee y_1) \wedge (x_1 \vee \bar{y_1}))$
2: **for** $i = 2$ **to** n **do**
3: $\quad f \leftarrow (\bar{x_i} \wedge y_i) \vee (((\bar{x_i} \vee y_i) \wedge (x_i \vee \bar{y_i})) \wedge f)$
4: **end for**
5: **return** f

The formula generated using algorithm Algorithm 3.2 is not in *CNF*. To convert it to *CNF* efficiently, we take advantage of Tseytin transformation [Tse68] (see Section 2.1)

[2]In fact *OWA-Winner* is an optimization problem, so we will consider it's decision version.

Encoding Inequality between a Sequence and a Constant ($\mathbf{X \leq I}$)

Given a sequence X and an integer (constant) I, which is represented as a sequence of bits

$$X = \langle x_1, x_2, .., x_n \rangle$$
$$I = \langle i_1, i_2, .., i_n \rangle$$

We want to encode $X \leq I$, which is a special case of $X \leq Y$. Because of that we can obtain a more efficient encoding.

Algorithm 3.3 Encoding $X \leq I$

if $i_1 = 0$ **then**
    $f \leftarrow \bar{x_1}$
else if $i_1 = 1$ **then**
    $f \leftarrow x_1 \vee \bar{x_1}$
end if
for $j = 2$ **to** n **do**
    if $i_j = 0$ **then**
        $f \leftarrow \bar{x_j} \wedge f$
    else if $i_j = 1$ **then**
        $f \leftarrow \bar{x_j} \vee (x_j \wedge f)$
    end if
end for
return f

Formula expressing $X \leq I$ can be generated using Algorithm 3.3. We can employ Tseytin transformation to convert it to *CNF*.

Encoding Boolean Cardinality Constraints

By now, we have all the encodings necessary, to express the instances of *ILP* as *SAT-CNF* instances. In this section we consider various boolean cardinality constraints and their encodings, which allow us to express some specific integer linear programs as boolean formulas more efficiently. We show an efficient implementation of those constraints based on the work of Sinz [Sin]. The boolean cardinality constraints are giving bounds on how many boolean variables (from a given set of boolean variables) are TRUE. Below we define three major types of boolean cardinality constraints (at most k of, at least k of, exactly k of)

Definition 9. Let $X = \{x_1, x_2, .., x_n\}$ be a set of boolean variables. We define the "at most k of" constraint $_{\leq k}(X)$ by demanding that at most k variables from X are set to TRUE

Example 5. Let $X = \{x_1, x_2, x_3\}$. "At most 1 of X" constraint $_{\leq 1}(\{x_1, x_2, x_3\})$ can be represented as the following boolean formula: $(\overline{x_1} \wedge \overline{x_2} \wedge \overline{x_3}) \vee (x_1 \wedge \overline{x_2} \wedge \overline{x_3}) \vee (\overline{x_1} \wedge x_2 \wedge \overline{x_3}) \vee (\overline{x_1} \wedge \overline{x_2} \wedge x_3)$. It enforces that there are no 2 variables set to $TRUE$ at the same time.

Definition 10. Let $X = \{x_1, x_2, .., x_n\}$ be a set of boolean variables. We define "at least k of" constraint $_{\geq k}(X)$ by demanding that at least k variables from X are set to TRUE

Definition 11. Let $X = \{x_1, x_2, .., x_n\}$ be a set of boolean variables. We define "exactly k of" constraint $_{=k}(X)$ by demanding that exactly k variables from X are set to TRUE

Remark 3. Let k be an integer and let X be a set of propositional (boolean) variables. The "exactly k of X" can be encoded as a conjunction of the "at least k of X" and the "at most k of X".

In the work of Sinz [Sin] the efficient encodings of "at most k of" and "at least k of" were given. These encodings work by introducing a set of additional variables s. Below we state the theorem that allows us to encode "at most k of" efficiently.

