A QBF Instance Generator for Chen Formulae: Implementation and Results

Noel Arteche Echeverría

Supervisor:
Montserrat Hermo

Contents

Abstract

In this document we present an implementation for the so-called Chen Formulas of type 1 and 2. These are two classes of QBF formulas presented by Hubie Chen in his article *Proof Complexity Modulo the Polynomial Hierarchy: Understanding Alternation as a Source of Hardness*. We propose a computer program for generating instances of these classes and checking their running times on popular QBF solvers.

1 Introduction

In his 2016 article *Proof Complexity Modulo the Polynomial Hierarchy: Understanding Alternation as a Source of Hardness*, Hubie Chen showed a few exponential lower bounds on certain proof systems for the problem of deciding the unsatisfiability of quantified Boolean formulas.

The first set of formulas, now named *Chen Formulas of Type 1*, is a class of formulas defined for every positive natural number n such that the resulting formula is always unsatisfiable. Surprisingly, though, these formulae have proofs of linear size in some proof systems while, for others, these formulas constitute an example of exponential lower bound on the system.

The second set of formulas, the *Chen Formulas of Type 2*, are a more subtle set of QBF instances defined by an arithmetic characterizing property.

In this document, we define both classes of formulae, we review the basic properties shown in Chen's article and propose a Python program that can generate the instances of any size and send them as input on popular QBF solvers available at the time of writing this report.

2 Preliminaries

In what follows we will try to briefly define and introduce the main concepts, tools and remarks from Hubie Chen's original article, though we strongly recommend a thorough read of the original paper for the complete definitions and additional formality. In this report we only recover the essential definitions and results, which, in addition, have been slightly modified in terms of notation.

2.1 Notation and general concepts

Let $\mathbb{N}^* = \mathbb{N} \setminus \{0\}$. For every $k \in \mathbb{N}^*$ we denote by $[k]$ the set $\{1, \ldots, k\}$.

A *literal* is a a propositional variable x or its negation, $\neg x$. A *clause* is a disjunction of literals that, for every variable, contains at most one literal of it. A Boolean propositional formula is said to be in *Conjunctive Normal Form* (CNF) if it is written as a conjunction of clauses.

A *Quantified Boolean Circuit* (QBC) Φ is a pair usually written as $\Phi = \vec{P} : \phi$, where $\vec{P}$ is a *quantifier prefix* or *quantifier vector*, $\vec{P} = Q_1 v_1 \ldots Q_n v_n$, such that $v_1, \ldots, v_n$ are propositional variables and $Q_1, \ldots, Q_n$ are quantifiers in $\{\exists, \forall\}$; and ϕ is a Boolean circuit built from constants 0 and 1, propositional variables $v_1, \ldots, v_n$ and the logic gates AND ($\wedge$), OR ($\vee$), NOT ($\neg$) and XOR ($\oplus$). The problem of deciding whether a given QBC Φ is false is called the *QBC problem*. In the particular case when the circuit ϕ is a Boolean formula instead of a circuit (usually in CNF), we refer to Φ as a *Quantified Boolean Formula* (QBF).

2.2 A few remarks on proof systems

We assume basic familiarity with the concept of a *proof system*. A fundamental question in the area of proof complexity and QBF solvers is related to the proof systems underlying state of the art solvers. After all, the execution trace of a QBF solver is no more than a proof of its validity. Nowadays, popular QBF solvers rely on a well-known proof system called *Q-resolution*. The Q-resolution proof system is an extension of the famous *Resolution* proof system used with propositional formulas, so that additional rules allow to solve formulas preceded by quantifiers. Due to the fact that when applied to a propositional formula without quantification (that is, existential quantification over all variables) Q-resolution behaves exactly as Resolution, the known exponential lower bounds of Resolution are directly applied to Q-resolution.

It has been shown, however, that despite the fact that Q-resolution is refutation complete for the QBF problem, not all logically implied clauses can be derived within it (see [?]). Due to this rather unfortunate situation, an slight extension of Q-resolution has been recently introduced, *QU-resolution*, which is complete in this stronger sense.

