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: dr hab. inż. Piotr Faliszewski, prof. AGH

Kraków, 2017

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

The boolean satisfiability problem 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, the boolean satisfiability (*SAT*) arguably still remains 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, we provide a step-by-step description of the reduction from the *OWA-Winner* problem (to be precise, its decision version) to *SAT-CNF*. In order to do this, we 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*, we consider the *clauses-to-variables* ratio as one of the structural properties of obtained boolean formulas instances. We also briefly analyze the *running time* of the *PicoSAT* solver on the obtained instances.

1 Introduction

The 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. A satisfying assignment simply means an assignment of truth values to the variables so that the formula 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.

After the boolean satisfiability problem was shown NP-complete, people found literally hundreds of problems in graph theory, mathematical programming, puzzles and games that are NP-complete, meaning that knowing how to solve one of these problems in polynomial-time would imply polynomial-time algorithms for all of them. One of such problems is the so-called *Hamiltonian-Cycle* problem. In this problem we want to answer the question whether there is a cycle in a given graph that starts in a given vertex and goes through all the vertices, but visits each of them only once. The optimization version of this problem, the so-called *Travelling-Salesman* problem, is NP-hard. It is about finding a hamiltonian cycle with the lowest edge cost possible.

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 [Lar06], routing of FPGAs [NSR02], planning [Kau06], and scheduling [ZLS04] problems. In this thesis we consider the *Integer-Factorization* problem and we analyze the way of reducing it to *SAT*. The purpose of this is to obtain a set of similarly-structured *SAT* instances[2] and inspect their properties. We want to evaluate the performance of modern *SAT* solvers on these particular problem instances.

We also consider the *OWA-Winner* problem (an optimization problem in the election and voting theory) and the way of reducing this problem to *SAT-CNF*. In

[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.

the *OWA-Winner* problem there are agents and items. Each agent has to rate each item, which is done by assigning a number (intrinsic utility) to the item. Then the purpose is to find a set of items that maximizes the general utility over all agents. The actual utility that the agent gains from an item might be different then the intrinsic utility that was assigned to the item by the agent, and depends on the order in which an agent ranks the items.

There are two main goals of this thesis. The first and the most important one is a reduction of the *Integer-Factorization* and *OWA-Winner* problems to *SAT-CNF* in a possibly efficient way. The second goal is related to the experimental exploration of the formulas obtained from the reductions. We investigate two main metrics in our experiments: *clauses-to-variables ratio* and *running time.*

The thesis is organized as follows. First, in Chapter 2, we introduce the boolean satisfiability problem *SAT*, the *Integer-Factorization* problem and the *OWA-Winner* problem. In Section 2.1, we discuss the formal settings needed to understand what boolean formulas are, what is a truth assignment and valuation. We discuss the satisfiability and unsatisfiability, conjunctive normal form and we show that it is possible to convert all the booleans formulas to *SAT-CNF* efficiently using the transformation that preserves the satisfiability property of the original formula (Tseytin transformation). In Section 2.2, we introduce the *Integer-Factorization* problem. We mention the importance of this problem in modern public-key cryptography. We briefly talk about two elementary algorithms of factoring integers: trial division and Fermat's factorization method. We also list some more advanced algorithms for tackling the *Integer-Factorization* problem. In Section 2.3, we introduce the *OWA-Winner* problem giving the intuition behind it as well as real-world applications of this problem. We also give an integer linear programming formulation of the general *OWA-Winner* problem. This is important since in the next chapter, we develop a *SAT-CNF* encoding of the *OWA-Winner* problem (its decision version to be precise) based on this *ILP* formulation.

In Chapter 3, we develop a *SAT-CNF* encoding of the *Integer-Factorization* problem based on the work of Srebrny [Sre04] as well as the *OWA-Winner* problem. In Section 3.1, we introduce basic notions and definitions of objects used to express boolean constraints. We define an important notion of sequences of boolean variables and establish conventions to be used in the rest of Chapter 3. In Section 3.2, we explain the encoding of constraints necessary to reduce the *Integer-Factorization* to *SAT-CNF.* Among others, we encode both the addition and multiplication of sequences of boolean variables. In Section 3.3, we reduce the *OWA-Winner* problem (its decision version) to the *SAT-CNF.* To do this, we first consider the efficient encodings of boolean cardinality constraints. We reduce both the general *OWA-Winner* problem and the approval version of the *OWA-Winner* problem to *SAT-CNF.* The latter reduction is more efficient in terms of the number of variables and clauses in generated formulas and relies to the greater degree on the encodings of boolean cardinality constraints.

In Chapter 4, we investigate both the *clauses-to-variables ratio* and *running time* for selected instances of the *OWA-Winner* problem and *Integer-Factorization* prob-

lem. In Section 4.1, we focus on the *clauses-to-variables ratio.* We mention that for randomly generated *SAT-CNF* instances there is a transition phenomenon (from satisfiability to unsatisfiability) when *clauses-to-variables ratio* approaches $M = 4.26$. We investigate if this phenomenon also happens for the *SAT-CNF* instances generated both from the *Integer-Factorization* and *OWA-Winner* problem instances. In Section 4.2, we analyze *running time* of the *PicoSAT* solver on the generated instances. We show *running time* may behave in really different ways on apparently quite similar problem instances for the *OWA-Winner* problem.

We conclude our work in Chapter 5.

2 Preliminaries

We assume that the reader is familiar with basic notions regarding mathematics, logic and complexity theory. First, in Section 2.1, we recall notions needed to understand the *SAT* problem. Then, in Section 2.2, we briefly discuss the *Integer-Factorization* problem with its applications and basic factorization algorithms. Finally, In Section 2.3, we give an overview of the *OWA-Winner* problem and formulate this problem as an integer linear program.

2.1 The Boolean Satisfiability and CNF

In this section we give a formal definition of concepts related to the boolean satisfiability and conjunctive normal form. The definitions below can be easily found in introductory mathematical logic and theory of computation textbooks such as a classical textbook on the computational complexity by Christos H. Papadimitriou (Chapter 4) [Pap95]. We define formally what we mean by a boolean formula.