Theorem 2. *[Sin] Encoding $LT_{SEQ}^{n,k}$ expressing $_{\leq k}(\{x_1, x_2, .., x_n\})$ $n > 1, k > 0$ can be stated as follows:*

$$\begin{array}{ll}
(\overline{x_1} \vee s_{1,1}) & \\
(\overline{s_{1,j}}) & 1 < j \leq k \\
(\overline{x_i} \vee s_{i,1}) & 1 < i < n \\
(\overline{s_{i-1,1}} \vee s_{i,1}) & 1 < i < n \\
(\overline{x_i} \vee \overline{s_{i-1,j-1}} \vee s_{i,j}) & 1 < i < n, 1 < j \leq k \\
(\overline{s_{i-1,j}} \vee s_{i,j}) & 1 < i < n, 1 < j \leq k \\
(\overline{x_i} \vee \overline{s_{i-1,k}}) & 1 < i < n \\
(\overline{x_n} \vee \overline{s_{n-1,k}}) &
\end{array}$$

In the theorem 2 the variable $s_{i,j}$ asserts that among i variables $\{x_1, x_2, .., x_i\}$ at most j variables are set to TRUE. Below we state the corollary that allows us to encode "at least k of" efficiently.

Corollary 1. *Let $X = \{x_1, x_2, .., x_n\}$. Encoding $GT_{SEQ}^{n,k}$ expressing $_{\geq k}(X)$ $n > 1, k > 0$ can be stated as a $LT_{SEQ}^{n,n-k}$ encoding with all variables from X negated.*

Proof. Corollary 1 is simply obtained by using the fact that the condition "at least k of" variables are TRUE is the same as "at most $n - k$ of" variables are FALSE. The variable $s_{i,j}$ asserts that among i variables $\{x_1, x_2, .., x_i\}$ at most j variables are set to FALSE. We need to negate all the occurrences of x_j variables, because we are only counting variables set to FALSE in this case. □

Encoding Decision Version of OWA-Winner Problem

Let us state the decision version of *OWA-Winner* problem based on the *ILP* formulation from Chapter 1. Decision version of *OWA-Winner* problem reduces to checking feasibility of following integer linear program:

$$
\begin{aligned}
&(a): \sum_{i=1}^{n}\sum_{j=1}^{m}\sum_{k=1}^{K} \alpha_k u_{i,a_j} x_{i,j,k} \geq L && L \in \mathbb{N}\\
&(b): \sum_{i=1}^{m} y_i = K &&\\
&(c): x_{i,j,k} \leq y_j && , i \in [n]; j \in [m]; k \in [K]\\
&(d): \sum_{j=1}^{m} x_{i,j,k} = 1 && , i \in [n]; k \in [K]\\
&(e): \sum_{k=1}^{K} x_{i,j,k} \leq 1 && , i \in [n]; j \in [m]\\
&(f): \sum_{j=1}^{m} u_{i,a_j} x_{i,j,k} \geq \sum_{j=1}^{m} u_{i,a_j} x_{i,j,(k+1)} && , i \in [n]; k \in [K-1]\\
&(g): x_{i,j,k} \in \{0,1\} && , i \in [n]; j \in [m]; k \in [K]\\
&(h): y_j \in \{0,1\} && , j \in [m]
\end{aligned}
$$

Having stated what we mean by the decison version of *OWA-Winner* problem, we can finally present a way of encoding arbitrary *OWA-Winner* problem instances as a *SAT-CNF* formula. $CNF(c)$ simply denotes the *CNF* encoding of a constraint c e.g. $CNF(A = B + C)$ means the CNF encoding of an addition $A = B + C$.