Fundamentally, QU-resolution is a proof system where one can derive new clauses in two ways: either by eliminating literals on universally quantified variables or by *resolving* two clauses on any variable, whereas in Q-resolution one can only resolve on existential variables. We shall remind that *resolving* means obtaining a new clause γ from existing clauses α, β such that some variable v appears as a literal in α and as a negated literal in β. Particularly, the *resolvent* clause γ is $\gamma = (\alpha \setminus \{v\}) \cup (\beta \setminus \{\neg v\})$.

As Chen points out, both Q-resolution and QU-resolution are usually defined for QBF instances in CNF or clausal form. With the aim of parameterizing QU-resolution so that it can work with circuits, the notion of a *proof system ensemble* is introduced, as well as a particular example, *Relaxing QU-res*, which is the desired parametrization.

More precisely, a *proof system ensemble* (see Definition 3.1 in [1]) is an infinite collection of proof systems, where for each proof system, one can determine whether or not a given string π is a proof of a given formula Φ by calling an oracle of some level in the polynomial hierarchy (PH).

Now let Φ be a QBF. We define an *axiom* of Φ to be a clause that is in a certain precise sense entailed by Φ. Given a partial assignment a to some of the variables in Φ, if it turns out that $\Phi[a]$ is false, then the unique clause that made Φ false is an axiom.

Since our proof system ensemble may call an oracle from some level of the PH, then the QBF problem restricted to a certain alternation may be used as an oracle. Ideally, we could derive axioms by calling one of this oracles, but in general they may have many alternations and this may not be feasible. Thus, we have the notion of a *relaxation* of a QBF, such that whenever the relaxation is false, the original QBF is false. In very general terms, the relaxation is obtained from the original Φ by changing the order of the quantifiers in the prefix, such that universal quantifiers can be moved to the left and existential quantifiers can be moved to the right. This way, one can easily obtain relaxations and then call an oracle at an affordable level of the polynomial hierarchy.

Following Chen's own overview, for each $k \geq 2$ we define the set $H(\Phi, \Pi_k)$ to contain the axioms than can be derived tom QBFs $\Phi[a]$ having a Π_k relaxation (relaxations with a Π_k prefix, where Π_k is the regular expression $\forall^*\exists^* \ldots \forall^*\exists^*$ with k starred quantifiers, which is related to the level Π_k in PH). If we consider the sequence of axiom sets for increasing values of k we have

$$H(\Phi, \Pi_2) \subseteq H(\Phi, \Pi_3) \subseteq H(\Phi, \Pi_4) \subseteq \ldots$$

One could then work under the rules of QU-resolution on each of those levels. Thus, for every k we have a different version of QU-resolution. The union of all those proof systems is a proof system ensemble, as defined earlier, which we call *Relaxing QU-res*.

We should note that relaxing QU-res is polynomially bounded (in the sense that a proof system ensemble may be bounded) on any set of false QBFs having bounded alternation.

Finally, we shall mention that a QU-resolution proof π for a QBC Φ may be translated to a directed acyclic graph $G(\pi)$ (see Proposition 4.9 in [1]). Whenever $G(\pi)$ is a tree we say that the proof π is *treelike*.

The previous paragraphs were a brief rephrasing of Chen's original Section 1 and its definitions. For detailed information regarding the concepts discussed here, consult the original paper.

2.3 Overview of the original results

In his article ([1]), Chen defines two sets of QBF instance. The first set, what we now call Chen Formulas of Type 1, is used to show and exponential separation between the treelike and the general version of relaxing QU-res. This is an alternation-based analog fo the known separation between treelike and general Resolution. Since treelike relaxing QU-res can be viewed ast hte traces of natural backtrack on QBF solving, this shows the inherent power of QU-resolution against simple backtracking, since QU-resolution can solve Chen Formulas of Type 1 in linear size.

On the other hand, Chen Formulas of Type 2 are used to show an exponential lower bound on the relaxing QU-res proof system per se.