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

1. a boolean variable (a 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 simplify the 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. $\Psi(F_1 \wedge F_2, \psi) = \begin{cases} \text{TRUE} & \text{if } \Psi(F_1, \psi) = \text{TRUE} \text{ and } \Psi(F_2, \psi) = \text{TRUE} \\ \text{FALSE} & \text{otherwise} \end{cases}$
5. $\Psi(F_1 \vee F_2, \psi) = \begin{cases} \text{TRUE} & \text{if } \Psi(F_1, \psi) = \text{TRUE} \text{ or } \Psi(F_2, \psi) = \text{TRUE} \\ \text{FALSE} & \text{otherwise} \end{cases}$
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. We give a 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.

To illustrate the definition below we give an example of a 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, the assignment $x_1 = \text{TRUE}$ and $x_2 = \text{TRUE}$ is a satisfying assignment.

There are also boolean formulas with no satisfying assignment. One of such formulas is shown as an example.

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. We can brute force through all the possible assignments to verify that. Setting $x_1 = \text{TRUE}$ and $x_2 = \text{TRUE}$ causes $(\overline{x_1} \vee \overline{x_2})$ to evaluate to FALSE which makes the whole formula also FALSE. Similarly, setting $x_1 = \text{TRUE}$ and $x_2 = \text{FALSE}$ causes $(\overline{x_1} \vee x_2)$ to evaluate to FALSE and so on. Careful reader may verify that every possible assignment falsifies exactly one of the disjunctions from which the formula is constructed. The whole formula is a conjunction of disjunctions and evaluates to FALSE if at least one of the disjunctions evaluates to FALSE.

A boolean variable or its negation is also called a *literal*. Both x and $\overline{x}$ are literals. A disjunction of literals is called a *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. $F \equiv \bigwedge_{i=1}^{m} c_i$, where each c_i is a clause.

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* formula. 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 *CNF.*

Remark 1. Every boolean formula can be transformed into *CNF*, but this transformation may require exponentially more clauses than the original formula has if we want to preserve the set of satisfying assignments. Sometimes we may consider transformations that do not preserve exact satisfying assignments but are still useful. One of such transformations is the so-called Tseytin transformation [Tse68]. The result of this transformation is a 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 a 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 the subformula derived from its children. For instance, consider the formula $F_1 \rightarrow F_2 \vee F_3$ (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_1 to represent F_1, 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 r, we need to assure that the 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 $n \in \mathbb{Z}$ 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 a composite number. 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 a special importance to us because we consider boolean formulas generated for *Integer-Factorization* of n-bit integers in the following chapters.

The *Integer-Factorization* problem is not only an interesting mathematical problem. It is important in public-key cryptography since one of the most popular cryptosystems, named RSA, is relying on the hardness of factoring integers. The acronym RSA is made of the initial letters of the surnames of Ron Rivest, Adi Shamir, and Leonard Adleman, who first publicly described the algorithm in 1977 [RSA78]. In the RSA cryptosystem we use a big integer n, which is a composite number that can be factored into exactly two prime numbers $n = pq$, as a public key. Everyone can know n and use it to encrypt a message. However, decrypting the message requires knowing the prime factors p and q. If factoring were easy, then clearly one could factor the number n and recover p and q, breaking the whole RSA cryptosystem.

There is a truly brilliant algorithm that factors the integers in polynomial time. This algorithm leverages the power of a quantum computing model of computation and was introduced by Peter Shor [Sho97]. Over 20 years has passed since then and

we are still not able to engineer the quantum computers that are able to endanger the RSA cryptosystem. Thus, we are still much better off trying classical algorithms.

We briefly give examples of classic integer-factorization algorithms. The simplest possible algorithm is a so-called trial division algorithm. In this algorithm we simply try all the odd numbers d such that $d \leq \sqrt{n}$. If a number d divides n then it is a factor of n by definition. We also need to account for a fact that a power of d may divide n. This is simply done by dividing n by d multiple times till the resulting rest is not 0. After this procedure, we are left with a list of odd factors of n. What remains in the process is a power of 2. This simple method of factoring integers requires $\Theta(\sqrt{n})$ divisions to be performed, thus the algorithm is exponential in the number of bits of n. Another elementary approach of factoring integers is the so-called Fermat's factorization method. It is named after the famous French mathematician Pierre de Fermat. This simple method relies on representing n as a difference of squares $n = a^2 - b^2 = (a - b)(a + b)$. If neither factor is 1, then it is a proper factorization of n. It is crucial that each odd number can be represented as a difference of squares. To demonstrate this fact let $n = lm$, then

$$n = \left(\frac{l+m}{2}\right)^2 - \left(\frac{l-m}{2}\right)^2$$

Since n is odd, then necessarily l and m are also odd, thus the fractions in parentheses are integers and we get a desired representation as a difference of squares. The task is to find a and b such that $n = a^2 - b^2$. In a basic method we pick $a = \lceil \sqrt{n} \rceil$. If we are lucky then $a^2 - n$ is a perfect square. In this case we can use $b^2 = a^2 - n$. We may need to keep on incrementing a till we get a perfect square. For example, to factor $n = 5959$, we shall start with $a = \lceil \sqrt{5959} \rceil = 78$. $78^2 - 5959 = 125$, which is not a perfect square. We try with $a = 79$. $79^2 - 5959 = 282$, which is still not a perfect square. When trying $a = 80$, we get $80^2 - 5959 = 441 = 21^2$. Therefore, we can use $a = 80$, $b = 21$, which leads to the following factorization of $n = 5959 = (80 - 21)(80 + 21) = 59 \cdot 101$. This basic method can be improved as demonstrated by James McKee [McK99]. There are lots of different factoring algorithms including the Pollard's rho algorithm [Pol75] and the sieve algorithms, such as the general number field sieve and the quadratic sieve.

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 [SFL16] and is related to voting and elections. Let us consider the following problem: There is a collection of items (e.g., facultative academic courses) and a group of agents (e.g., students); each agent has some intrinsic utility for each of the items. Our goal is to pick a set of K items that maximize the total derived utility

[1] OWA stands for Ordered Weighted Average

of all the agents (i.e., in our example we are to pick K facultative academic courses that we teach a group of students in the upcoming semester). The actual utility that an agent derives from a given item is only a fraction of its intrinsic one, and this fraction depends on how the agent ranks the item among the chosen, available, ones. There are lots of real-world problems that are related to selecting a set of items for a group of agents. Examples include voting for a parliament or deciding which laptop models to buy depending on the preferences of employees working in a company.