Theorem 3. *Decision OWA-Winner problem instances can be encoded as SAT-CNF formulas in the following way:*

$$
\begin{aligned}
&(a): CNF(\sum_{i=1}^{n}\sum_{j=1}^{m}\sum_{k=1}^{K} \alpha_k u_{i,a_j} x_{i,j,k} \geq L) && L \in \mathbb{N}\\
&(b): CNF(_{=K}(\{y_j | j \in [m]\})) &&\\
&(c): (\overline{x_{i,j,k}}, y_j) && , i \in [n]; j \in [m]; k \in [K]\\
&(d): CNF(_{=1}(\{x_{i,j,k} | j \in [m]\})) && , i \in [n]; k \in [K]\\
&(e): CNF(_{\leq 1}(\{x_{i,j,k} | k \in [K]\})) && , i \in [n]; j \in [m]\\
&(f): CNF(\sum_{j=1}^{m} u_{i,a_j} x_{i,j,k} \geq \sum_{j=1}^{m} u_{i,a_j} x_{i,j,(k+1)}) && , i \in [n]; k \in [K-1]\\
&(g): x_{i,j,k} \in \{0,1\} && , i \in [n]; j \in [m]; k \in [K]\\
&(h): y_j \in \{0,1\} && , j \in [m]
\end{aligned}
$$

Proof. We need to show that constraints (a) - (h) are expressible using *SAT-CNF* encodings constructed so far. Constraints (g) and (h) are clearly just declaring sets of propositional variables: x and y, and therefore produce no clauses in a *CNF* encoding. Constraint (a) is simply an inequality between sequence constructed from sum of products and integer ($S \geq L$ and $S = \sum_{i=1}^{n} \sum_{j=1}^{m} \sum_{k=1}^{K} \alpha_k u_{i,a_j} x_{i,j,k}$). So, while being quite costly (in terms of number of variables and clauses) it is expressible in the *SAT-CNF* format.

Similarly, for constraint (f) we can write $S_1 \geq S_2$ where S_1 is a sequence ($S_1 = \sum_{j=1}^{m} u_{i,a_j} x_{i,j,k}$) and S_2 is a sequence ($S_2 = \sum_{j=1}^{m} u_{i,a_j} x_{i,j,(k+1)}$). Constraint (c) is a simple clause logically equivalent to: ($x_{i,j,k} \Rightarrow y_j$), which behaves alike $x_{i,j,k} \leq y_j$. Constraints (b), (d) and (e) are all boolean cardinality constraints for which we have already shown efficient encodings. □

Constraints (a) and (f) are the most costly elements in the model. In the next section we will look at somewhat restricted version of decision *OWA-Winner* problem in which these constraints are simplified.

Encoding Decision Version of k-Best-OWA-Approval-Winner Problem

As we saw in the previous subsection, it is possible to convert any *Decision OWA-Winner* problem instance to a *SAT-CNF* formula. It is prohibitevely expensive to encode constraints (a) and (f) from theorem 3 (requiring lots of sequence multiplications). In this subsection we will present more restricted yet still computationally demanding version of Decision *OWA-Winner* problem.

Definition 12. Decision version of *k-Best-OWA-Approval-Winner* problem is obtained from Decision version of *OWA-Winner* problem by:

- Requiring an OWA vector α and a derived utility u to be binary ($\alpha_i \in \{0,1\}, u_{i,a_j} \in \{0,1\}$)
- Removing the following constraint: $(f) : \sum_{j=1}^{m} u_{i,a_j} x_{i,j,k} \geq \sum_{j=1}^{m} u_{i,a_j} x_{i,j,(k+1)}, i \in [n]; k \in [K-1]$, which basically is not needed when α is non-increasing.

SAT-CNF encoding of Decision *k-Best-OWA-Approval-Winner* problem follows:

Theorem 4. *Encoding Decision k-Best-OWA-Approval-Winner problem instances as SAT-CNF formulas*

$$
\begin{aligned}
&(a) : CNF(_{\geq L}(\{x_{i,j,k} | i \in [n], j \in [m], k \in [K], \alpha_k u_{i,a_j} > 0\})) && \\
&(b) : CNF(_{=K}(\{y_j | j \in [m]\})) && \\
&(c) : (\overline{x_{i,j,k}}, y_j) && , i \in [n]; j \in [m]; k \in [K] \\
&(d) : CNF(_{=1}(\{x_{i,j,k} | j \in [m]\})) && , i \in [n]; k \in [K] \\
&(e) : CNF(_{\leq 1}(\{x_{i,j,k} | k \in [K]\})) && , i \in [n]; j \in [m] \\
&(f) : x_{i,j,k} \in \{0,1\} && , i \in [n]; j \in [m]; k \in [K] \\
&(g) : y_j \in \{0,1\} && , j \in [m]
\end{aligned}
$$