3 Chen Formulae of Type 1

3.1 Definition and basic properties

The Type 1 formulae, as defined by Chen in section 6 of his article, are the following:

Definition 1 *For every positive natural number n and every $i \in \{0\} \cup [n]$, let X_i be the set of variables $\{x_{i,j,k} \mid j,k \in \{0,1\}\}$. Analogously, we have the set $X'_i = \{x_{i,j,k} \mid j,k \in \{0,1\}\}$, except for $i = 0$, when X'_i is not defined. Additionally, we have, for every positive i, a variable y_i. With these variables, we define $\vec{P}_n$ to be the quantifier prefix $\exists X_0 \exists X'_1 \forall y_1 \exists X_1 ... \exists X'_n \forall y_n \exists X_n$. Now, we define the following sets of clauses:*

- $B = \{(\neg x_{0,j,k}) \mid j,k \in \{0,1\}\} \cup \{(x_{n,j,0} \vee x_{n,j,1}) \mid j \in \{0,1\}\}$
- *For every $i \in [n]$ and every $j \in \{0,1\}$, $H_{i,j} = \{(\neg x'_{i,0,k} \vee \neg x'_{i,1,l} \vee x_{i-1,j,0} \vee x_{i-1,j,1}) \mid k,l \in \{0,1\}\}$*
- *For every $i \in [n]$, $T_i = \{(\neg x_{i,0,k} \vee y_i \vee x'_{i,0,k} \mid k \in \{0,1\}\} \cup \{(\neg x_{1,i,k} \vee \neg y_i \vee x'_{i,1,k}) \mid k \in \{0,1\}\}$*

For every positive natural number n, a QBF instance Φ is a Chen Formula of Type 1 *when $\Phi = \vec{P}_n : \phi$, where $\vec{P}_n$ is the prefix vector defined before and ϕ is the Boolean formula obtained from the conjunction of all the clauses in B, $H_{i,j}$ and T_i for every $i \in [n]$ and every $j \in \{0,1\}$.*

The fact that the sentences Φ_n are false can be understood following Chen's intuitive argument on his article. The interest of these formulae is in the essential property that the proof size needed to show their unsatisfiability varies drastically from one proof system to another. This is in shown in Proposition 6.1 and Theorem 6.6 in the original article. Here, we reproduce them in the following theorems:

Theorem 1 *The sentences $\{\Phi_n\}_{n\geq 1}$ have QU-resolution proofs of size linear in n.*

Theorem 2 *Treelike relaxing QU-res requires proof of size $\Omega(2^n)$ on the sentences $\{\Phi_n\}_{n\geq 1}$.*

3.2 Implementation

With the above context in mind, we easily understand the interest on being able to construct instances of these sentences and check whether available QBF solvers deal with them in linear time or in exponential time.

In order to do so, we have implemented a Python program that can generate instances of both Type 1 and Type 2 formulae (Type 2 is described later in this document). We now present this program.

3.2.1 Structure

The Python project consists of the following packages and modules:

- ***main.py***: the main script, where the menu is created and presented to the user and the options are displayed.
- ***tests.py***: testing functions for repeated tests, called from the main module.
- ***/tools***: a Python package with the module *system_tools*, used from some operations performed on the operating system.
- ***/instance_encodings***: two Python classes, *QBF* and *QBC*, each encapsulating formulas and circuits. A QBF instance consists of a quantifier prefix and a formula. As in classic SAT solvers, the formula is represented (written in Conjunctive Normal Form) as a list of lists, where each sublist

represents a clause and literals are represented using either positive or negative integers. Additionally, the QBF class has some methods for transforming a QBF instance stored in an object into a string encoded in the QDIMACS standard, which can then be printed onto a file. QBCs work the same way, except for the internal representation, inspired by the QCIR format.

- ***/generators***: this package contains two modules, one for each class of Chen Formulae. In this section we will be discussing the generator for Type 1 formulae.

3.2.2 The generator for Type 1

The generator for Type 1 is quite simple, as Definition 1 already provides a syntactic description of the formulas. Thus, the generator consists of four functions. The main one, `generate_ChenType1(n)`, receives a positive natural number n as input and is in charge of generating the formula. This is the method that can be called from outside.

In order to construct the formula, we create an object of the class `QBF` to store the formula and then call `generate_B_clauses(phi, n)`, `generate_H_clauses(phi, n)` and `generate_T_clauses(phi, n)` to add the corresponding clauses to the formula. Once it is ready, we return the complete formula object.