The formal setting of the *OWA-Winner* problem is presented below. Given a set of n agents $N = \{1, 2, ..., n\}$, a set of m items $A = \{a_1, a_2, ..., a_m\}$ and an integer K 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 of agent i derived from set W is measured as an ordered weighted average of this agent's utilities for these items. An *ordered weighted 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 the OWA operator in the following way:

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

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 the ordered weighted operator is, we focus on formalizing the problem of computing "the most satisfying set of K items" as follows.

Definition 6. [SFL16] In the OWA-Winner problem we are given a set $N = [n] = \{1, 2, ..., n\}$ of agents, a set $A = \{a_1, ..., a_m\}$ of items, a collection of agent's utilities $(u_{i,a_j})_{i \in [n], a_j \in A}$, a positive integer $K (K \leq m)$, and a K-number $\mathrm{OWA}\, \alpha^{(K)}$. 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. [SFL16] . In this thesis, we reconsider this translation and provide corrections to minor errors present in the original.

Theorem 1. *[SFL16] OWA-Winner problem can be stated as the following integer linear program:*

$$\text{maximize} \sum_{i=1}^{n} \sum_{j=1}^{m} \sum_{k=1}^{K} \alpha_k u_{i,a_j} x_{i,j,k}$$

subject to :

$$(a): \sum_{i=1}^{m} y_i = K$$
$$(b): x_{i,j,k} \leq y_j \qquad , i \in [n]; j \in [m]; k \in [K]$$
$$(c): \sum_{j=1}^{m} x_{i,j,k} = 1 \qquad , i \in [n]; k \in [K]$$
$$(d): \sum_{k=1}^{K} x_{i,j,k} \leq 1 \qquad , i \in [n]; j \in [m]$$
$$(e): \sum_{j=1}^{m} u_{i,a_j} x_{i,j,k} \geq \sum_{j=1}^{m} u_{i,a_j} x_{i,j,(k+1)} \qquad , i \in [n]; k \in [K-1]$$
$$(f): x_{i,j,k} \in \{0,1\} \qquad , i \in [n]; j \in [m]; k \in [K]$$
$$(g): y_j \in \{0,1\} \qquad , j \in [m]$$

We have that $[n] = \{1, 2, ..., n\}$ 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.

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

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

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

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 derive 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 on the *ILP* formulation of the OWA-Winner problem will be very

useful when designing a *SAT-CNF* encoding of the *OWA-Winner* problem. It is sufficient to reduce the decision version of integer linear programming to *SAT-CNF* to get the *SAT-CNF* encoding of the *OWA-Winner* problem.

3 Reducing Selected Computational Problems to SAT-CNF

In this chapter, we present the detailed description of how to reduce both the *Integer-Factoriazation* and *OWA-Winner* problems to *SAT-CNF*. We first start by defining notions used to express boolean constraints.

3.1 Basic Notions and Definitions Used to Express Boolean Constraints

Below we introduce vocabulary used in the following sections 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 a fixed size. The length of a sequence is a number of boolean variables associated with a sequence. It is denoted as $\text{length}(\langle x_1, x_2, .., x_n \rangle) = n$

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.

3.2 Reducing Integer Factorization to SAT-CNF

Since the *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

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

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

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

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 to a given integer I. For example, we may wish that our factor X (represented by the sequence) is not equal to 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

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

I is a sequence of bits and X is a sequence of boolean variables. Therefore, these objects cannot be directly compared. We introduce an auxiliary sequence $Y = \langle y_1, y_2, .., y_n \rangle$, where $y_j = x_j$ if the j-th bit of I (i_j) is 0 (If the j-th bit of I (i_j) is 1 then $y_j = \overline{x_j}$). The crucial observation is that setting $y_j = 0$ enforces $x_j = i_j$. This is exactly what we wrote when defining Y. If there exists j such that $y_j = 1$ then $x_j \neq i_j$ and as a consequence $X \neq I$. We can encode this existential condition as a boolean formula

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

To better illustrate the constraint introduced above we examine an example.

Example 4. Let $I = 13 = 1 \cdot 2^0 + 0 \cdot 2^1 + 1 \cdot 2^2 + 1 \cdot 2^3 = \langle 1, 0, 1, 1 \rangle$ and $X = \langle x_1, x_2, x_3, x_4 \rangle$. Constraint $X \neq I$ can be encoded as $(\overline{x_1} \vee x_2 \vee \overline{x_3} \vee \overline{x_4})$. The only way to make $(\overline{x_1} \vee x_2 \vee \overline{x_3} \vee \overline{x_4})$ evaluate to 0 is to set $x_1 = 1$, $x_2 = 0$, $x_3 = 1$, $x_4 = 1$ and this is the only assignment that leads to $X = I$. A satisfying assignment to $(\overline{x_1} \vee x_2 \vee \overline{x_3} \vee \overline{x_4})$ leads to $X \neq I$, which is what we wanted.

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 ($\mathbf{bX = Y}$)

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 an additional sequence C representing carry bits:

$$C = \langle c_1, c_2, .., c_{n+1} \rangle$$

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

c_{n+1}	c_n	...	c_2	c_1
	x_n	...	x_2	x_1
+	y_n	...	y_2	y_1
	z_n	...	z_2	z_1

For the whole addition to be valid, we require that c_1 and c_{n+1} are both 0 (FALSE), and c_i is 1 (TRUE) if at least two of $\{x_{i-1}, y_{i-1}, c_{i-1}\}$ are 1. This enforces the following conditions: $(x_{i-1} \wedge y_{i-1}) \Rightarrow c_i$, $(x_{i-1} \wedge c_{i-1}) \Rightarrow c_i$, $(y_{i-1} \wedge c_{i-1}) \Rightarrow c_i$. We can also write these conditions by transforming the implication into the logical disjunction using the logical tautology $(p \Rightarrow q) \iff (\overline{p} \vee q)$. We obtain $(c_i \vee \overline{x_{i-1}} \vee \overline{c_{i-1}}) \wedge (c_i \vee \overline{x_{i-1}} \vee \overline{y_{i-1}}) \wedge (c_i \vee \overline{y_{i-1}} \vee \overline{c_{i-1}})$. We can also conclude that if at least two of $\{x_{i-1}, y_{i-1}, c_{i-1}\}$ are 0 then c_i is 0. The way of converting this constraint into *SAT-CNF* formula works as above and yields $(\overline{c_i} \vee x_{i-1} \vee c_{i-1}) \wedge (\overline{c_i} \vee x_{i-1} \vee y_{i-1}) \wedge (\overline{c_i} \vee y_{i-1} \vee c_{i-1})$. The 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, which enforces the following clauses: $(x_i \wedge \overline{y_i} \wedge \overline{c_i}) \Rightarrow z_i$, $(\overline{x_i} \wedge y_i \wedge \overline{c_i}) \Rightarrow z_i$, $(\overline{x_i} \wedge \overline{y_i} \wedge c_i) \Rightarrow z_i$, $(x_i \wedge y_i \wedge c_i) \Rightarrow z_i$. It is easy to verify that these clauses can be converted by the rules of logic into $(z_i \vee y_i \vee x_i \vee \overline{c_i}) \wedge (z_i \vee y_i \vee \overline{x_i} \vee c_i) \wedge (z_i \vee \overline{y_i} \vee x_i \vee c_i) \wedge (z_i \vee \overline{y_i} \vee \overline{x_i} \vee \overline{c_i})$. The value of z_i is 0 if either all $\{x_i, y_i, c_i\}$ are 0 or exactly two of them are 1. By following the same procedure of encoding this constraint as above, we obtain $(\overline{z_i} \vee y_i \vee x_i \vee c_i) \wedge (\overline{z_i} \vee y_i \vee \overline{x_i} \vee \overline{c_i}) \wedge (\overline{z_i} \vee \overline{y_i} \vee x_i \vee \overline{c_i}) \wedge (\overline{z_i} \vee \overline{y_i} \vee \overline{x_i} \vee c_i)$. All the conditions we have already discussed are necessary and sufficient conditions to get a valid addition of boolean sequences encoded. Careful reader may want to verify himself if this is really the case. We give a boolean formula representing addition below.

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

$$
\begin{aligned}
&(\overline{c_1}) \wedge (\overline{c_{n+1}}) \\
&\wedge \bigwedge_{i=2}^{n+1} ((\overline{c_i} \vee x_{i-1} \vee c_{i-1}) \wedge (\overline{c_i} \vee x_{i-1} \vee y_{i-1}) \wedge (\overline{c_i} \vee y_{i-1} \vee c_{i-1}) \\
&\wedge (c_i \vee \overline{x_{i-1}} \vee \overline{c_{i-1}}) \wedge (c_i \vee \overline{x_{i-1}} \vee \overline{y_{i-1}}) \wedge (c_i \vee \overline{y_{i-1}} \vee \overline{c_{i-1}})) \\
&\wedge \bigwedge_{i=1}^{n} ((z_i \vee y_i \vee x_i \vee \overline{c_i}) \wedge (z_i \vee y_i \vee \overline{x_i} \vee c_i) \wedge (z_i \vee \overline{y_i} \vee x_i \vee c_i) \wedge (z_i \vee \overline{y_i} \vee \overline{x_i} \vee \overline{c_i}) \\
&\wedge (\overline{z_i} \vee y_i \vee x_i \vee c_i) \wedge (\overline{z_i} \vee y_i \vee \overline{x_i} \vee \overline{c_i}) \wedge (\overline{z_i} \vee \overline{y_i} \vee x_i \vee \overline{c_i}) \wedge (\overline{z_i} \vee \overline{y_i} \vee \overline{x_i} \vee c_i))
\end{aligned}
$$

Encoding Multiplication (PQ = N)

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

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

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

$$pq = q_1 p + q_2 2p + q_3 2^2 p + ... + q_k 2^{k-1} p$$

We extend the notion of multiplication to the sequences of boolean variables. 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:

$$PQ = q_1 P + q_2 2P + q_3 2^2 P + ... + q_k 2^{k-1} P$$

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 a *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 the *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 the explicit *CNF* formula, we take an 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. In 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. For example, $CNF(A = B+C)$ means the *CNF* encoding of an addition $A = B+C$. In the algorithm, we introduce some auxiliary variables and then incrementally we use these variables to build an encoding for the multiplication PQ. Please follow the comments next to the algorithm steps to understand what is encoded by each step. Last two for loops are there to fix some variables in P and Q that have to have fixed values due to the nature of multiplication. This can be useful for efficiency reasons when solving these formulas with SAT solvers.

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

$f \leftarrow \epsilon$ // starting with an empty SAT-CNF formula (no clauses)
$f \leftarrow CNF(S_0 = P) \wedge f$ // generating encoding of P
for $i = 1$ **to** $lq - 1$ **do**
  $f \leftarrow CNF(S_i = 2S_{i-1}) \wedge f$ // generating encodings of $2P$, 2^2P,.., $2^{lq-1}P$
end for
for $i = 0$ **to** $lq - 1$ **do**
  $f \leftarrow CNF(M_i = Q_iS_i) \wedge f$ // generating encodings of PQ_0, $2PQ_1$, 2^2PQ_2,.., $2^{lq-1}PQ_{lq-1}$
end for
$f \leftarrow CNF(R_0 = M_0) \wedge f$ // initializing partial sum encodings $R_0 = PQ_0$
for $i = 1$ **to** $lq - 1$ **do**
  $f \leftarrow CNF(R_{i-1} + M_i = R_i) \wedge f$ // more partial sum encodings, $R_i = R_{i-1} + 2^iPQ_i$, this requires carry=C_{i-1}
end for
$f \leftarrow CNF(R_{lq-1} = N) \wedge f$ // encoding of the fact that last partial sum is N, $N = PQ_0 + 2PQ_1 + .. + 2^{lq-1}PQ_{lq-1}$
for each pair $(i, j) \in [0, 1, .., ln - 1] \times [0, 1, .., lq - 1]$ **do**
  if $i + j \geq ln$ **then**
    $f \leftarrow (\overline{P_i} \vee \overline{Q_j}) \wedge f$ // to ensure that the 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**
    $f \leftarrow (\overline{Q_i}) \wedge f$ // Limiting the 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 the *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:

$$Q \neq 1, P \neq 1$$

Up to this point, we have shown all steps necessary to convert an arbitrary *Integer-Factorization* problem instance to a 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

3.3 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 2.

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 comparing binary expansions of integers. If we treat X and Y in the same way as we treat the binary expansions of integers, we can write the following definition 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. We encode $x_i < y_i$ as $(\overline{x_i} \wedge y_i)$. We also encode $x_i = y_i$ using the rules of logic as $x_i \Leftrightarrow y_i \iff (x_i \Rightarrow y_i) \wedge (y_i \Rightarrow x_i) \iff (\overline{x_i} \vee y_i) \wedge (\overline{y_i} \vee x_i)$

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 using the definition of $X \leq Y$ (going from inside to outside) and the fact that we know the encodings of $x_i < y_i$ and $x_i = y_i$.