Proof. We remove $\sum_{j=1}^{m} u_{i,a_j} x_{i,j,k} \geq \sum_{j=1}^{m} u_{i,a_j} x_{i,j,(k+1)}, i \in [n]; k \in [K-1]$ constraints. We can easily see that $\alpha_k u_{i,a_j}$ has to be either 0 or 1 (α and u are binary). This fact allows us to transform $\sum_{i=1}^{n} \sum_{j=1}^{m} \sum_{k=1}^{K} \alpha_k u_{i,a_j} x_{i,j,k} \geq L$ into $_{\geq L}(\{x_{i,j,k} | i \in [n], j \in [m], k \in [K], \alpha_k u_{i,a_j} > 0\})$ □

4. Experimental Analysis of Structure of Obtained Formulas

In this chapter we present a set of experimental results for some instances of boolean formulas, generated based on our two example problems, *Integer Factorization* and *OWA-Winner*. We compare properties of generated instances to what is known about randomly generated *SAT-CNF* instances (e.g. we consider the *clauses-to-variables ratio*). The purpose of such an experimental analysis is to capture different measures of hardness related to boolean formulas and how those measures vary depending on the selected computational problem.

4.1. Clauses to Variables Ratio

One of the simplest metrics that can be used when we want to distinguish between satisfiable and unsatisfiable boolean formulas is the so-called *clauses-to-variables ratio*. It is simply the number of clauses divided by the number of distinct variables in the given formula. We consider the following example.

Example 6. Consider the formula$(\overline{x_1} \land \overline{x_2} \land \overline{x_3}) \lor (x_1 \land \overline{x_2} \land \overline{x_3}) \lor (\overline{x_1} \land x_2 \land \overline{x_3}) \lor (\overline{x_1} \land \overline{x_2} \land x_3)$. It has 3 variables and 4 clauses, which gives a *clauses-to-variables ratio* equal to $\frac{4}{3}$.

Intuitively, if this ratio is high then the number of clauses is much bigger than the number of variables. It is clear that adding clauses when keeping number of variables fixed can only make a formula more constrained (harder to satisfy). It turns out that for randomly generated CNF formulas there is a magical constant $M = 4.26$ such that formulas with *clauses-to-variables ratio* smaller than M are mostly satisfiable and formulas with *clauses-to-variables ratio* greater than M are mostly unsatisfiable. Randomly generated formulas with *clauses-to-variables ratio* around M are the hardest ones for modern boolean satisfiability solvers to decide satisfiability. This phenomenon was studied thoroughly and the original idea comes from Selman, Mitchell and Levesque [SML96].

Clauses to Variables Ratio for Integer Factorization Formulas

In the previous chapters we defined all steps necessary to reduce *Integer-Factorization* problem to *SAT-CNF*. Given an integer N, we generate a boolean formula which is satisfiable if and only if N is composite. Satisfying assignment gives us information about the computed factors. By carefully looking at the steps of this reduction, we can notice that the size of the generated formula (understood as the total number of literals in the formula) depends only on the number of bits of N. An even more careful analysis leads us to a conclusion that formulas generated for n-bit integers are identical (by construction) up to the polarity[1] of literals (clauses) enforcing (fixing) bits of N. A simple consequence of this fact is that all formulas generated for n-bit integers have the same *clauses-to-variables ratio*. The natural question is how this *clauses-to-variables ratio* varies as n (the number of bits) increases. We answer this question on Figure 4.1