The main particularity of this generator is related to the way the formulas are represented. As mentioned above, we use objects that encapsulate CNF formulas. Following QDIMACS' tradition of using integers for variables, we encode clauses as lists of integers. However, Definition 1 refers to variables of the form $x_{i,j,k}$, $x'_{i,j,k}$ and y_i. It is clear that there is a need for a function that given a certain variable for one of those types can return a unique positive integer that represents it. In other words, we need a bijection between the set of variables and a subset of $\mathbb{N}$. Since it is straightforward to verify that an instance Φ_n has $4(n+1)+4n+n = 9n+4$ variables, we need a bijection from the set of variables V to the set $\{1,\ldots,9n+4\} \subset \mathbb{N}$. For this generator, we will use the following bijection, where jk_{10} is the number jk in base 10, since $j,k \in \{0,1\}$. The result of using this mapping can be seen on the table on the right:

$$f(v_{i,j,k}) = \begin{cases} jk_{10}+1 & \text{if } i = 0 \\ 9i+4 & \text{if } v = y \\ 9(i-1)+jk_{10}+5 & \text{if } v = x \\ 9(i-1)+jk_{10}+9 & \text{if } v = x' \end{cases}$$

In Python, the function receives a tuple that can be of the form `(i, None, 'y')` for the variables y_i and `(i, jk, 'p')` or `(i, jk, 'np')` for the rest of them.

Naturally, since f is a bijection there is an inverse function that converts natural numbers to variables encoded as tuples, but, for the time being, it has not been implemented, as it is of no use in this generator.

$x_{0,0,0}$	1
$x_{0,0,1}$	2
$x_{0,1,0}$	3
$x_{0,1,1}$	4
$x_{1,0,0}$	5
$x_{1,0,1}$	6
$x_{1,1,0}$	7
$x_{1,1,1}$	8
$x'_{1,0,0}$	9
$x'_{1,0,1}$	10
$x'_{1,1,0}$	11
$x'_{1,1,1}$	12
y_1	13
$x_{2,0,0}$	14
...	
$x'_{2,0,0}$	18
...	
y_2	22
⋮	

3.3 Experimental results

4 Chen Formulae of Type 2

4.1 Definition and basic properties

The Type 2 formulae, as defined by Chen in section 7 of his article, are the following:

Definition 2 *Let n be a positive natural number, and let $\vec{P}_n$ be the quantifier prefix $\exists x_1 \forall y_1 \ldots \exists x_n \forall y_n$. Now, we consider Boolean circuits $\phi_{n,j}$ such that $\phi_{n,j}$ is true if and only if $j + \sum_{i=1}^{n}(x_i + y_i) \not\equiv n \pmod 3$. A QBC $\Phi_n = \vec{P}_n : \phi_{n,0}$ is called a* Chen Formula of Type 2.

The basic properties of these circuits are the following:

Theorem 3 *For each $n \geq 1$, the sentence Φ_n is false.*

Theorem 4 *Relaxing QU-res requires proofs of size $\Omega(2^n)$ on the sentences $\{\Phi_n\}_{n\geq 1}$.*

4.2 Building the formulas: from CNF to circuit representation

In contrast with Definition 1, which provides a complete syntactic description of Type 1 formulae in CNF, Type 2 formulae are described in a much subtler way. The quantifier vector is precisely described ($\vec{P}_n = \exists x_1 \forall y_1 \ldots \exists x_n \forall y_n$), but the formula's matrix is defined by an arithmetic property. We will now discuss how to build these Boolean formulas in a way that does not lead to a combinatorial explosion, which will force us to abandon formulas in Conjunctive Normal Form and embrace ciruit representations.

We will first see the shortcomings of using CNF.

Let n be a positive natural number, and let us consider the Boolean formulas $\phi_{n,0}$, which we will simply refer to as ϕ. Theorem 3 states that the sentences $\{\Phi_n\}_{n\geq 1}$ are always false, but we should note that this refers to the whole QBC instances, not to the Boolean formulas in the matrix. Thus, ϕ is simply a Boolean formula over $2n$ variables, such that for all assignments $(x_1, y_1, \ldots, x_n, y_n)$ that make ϕ true, it must hold that

$$\sum_{i=1}^{n}(x_i + y_i) \not\equiv n \pmod 3$$

The question is, of course, what formulae verify this property?