Algorithm 3.2 Encoding $X \leq Y$

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

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

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

$$X = \langle x_1, x_2, .., x_n \rangle$$

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

$$I = \langle i_1, i_2, .., i_n \rangle$$

We want to encode $X \leq I$, which is a special case of $X \leq Y$. We define $X \leq I$

$$X \leq I \iff (x_n < i_n) \vee (x_n = i_n \wedge (x_{n-1} < i_{n-1} \vee ...(x_1 = i_1 \vee (x_1 < i_1))))$$

In Algorithm 3.3, we also go from the inside to the outside (we first consider $(x_1 = i_1 \vee (x_1 < i_1))$). This time, however, i_j's are constants (values) and not variables, so we use them during the encoding process. In the line 1 of Algorithm 3.3, we check if the least significant digit of I (i_1) is 0. If this is true, then we demand $\overline{x_1}$. This is because allowing x_1 may lead to a situation when on all other positions, except from the least significant position, $X = I$ but the least significant digit may be $i_1 = 0 < 1 = x_1$ and hence causing $X > I$, which is against the requirement we want to encode: $X \leq I$. Therefore, it is only safe to demand $\overline{x_1}$. If i_1 is 1 then it does not matter what we do with the least significant position in X. Therefore we may as well do not take any decision and simply assign a formula always true to f $(x_1 \vee \overline{x_1})$. Starting from the line 6 of the algorithm, we keep building the formula f by extending what we have already encoded and taking the decisions similar to what we took for the least significant position/variable/bit. We can also think about the whole process of encoding in another way. We simply take the encoding of $X \leq Y$, where both X and Y are sequences, and we substitute concrete values for the variables in Y. After the substitution, we can perform the following transformations: $x_i < 0 \rightarrow \overline{x_i}$, $x_i < 1 \rightarrow (x_i \vee \overline{x_i})$ to get the *SAT* encoding.