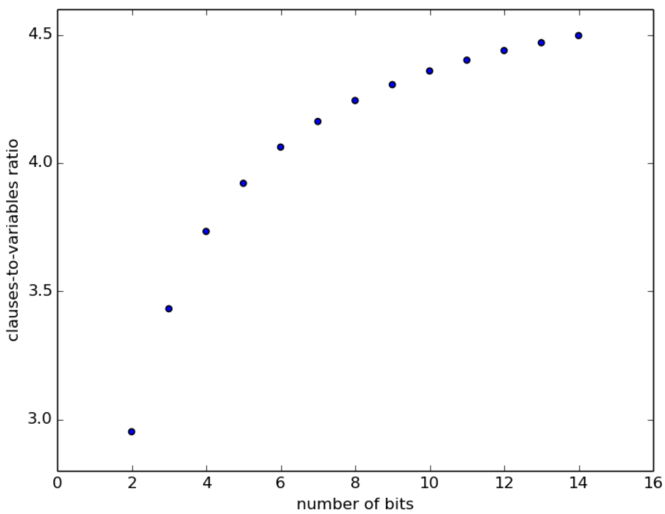

Figure 4.1.: Clauses to Variables Ratio for Integer Factorization Formulas

By looking at Figure 4.1 we can notice that formulas generated for 10-bit numbers have already *clauses-to-variables ratio* clearly above $M = 4.26$. The general observation is that *clauses-to-variables ratio* increases as n (number of bits) increases (the rate of change is getting smaller and smaller as n gets bigger and bigger). To be more precise, the number of variables is $\Theta(n^2)$ and the number of clauses is also $\Theta(n^2)$. The table below shows both the number of variables and the number of clauses with respect to the number of bits.

[1] polarity of a literal refers to the fact if literal appears as a variable or a negated variable. Negated variables are also reffered to as negative literals. Non-negated variables are reffered to as positive literals.

number of bits	number of variables	number of clauses
1	6	11
2	21	62
3	44	151
4	75	280
5	114	447
6	161	654
7	216	899
8	279	1184

We use a polynomial interpolation to obtain the coefficients for the quadratic function representing the number of variables.

$$\text{number of variables} = 4n^2 + 3n - 1$$

For the number of clauses we need to separately consider odd and even values of n.

$$\text{number of clauses} = \begin{cases} \frac{39}{2}n^2 - 8n - \frac{1}{2} & \text{if n is odd} \\ \frac{39}{2}n^2 - 8n & \text{if n is even} \end{cases}$$

We compute the limit of the *clauses-to-variables ratio* as $n \to \infty$.

$$\lim_{n\to\infty} \frac{\text{number of clauses}}{\text{number of variables}} = \frac{39}{8}$$

If we could apply what we know about randomly generated *SAT-CNF* instances, then we would conclude that formulas for bigger n are harder to satisfy (because of bigger *clauses-to-variables ratio*). If those formulas were indeed harder to satisfy then we should have relatively more unsatisfiable instances generated for big n compared to instances generated for small n. On the other hand, it is well known that prime numbers (corresponding to unsatisfiable instances) are getting rarer and rarer as n increases. It seems that the exact structure of generated boolean formula instances is of much greater importance than *clauses-to-variables ratio* here. This leads us to belief that as far as *clauses-to-variables ratio* the set of boolean formula instances generated for *Integer Factorization* behaves in a completely different way than the set of randomly generated instances.

Clauses to Variables Ratio for OWA-Winner Formulas

Since *OWA-Winner* problem is a maximization problem, we consider a set of boolean formulas (one for each utility value in a fixed interval) corresponding to a given

OWA-Winner problem instance. It is clear that formulas corresponding to big utility values should be harder to satisfy than those corresponding to small ones. It is also obvious that there exists a maximum utility value *opt* for which the generated boolean formula is still satisfiable but for all values bigger than *opt* generated formulas are unsatisfiable. Let's consider a particular instance of *OWA-Winner* problem (or it's *OWA* Approval variant). We adopt the following notation to represent information about problem instance(s):

$$\mathrm{kBestOWAApprovalWinner}(\mathrm{N}, \mathrm{M}, \mathrm{K}, \mu, \alpha, \mathrm{p}, \mathrm{v}),$$

where kBestOWAApprovalWinner is the type of the problem, N is the number of agents, M is the number of candidates, K is the size of a committee (the number of chosen candidates), μ are the agent's utilities, α is the number of leading 1's in *OWA* vector, p - expected percentage of 1's in utility vector, v is the lower bound for total utility function values (i.e. to meet criteria total utility function value should be at least v).