Let $S = \sum_{i=1}^{n}(x_i + y_i)$. It is clear that, since there are $2n$ terms in the sum, which can take value either 0 or 1, the total value of S must be between 0 and $2n$. Some values in the set $\{0, \ldots, 2n\}$ of possible values for S will be congruent with $n \pmod 3$ while other will not. Actually, approximately one third of the values in the set of possible values for S will be congruent with $n \pmod 3$. The values in the set for which the congruence holds are what we call the *problematic values*.

Of course, we want to build a formula such that whenever an assignment leads to S being a problematic value, the formula is falsified. On the other hand, all assignments that lead to a normal, non-problematic value of S, must make the formula true.

Let us consider, for instance, the example for $n = 2$, where we have a formula $\phi(x_1, y_1, x_2, y_2)$. We have $S = (x_1 + y_1) + (x_2 + y_2) \in \{0, 1, 2, 3, 4\}$. There is a single problematic value in this case, $S = 2$, because $2 \equiv 2 \pmod 3$. This means that we must rule out all assignments with two 1's, while the rest of them must make the formula true. For each problematic assignment we can build a specific clause such that only that assignment makes it false. For $n = 2$, the problematic assignments and their corresponding clauses are the following:

$$(0,0,1,1) \to (x_1 \vee y_1 \vee \neg x_2 \vee \neg y_2)$$
$$(0,1,0,1) \to (x_1 \vee \neg y_1 \vee x_2 \vee \neg y_2)$$
$$(1,0,0,1) \to (\neg x_1 \vee y_1 \vee x_2 \vee \neg y_2)$$
$$(0,1,1,0) \to (x_1 \vee \neg y_1 \vee \neg x_2 \vee y_2)$$
$$(1,0,1,0) \to (\neg x_1 \vee y_1 \vee \neg x_2 \vee y_2)$$
$$(1,1,0,0) \to (\neg x_1 \vee \neg y_1 \vee x_2 \vee y_2)$$

The conjunction of those clauses is a Boolean formula written in Conjunctive Normal Form, verifying the desired modular property. In general, we can build Chen Formulas of Type 2 by looking for the problematic values in $\{0,\dots,2n\}$ and then adding a clause for each assignment that makes S add up to one of those values.

The reader may feel, however, that there are shortcomings to this representation of ϕ. The most urgent concern comes from the nagging question of *how many clauses will ϕ have when written this way?* The set of all possible assignments has cardinality $2^{2n} = 4^n$. Since, on average, one third of those assignments are related to problematic values of S, ϕ would end up having $\left\lceil \frac{4^n}{3} \right\rceil$ clauses. It is intuitive to see why this happens, though we can look for a more convincing proof in the fact that the exact number of clauses C is determined by the following sum, which turns out to be the mentioned $\left\lceil \frac{4^n}{3} \right\rceil$:

$$C = \sum_{i=0}^{\left\lfloor \frac{2n}{3} \right\rfloor} \binom{2n}{3i + (n \bmod 3)} = \left\lceil \frac{4^n}{3} \right\rceil$$

The previous equality can be easily proven using induction on n.

Having $\left\lceil \frac{4^n}{3} \right\rceil$ clauses provokes an exponential blowup when building the formulas. In addition, even if we managed to build some significant instances, QBF solvers would spend exponential time parsing the formulas, which would lead to useless experimental results.

Fortunately, Chen Formulas of Type 2 admit a succinct representation when written in the form of a circuit, where we do not force ϕ to be in CNF and we are allowed some additional operators, such as XOR gates, $\oplus$.