Algorithm 3.3 Encoding $X \leq I$

```
1: if i_1 = 0 then
2:     f ← ¬x_1
3: else if i_1 = 1 then
4:     f ← x_1 ∨ ¬x_1
5: end if
6: for j = 2 to n do
7:     if i_j = 0 then
8:         f ← ¬x_j ∧ f
9:     else if i_j = 1 then
10:        f ← ¬x_j ∨ (x_j ∧ f)
11:    end if
12: end for
13: 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 [Sin05]. 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 [Sin05] the efficient encodings of "at most k of" and "at least k of" were given. Suppose we want to encode $_{\leq k}(\{x_1, x_2, .., x_n\})$. We introduce a matrix of additional variables s such that $s_{i,j} = 1$ iff among $\{x_1, x_2, .., x_i\}$ there are at least j variables set to TRUE ($_{\geq j}(\{x_1, x_2, .., x_i\})$). For example, $s_{3,2}$ indicates a subconstraint $_{\geq 2}(\{x_1, x_2, x_3\})$. Let us make some observations about $s_{i,j}$. If $x_i = 1$ and $s_{i-1,j-1} = 1$, then necessarily $s_{i,j} = 1$. Moreover, if $s_{i-1,j} = 1$, then clearly $s_{i,j} = 1$. These two observations together with boundary conditions allowed Sinz to encode $_{\leq k}(\{x_1, x_2, .., x_n\})$. Below we state the theorem about encoding "at most k of" efficiently. The meaning of variables is as discussed above.

Theorem 2. *[Sin05] Encoding $LT^{n,k}_{SEQ}$ expressing $_{\leq k}(\{x_1, x_2, .., x_n\})$ $n > 1, k > 0$ can be stated as follows:*

$$(\overline{x_1} \vee s_{1,1})$$
$$(\overline{s_{1,j}}) \qquad\qquad 1 < j \leq k$$

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

In the Theorem 2 the variable $s_{i,j}$ asserts that among i variables $\{x_1, x_2, .., x_i\}$ at least 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 least 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 the *OWA-Winner* problem based on the *ILP* formulation from Chapter 2. Decision version of the *OWA-Winner* problem reduces to checking feasibility of the 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 the *OWA-Winner* problem, we finally present a way of encoding an arbitrary *OWA-Winner* problem instance

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 a sequence constructed from sum of products and an 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 like $x_{i,j,k} \leq y_j$. 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 a 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 present a 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 the *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 take the *SAT-CNF* encoding of a decision version of the general *OWA-Winner* problem. 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 know that $\alpha_k u_{i,a_j}$ has to be either 0 or 1 (α and u are binary in the approval version of the problem). This fact allows us to replace the summation $\sum_{i=1}^{n} \sum_{j=1}^{m} \sum_{k=1}^{K} \alpha_k u_{i,a_j} x_{i,j,k} \geq L$ with $\sum_{i=1}^{n} \sum_{j=1}^{m} \sum_{k=1}^{K} x_{i,j,k} \geq L$ for i,j,k such that $\alpha_k u_{i,a_j} = 1$. Finally, we can write the summation as a boolean cardinality constraint $_{\geq L}(\{x_{i,j,k} | i \in [n], j \in [m], k \in [K], \alpha_k u_{i,a_j} > 0\})$ □

4 Analysis of Structure and Solving Time of Obtained Formulas

In this chapter, we present a set of experimental results for some instances of the boolean formulas, generated based on our two example problems, the *Integer-Factorization* and the *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*). We also briefly analyze the running time it takes to solve these instances. 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 $(\overline{x_1} \wedge \overline{x_2} \wedge \overline{x_3}) \vee (x_1 \wedge \overline{x_2} \wedge \overline{x_3}) \vee (\overline{x_1} \wedge x_2 \wedge \overline{x_3}) \vee (\overline{x_1} \wedge \overline{x_2} \wedge x_3)$. It 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 the number of variables fixed can only make a formula more constrained (harder to satisfy). It turns out that for randomly generated[1] *SAT-CNF* formulas there is a "magic" 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].

[1] We generate the formulas in the following way: We pick the literals in each clause at random with equal probability $\frac{1}{2n}$, where n is the number of variables. If we get either both x and $\overline{x}$ or both x and x in the same clause, then we discard the clause and try to generate this clause again until we succeed.

Clauses to Variables Ratio for Integer Factorization Formulas

In the previous chapters we have shown all the steps necessary to reduce the *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. A 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[2] 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 in Figure 4.1

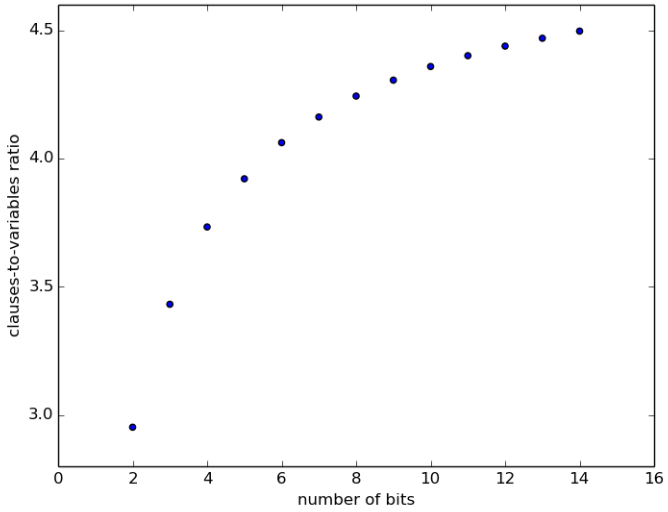

Figure 4.1: Clauses to Variables Ratio for Integer Factorization Formulas

By looking at Figure 4.1, we can notice that formulas generated for 10-bit numbers have already the *clauses-to-variables ratio* clearly above $M = 4.26$. The general observation is that the *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 the number of variables, the number of clauses and the *clauses-to-variables ratio* with respect to the number of bits.

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

number of bits	number of variables	number of clauses	ratio
1	6	11	1.833
2	21	62	2.952
3	44	151	3.432
4	75	280	3.733
5	114	447	3.921
6	161	654	4.062
7	216	899	4.162
8	279	1184	4.244
9	350	1507	4.306
10	429	1870	4.359
11	516	2271	4.401
12	611	2712	4.439
13	714	3191	4.469
14	825	3710	4.497

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

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

To find the formula 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 a 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 the *Integer-Factorization* behaves in a completely different way than a set of randomly generated instances.

Clauses to Variables Ratio for OWA-Winner Formulas

Since the *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 the *OWA-Winner* problem (or its approval variant). We adopt the following notation to represent information about problem instance(s):

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