Consider a set of boolean formulas corresponding to the following set of problem instances: $S = \{\mathrm{kBestOWAApprovalWinner}(50, 12, 6, \mu, 4, 0.3, v) | \mu\mathrm{-agent's\,utilities}, v \in [200]\}$. In addition to this, we know that maximum utility value that can be obtained for this particular problem instance is 107. We are interested in how the *clauses-to-variables ratio* changes when v increases.

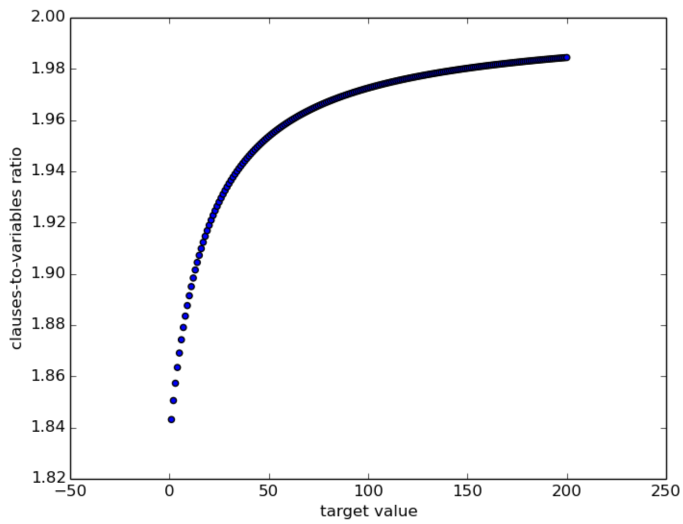

Figure 4.2.: Clauses To Variables Ratio for Particular Instance of k-Best-OWA-Approval-Winner Problem

On Figure 4.2, we can see that for this particular *OWA-Winner* problem instance (represented by S) *clauses-to-variables ratio* increases as v increases, but is below

2.0 for all considered target values (i.e. utility function values). From *clauses-to-variables ratio* perspective all those formulas seem easy (randomly generated instances with such a ratio are very very likely to be satisfiable). This is in fact simply not true in this case because all formulas with target value bigger than 107 are not satisfiable. Formulas are becoming harder and harder to satisfy (bigger target value) as *clauses-to-variables ratio* increases, which matches the behaviour observed for the randomly generated *SAT-CNF* instances.

4.2. Running Time

Probably the simplest (but also somewhat subjective) criterion of telling hard boolean formula instances from easy ones is to measure the time spent by a particular solver of choice on deciding satisfiability of those formulas. Modern boolean satisfiability solvers are sophisticated tools that use many advanced techniques to deal with even millions of variables and clauses. We employ *PicoSAT* solver to evaluate the *running time* it takes to decide satisfiability/unsatisfiability of some specific boolean formula instances.

Running Time for OWA-Winner Formulas

We consider the very same instance of *k-Best-OWA-Approval-Winner* that we were using when discussing *clauses-to-variables ratio.*

We consider $\mathrm{kBestOWAApprovalWinner}(50, 12, 6, \mu, 4, 0.3, \mathrm{v})$.

The goal is to observe how *running time* is changing when v increases.

On Figure 4.3 we can see that *running time* (in seconds) is close to 0 for target values below 70. It suddenly jumps up near 100 (this means that finding a satisfying assignment is getting really hard). At target value 108 problem becomes unsatisfiable. It turns out that proving unsatisfiability for the formulas associated with target values such as 150 or 200 is also really hard. We can see some variance in running times for similar target values. It can be explained by the fact that *SAT* solvers, such as *PicoSAT,* are using heuristics incorporating randomization so as to not get stuck in unpromising areas of the search tree.