For every $k \in \{1,\dots,n\}$ and every $m \in \{0,1,2\}$, we will now consider the auxiliary circuits μ_m^k over variables $(x_1, y_1, \dots, x_k, y_k)$, which are defined so that they verify the following property:

$$\mu_m^k = 1 \Leftrightarrow \sum_{i=1}^{k} (x_i + y_i) \equiv m \pmod 3$$

For $k = 1$, the μ-circuits are:

$$\mu_0^1 = \neg x_1 \wedge \neg y_1$$
$$\mu_1^1 = x_1 \oplus y_1$$
$$\mu_2^1 = x_1 \wedge y_1$$

If we have circuits μ_m^k for every k up to $n-1$, we can easily obtain the ones for $k = n$:

$$\mu_0^n = (\mu_0^{n-1} \wedge \neg x_n \wedge \neg y_n) \vee (\mu_1^{n-1} \wedge x_n \wedge y_n) \vee (\mu_2^{n-1} \wedge (x_n \oplus y_n))$$

$$\mu_1^n = (\mu_0^{n-1} \wedge (x_n \oplus y_n)) \vee (\mu_1^{n-1} \wedge \neg x_n \wedge \neg y_n) \vee (\mu_2^{n-1} \wedge x_n \wedge y_n)$$

$$\mu_2^n = (\mu_0^{n-1} \wedge x_n \wedge y_n) \vee (\mu_1^{n-1} \wedge (x_n \oplus y_n)) \vee (\mu_2^{n-1} \wedge \neg x_n \wedge \neg y_n)$$

Now, we can easily express our formula ϕ as

$$\phi = \neg \mu_{n \bmod 3}^n$$

Since the size of every μ-circuit is constant and we have three μ-circuits for every $k \in \{1, \ldots, n\}$, we have $3n$ auxiliary circuits plus the final one for ϕ, which means we can build Chen Formulas of Type 2 in size $\Theta(n)$ when building them as circuits.

One could argue that using Tseitin variables we could easily transfrom our circuits to usual CNF without causing an exponential blowup. Indeed, we have considered this and we show the results of solving both the original circuits and the CNFied versions in the following sections.

4.3 The generator for Type 2

The program generating the instances for Type 2 circuits is very simple. There is, as in the first generator, a function `generate_ChenType2(n)` receiving a natural number $n \geq 1$; this is the function that should be called from outside.

Inside this method we call `generate_quantifier_blocks(phi, n)` and `generate_gates(phi, n)`, which complete the formula.

The main remark is that this time the formula is encoded as a circuit, in an object of the class `QBC` instead of `QBF`. The internal representation of these objects is inspired by the QCIR format and, in fact, the way in which they are written.

We generate some auxiliary gates in addition to the already known μ-circuits due to the somewhat cumbersome syntax of the QCIR format. Overall, the actual representation contains the $2n$ quantified variables and $15n - 11$ gates.

The following is an example formula for the case $n = 2$, written in the QCIR format:

```
#QCIR-14
# Circuit name: type2_size2
# Num. vars.: 4
# Num. gates: 19
exists(x1)
forall(y1)
exists(x2)
forall(y2)
output(-s_2_2)
s_1_0 = and(-x1, -y1)
s_1_1 = xor(x1, y1)
s_1_2 = and(x1, y1)
adder_2_0 = and(-x2, -y2)
adder_2_1 = xor(x2, y2)
adder_2_2 = and(x2, y2)
aux_2_0_0 = and(s_1_0, adder_2_0)
```

```
aux_2_0_1 = and(s_1_1, adder_2_2)
aux_2_0_2 = and(s_1_2, adder_2_1)
s_2_0 = or(aux_2_0_0, aux_2_0_1, aux_2_0_2)
aux_2_1_0 = and(s_1_0, adder_2_1)
aux_2_1_1 = and(s_1_1, adder_2_0)
aux_2_1_2 = and(s_1_2, adder_2_2)
s_2_1 = or(aux_2_1_0, aux_2_1_1, aux_2_1_2)
aux_2_2_0 = and(s_1_0, adder_2_2)
aux_2_2_1 = and(s_1_1, adder_2_1)
aux_2_2_2 = and(s_1_2, adder_2_0)
s_2_2 = or(aux_2_2_0, aux_2_2_1, aux_2_2_2)
```

4.4 Experimental results

References

[1] Hubie Chen. Proof Complexity Modulo the Polynomial Hierarchy: Undestanding Alternation as a Source of Hardness. *ACM Transactions on Computation Theory*, 9(3), 2017.