where kBestOWAApprovalWinner is the type of the problem, N is the number of agents, M is the number of items, K is the size of a committee (the number of chosen items), 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). Agent's utilities μ (votes) were generated as follows: For each agent A and for each item C, we add 1 with probability p to the agent A utilities list on the position with item C. This means that we simulate a situation in which agents are voting randomly for their candidates and choose (approve) each candidate independently with probability p.

Consider a set of boolean formulas corresponding to the following set of problem instances, where μ are concrete agent's utilities generated in the procedure described above:

$$S_1 = \{\text{kBestOWAApprovalWinner}(50, 12, 6, \mu, 4, 0.3, v) | \mu - \text{agent}'\text{s utilities}, v \in [200]\}$$

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.

In Figure 4.2, we can see that for this particular *OWA-Winner* problem instance (represented by S_1) *clauses-to-variables ratio* increases as v increases, but is below 2.0 for all the considered target values (i.e. utility function values). From the *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 behavior observed for the randomly generated *SAT-CNF* instances.

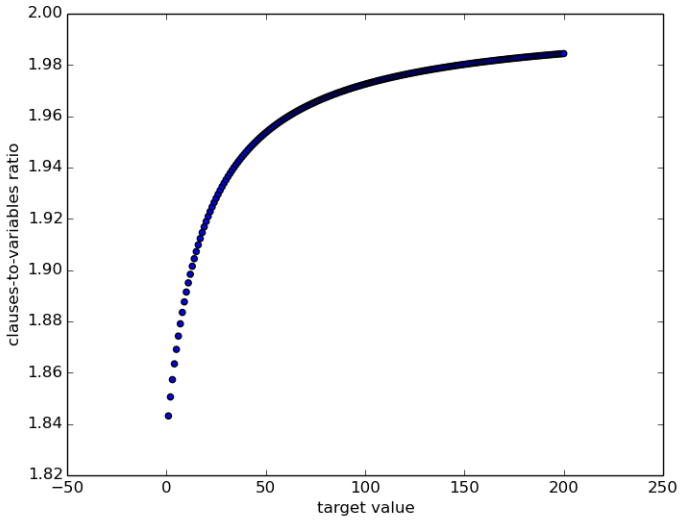

Figure 4.2: Clauses To Variables ratio for a first instance of k-Best-OWA-Approval-Winner problem

Let us also consider another example:

$$S_2 = \{\text{kBestOWAApprovalWinner}(100, 24, 10, \mu, 1, 0.3, v) | \mu - \text{agent's utilities}, v \in [200]\}$$

The purpose of this is to examine how the *clauses-to-variables ratio* changes when the *OWA* vector contains only one non-zero element. Please note that the number of voters and candidates is also bigger than in the previous example S_1.

In Figure 4.3, we can see that the *clauses-to-variables ratio* also increases as the target value increases for S_2. A target value 100 is the maximum value that can be reached for S_2. It is interesting to see that the *clauses-to-variables ratio* seems to be approaching 2.0 as in the previous example.

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 the *PicoSAT* solver to evaluate the *running time* it takes to decide satisfiability/unsatisfiability of some specific boolean formula instances.

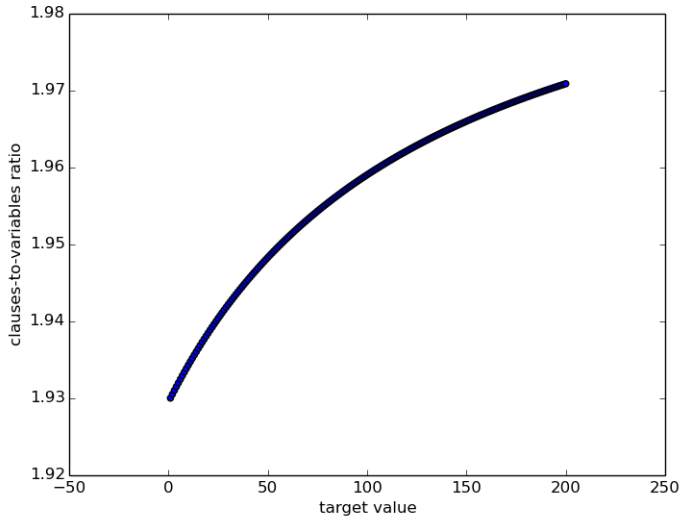

Figure 4.3: Clauses To Variables ratio for a second instance of k-Best-OWA-Approval problem

Running Time for Integer Factorization Formulas

It is obvious that factoring huge integers (e.g., RSA keys) is not really feasible using *SAT* solvers unless there is an efficient polynomial time algorithm for *SAT-CNF* or there is a specific structure in the *Integer-Factorization* problem that makes it polynomial-time solvable. In this paragraph we consider the *running time* it takes to factorize integers (or decide unsatisfiability) from range 100000000 to 100000100. All the numbers in this range have the same size in terms of the number of bits. Therefore they all lead to the boolean formulas of the same size for the *Integer-Factorization* problem. The goal is to observe the *running time* of our *PicoSAT* solver for these instances.

In Figure 4.4, we can see that time it takes to factor integers from range 100000000 to 100000100 by using the *PicoSAT* solver varies greatly from nearly 0 seconds to above 2 seconds. The color indicates the number of factors. It seems that prime numbers and integers that have a small number of factors generate much harder *SAT* instances than integers with a large number of factors. This is not always true because there are other things that also may affect the running time, such as the size of the smallest factor.

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.*

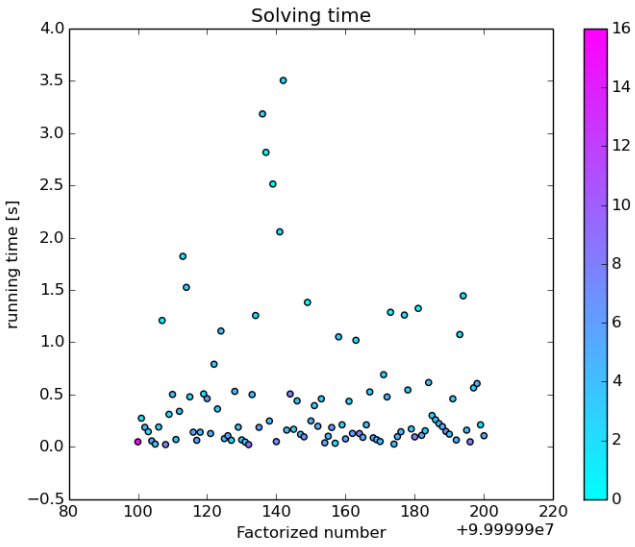

Figure 4.4: Running Time for Integer Factorization instances generated from numbers in range 100000000-100000100

$$S_1 = \{\text{kBestOWAApprovalWinner}(50, 12, 6, \mu, 4, 0.3, v) | \mu - \text{agent}'\text{s utilities}, v \in [200]\}$$

The goal is to observe how *running time* is changing when v increases. In Figure 4.5, we can see that *running time* (in seconds) is close to 0 seconds for target values below 70. It suddenly jumps up to near 100 seconds (this means that finding a satisfying assignment is getting really hard). At target value 108 the 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 time* 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.

To see how the *running time* behaves for the example with an *OWA* vector that contains only one non-zero element, let us consider

$$S_2 = \{\text{kBestOWAApprovalWinner}(100, 24, 10, \mu, 1, 0.3, v) | \mu - \text{agent}'\text{s utilities}, v \in [200]\}$$

In Figure 4.6, we can see that the *running time* increases with the target value and there is some noise/variance. We see a jump/spike to 12 seconds near the satisfiability threshold/optimal value, which is 100 in this case. However, the behavior is different compared to the problem instance S_1 because after the big jump near the optimal value *running time* decreases again.

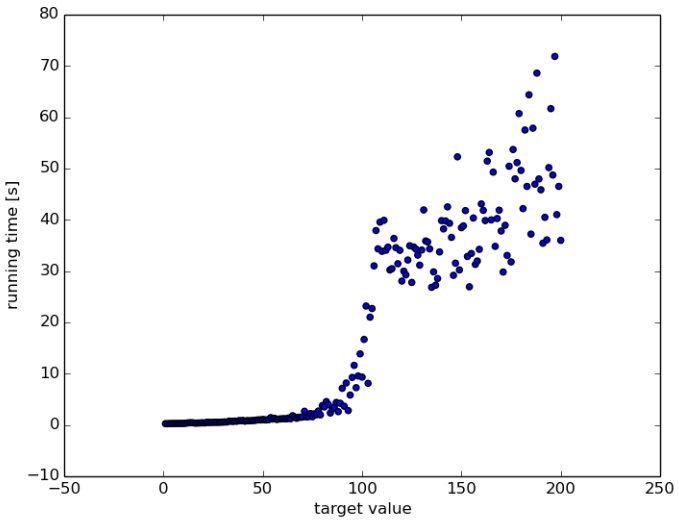

Figure 4.5: Running Time for a first instance of k-Best-OWA-Approval-Winner problem

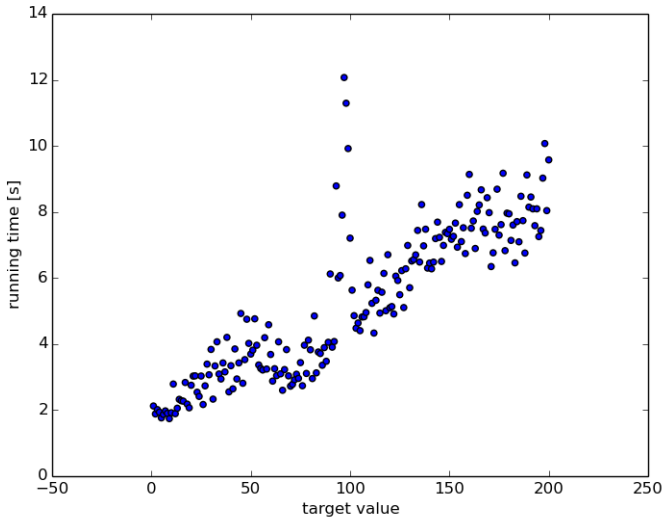

Figure 4.6: Running Time for a second instance of k-Best-OWA-Approval-Winner problem

5 Conclusions

The main goal of this thesis was reducing a decision version of the *OWA-Winner* problem to the *SAT-CNF*. This goal was achieved by first developing a way of encoding various boolean constraints. To reach this goal, the *Integer-Factorization* problem was first reduced to the *SAT-CNF*. This involved encoding the arithmetic constraints (addition, multiplication, shift of sequences) among others. In the reduction from the *OWA-Winner* problem to the *SAT-CNF,* we took advantage of efficient boolean cardinality constraints encodings developed by Sinz [Sin05]. In addition to this we based our reduction on an integer linear programming formulation of the *OWA-Winner* problem that is available in a paper by Skowron, Faliszewski and Lang [SFL16]. We gave two reductions of the *OWA-Winner* problem to the *SAT-CNF*. The first one was for the general *OWA-Winner* problem (its decision version). The second one was for the approval version of the *OWA-Winner* problem and was considerably more efficient in terms of the number of clauses and variables produced. This efficiency gain was due to the fact that the approval version is just a very special case of the general *OWA-Winner* problem.