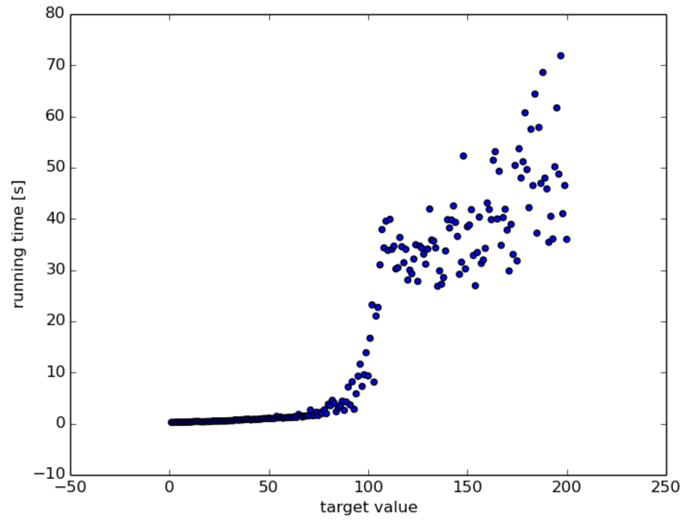

Figure 4.3.: Running Time for Particular Instance of k-Best-OWA-Approval-Winner Problem

Acknowledgments

Firstly I would like to thank my family for all the love and support.

I wish to thank my supervisor Prof. Dr. Piotr Faliszewski for his suggestions and advices.

Last but not least, I am really grateful to all the people who inspired me including colleagues and teachers.

A. Appendix

A.1. Overview

A.2. The next section

Bibliography

[BGV99] R. E. Bryant, S. M. German, and M. N. Velev. Microprocessor verification using efficient decision procedures for a logic of equality with uninterpreted functions. volume Automated Reasoning with Analytic Tableaux and Related Methods, pages 1–13, 1999.

[Coo71] Stephen A. Cook. The complexity of theorem-proving procedures, 1971.

[Gal15] D. Galvin. Erdos's proof of bertrand's postulate. `https://www3.nd.edu/~dgalvin1/pdf/bertrand.pdf`, May 2015.

[HZS] Dapeng Li Hantao Zhang and Haiou Shen. A sat based scheduler for tournament schedules. `http://www.satisfiability.org/SAT04/programme/74.pdf`.

[Kau] Henry Kautz. Deconstructing planning as satisfiability. `https://www.cs.washington.edu/ai/planning/papers/AAAI0609KautzA.pdf`.

[Lar] Tracy Larrabee. Test pattern generation using boolean satisfiability. `https://users.soe.ucsc.edu/~larrabee/ce224/tcad.sat.pdf`.

[NSR02] Gi-Joon Nam, K. A. Sakallah, and R. A. Rutenbar. A new fpga detailed routing approach via search-based boolean satisfiability. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems 21 (6): 674*, 2002.

[SFL14] Piotr Skowron, Piotr Faliszewski, and Jerome Lang. Finding a collective set of items: From proportional multirepresentation to group recommendation, 2014.

[Sin] Carsten Sinz. Towards an optimal cnf encoding of boolean cardinality constraints. `http://www.carstensinz.de/papers/CP-2005.pdf`.

[SML96] B. Selman, D.G. Mitchell, and H.J. Levesque. Generating hard satisfiability problems. *Artificial Intelligence, 81(1-2):17-29*, 1996.

[Sre04] Mateusz Srebrny. Factorization with sat - classical propositional calculus as a programming environment. `http://www.mimuw.edu.pl/~mati/fsat-20040420.pdf`, April 2004.

[Tse68] G. Tseytin. On the complexity of derivation in propositional calculus. *Studies in Constructive Mathematics and Mathematical Logic*, pages 115–125, 1968.

Nomenclature

NP	Nondeterministic Polynomial
OWA	Ordered Weighted Average
SAT	Boolean Satisfiability Problem