As a by-product of this thesis, a set of Python modules was created, that can be used to convert any decison version instance of integer linear programming into a *SAT-CNF* formula and even more. For example, software developed to aid the work on this thesis allows to encode the following constraints: $X^3 + Y^3 = Z^3$, $X > 0$, $Y > 0$, $Z > 0$, where all X, Y, Z are boolean sequences of the same fixed length (have the same number of boolean variables). Of course these constraints cannot be all satisfied at the same time and hence lead to unsatisfiable *SAT-CNF* instances.

In Chapter 4, we have explored a bit some of generated *SAT-CNF* instances for the *Integer-Factorization* problem and *OWA-Winner* problems. We have shown that the *clauses-to-variables ratio* goes to $\frac{39}{8}$, for the generated from the *Integer-Factorization* problem *SAT-CNF* instances, as the number of bits of an integer to be factorized goes to infinity. Since $\frac{39}{8} > 4.26$, then there are clearly *SAT-CNF* instances generated for the *Integer-Factorization* problem that are satisfiable even though they have bigger *variable-to-clauses ratio* than the so-called satisfiability threshold that applies to the randomly generated *SAT-CNF* instances. In this respect these instances behave in a different way than randomly generated instances. We have explored two *SAT-CNF* instances generated based on the *OWA-Winner* problem instances and we have concluded that the *clauses-to-variables* for both of them is increasing and seems to be approaching 2. However, we have never proved rigorously that the limit is exactly 2.

Then we have considered *running time* of the *PicoSAT* solver on the selected instances. For the *OWA-Winner* problem, we used exactly the same instances. It

was interesting to see that *running time* was behaving differently for the first and the second instance. The only common behavior was the jump in *running time* around the maximum feasible target value of the utility function. For the *Integer-Factorization* problem instances, we have experimentally illustrated that the number of factors seems to have an impact on *running time* of the *PicoSAT* solver. Roughly speaking, the smaller the number of factors is, the longer it takes to solve the problem.

For the future work, we recommend more in-depth analysis of the properties of obtained boolean formulas. There are simply so many possible instances to be generated based on the instances of the *OWA-Winner* problem and we may expect very different behavior in terms of metrics such as *running time* of a *SAT* solver. It is probably worth considering other, more sophisticated structural properties of boolean formulas than the *clauses-to-variables ratio* alone.

Acknowledgments

Firstly I would like to thank my family for all the love and support. I wish to thank my supervisor dr hab. inż. 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.

Bibliography

[BGV99] Randal Bryant, Steven German, and Miroslav Velev. Microprocessor Verification Using Efficient Decision Procedures for a Logic of Equality with Uninterpreted Functions. In *Automated Reasoning with Analytic Tableaux and Related Methods*, pages 1–13, 1999.

[Coo71] Stephen Cook. The complexity of theorem-proving procedures. In *STOC '71 Proceedings of the third annual ACM symposium on Theory of computing*, pages 151–158, 1971.

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

[Kau06] Henry Kautz. Deconstructing Planning as Satisfiability. In *AAAI'06 proceedings of the 21st national conference on Artificial intelligence*, volume 2, pages 1524–1526, 2006.

[Lar06] Tracy Larrabee. Test Pattern Generation Using Boolean Satisfiability. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 11(1): 4–15, November 2006, https://users.soe.ucsc.edu/ larrabee/ce224/tcad.sat.pdf.

[McK99] James McKee. Speeding Fermat's factoring method. *Mathematics of Computation*, 68(228): 1729–1737, October 1999.

[NSR02] Gi-Joon Nam, Karem Sakallah, and Rob 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): 674–684, June 2002.

[Pap95] Christos Papadimitriou. *Computational Complexity.* Addison Wesley Longman, 1995.

[Pol75] John Pollard. A Monte Carlo Method for Factorization. *BIT Numerical Mathematics*, 15(3): 331–334, September 1975.

[RSA78] Ronald Rivest, Adi Shamir, and Leonard Adleman. A Method for Obtaining Digital Signatures and Public-Key Cryptosystems. *Communications of the ACM*, 21(2): 120–126, February 1978.

[SFL16] Piotr Skowron, Piotr Faliszewski, and Jerome Lang. Finding a Collective Set of Items: From Proportional Multirepresentation to Group Recommendation. *Artificial Intelligence*, 241(C): 191–216, 2016.

[Sho97] Peter Shor. Polynomial-Time Algorithms for Prime Factorization and Discrete Logarithms on a Quantum Computer. *SIAM Journal on Computing*, 26(5): 1484–1509, October 1997.

[Sin05] Carsten Sinz. Towards an Optimal CNF Encoding of Boolean Cardinality Constraints. In *CP'05 Proceedings of the 11th International Conference on Principles and Practice of Constraint Programming*, pages 827–831, 2005.

[SML96] Bart Selman, David Mitchell, and Hector Levesque. Generating hard satisfiability problems. *Artificial Intelligence*, 81(1-2): 17–29, March 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] Gregory Tseytin. On the complexity of derivation in propositional calculus. *Studies in Constructive Mathematics and Mathematical Logic*, Part II: 115–125, 1968.

[ZLS04] Hantao Zhang, Dapeng Li, and Haiou Shen. A SAT Based Scheduler for Tournament Schedules. In *SAT 2004 - The Seventh International Conference on Theory and Applications of Satisfiability Testing*, May 2004.

Nomenclature

ILP	Integer Linear Programming
NP	Nondeterministic Polynomial
OWA	Ordered Weighted Average
SAT	Boolean Satisfiability Problem