A QBF Instance Generator for Chen Formulae

Noel Arteche Echeverría

Supervisor:
Montserrat Hermo

eman ta zabal zazu

Universidad
del País Vasco

Euskal Herriko
Unibertsitatea

FACULTY
OF COMPUTER
SCIENCE
UNIVERSITY
OF THE BASQUE
COUNTRY

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 2016 article *Proof Complexity Modulo the Polynomial Hierarchy: Understanding Alternation as a Source of Hardness*. We present a computer program for generating instances of these classes in popular representation formats (QDIMACS and QCIR).

Contents

1 Introduction

In his 2016 article (see [3]), Hubie Chen asked himself about how proof systems like QU-resolution were handling quantifier alternation in their proofs of unsatisfiability of QBF formulas.

He presented the notion of a *proof system ensemble*, an infinite collection of proof systems, each of them related to one level of the Polynomial Hierarchy. His new proof system, called *Relaxing QU-res*, was then compared against QU-resolution, to show how good it compared in size of proofs. Since Relaxing QU-res is closely related to the Polynomial Hierarchy, Chen's proof system showed that alternation of quantifiers is indeed a source of hardness for proof systems.

In order to show this, he presented two classes of formulas. The first class, now named *Chen Formulas of Type 1*, was used to show how QU-resolution can handle alternation, especially compared to a system like Relaxing QU-res, where alternation of quantifiers presents serious difficulties . Briefly, he showed that there are linear size proofs for these formulas in QU-resolution, while Treelike Relaxing QU-res needs exponential size to solve them.

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, which are also used to prove an exponential lower bound on his newly defined proof system ensemble, Relaxing QU-res. The main interest of these formulas is that they are defined as circuits, instead of in clausal form.

Until now, however, nobody had investigated how these formulas behave on real QBF solvers. The interest in doing so is related to two important questions. Firstly, can real solvers find the linear size proofs of Type 1 formulas? Secondly, is there a real difference between building formulas as circuits and CNF for Type 2 instances?

1.1 Our results

We define both classes of formulae, we review the basic properties shown in Chen's article and propose two Python programs that can generate instances of any size. We then show the running times of popular QBF solvers, and discuss whether they give relevant information to answer the two questions mentioned before.

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 ([3]), 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 $\{1,\dots,k\}$.

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*, $\vec{P} = Q_1 v_1 \dots Q_n v_n$, such that $v_1,\dots,v_n$ are propositional variables and $Q_1,\dots,Q_n$ are quantifiers in $\{\exists,\forall\}$; and ϕ is a Boolean circuit built from constants 0 and 1, propositional variables $v_1,\dots,v_n$ 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).

By a *partial assignment* to a QBF or QBC Φ we refer to a function a from a set of variables $S \subseteq V$ (where V is the set of all variables in Φ) to $\{0,1\}$. If $a : S \subseteq V \to \{0,1\}$ is a partial assignment, then $\vec{P}[a]$ is the quantifier prefix from Φ but with all the variables in S instantiated (and therefore removed). Analogously, we define $\phi[a]$ to be the circuit obtained when replacing each variable in S by the value given to it by a. Thus, $\Phi[a] = \vec{P}[a] : \phi[a]$.

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* (see [2] and [1]). The Q-resolution proof system is an extension of the famous *Resolution* proof system used with propositional formulas (see [8]), 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 [11]). 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 ([11]).

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. Thus, 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 parameterization.

More precisely, a *proof system ensemble* (see Definition 3.1 in [3]) 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) (see [6] or [9] for a formal introduction).

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, we now define the notion of *axiom sets*. Let Φ be a QBF instance. For each $k \geq 2$ we define the set $H(\Phi, \Pi_k)$ to contain the axioms than can be derived from QBFs $\Phi[a]$ having a Π_k relaxation (relaxations with a Π_k prefix, where Π_k is the regular expression $\forall^*\exists^* \dots \forall^*\exists^*$ 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

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

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 [3]). Whenever $G(\pi)$ is a tree we say that the proof π is *treelike*.

2.3 Overview of the original results

In his article, Chen defines two sets of QBF instances. The first set, what we now call *Chen Formulas of Type 1*, is used to show an exponential separation between the treelike and the general version of relaxing QU-res. This is an alternation-based analog of the known separation between treelike and general Resolution. Since treelike relaxing QU-res can be viewed as the trace 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 $\exists X_0 \exists X'_1 \forall y_1 \exists X_1 \ldots \exists X'_n \forall y_n \exists X_n$. 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 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 The generator for Type 1

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.

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(...)`, `generate_H_clauses(...)` and `generate_T_clauses(...)` 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 Table 1.

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

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.

The generated formulas will have $9n+4$ variables and $12n+6$ clauses.

Appendix B contains the relevant information regarding where to download the Python script and how to run it.

$x_{0,0,0}$	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
$\dots$	
$x'_{2,0,0}$	18
$\dots$	
y_2	22
$\vdots$	

Table 1: Result of the bijection for renaming variables of Type 1 formulas

3.3 Experimental results

Using the generator described above, we have been able to construct in QDIMACS formulas from $n = 1$ to $n = 5000$. Some of them are quite big instances and not all of them have been tested.

Our aim was to show how current state-of-the-art solvers perform on these formulas. In order to do so, we have selected the top 3 prenex-CNF solvers from the 2018 QBFEVAL competition (see [7] for complete results). These are the *CAQE* solver, the *depQBF* solver and *QBF Portfolio*. At the time of writing these pages, the QBF Portfolio solver was nowhere to be found: the QBFEVAL site did not offer enough reference about where to find it and the team behind the event did not answer to the question of where it could be obtainted. Thus, the tests we present now have been performed on the first two solvers: CAQE and depQBF. These have been conducted on an system with an Intel Core i5-8250U processor with 8 GB of RAM. This is a rather limited machine for performing large scale tests and thus the experimental results gathered are a bit limited in some cases. More powerful computational resources would lead to more clarifying data.

Firstly, in Figure 1 we show the running times for the generator *per se*, from $n = 1$ to $n = 5000$. As expected, the running time is linear, as the genertor preserves the definition of the formulas, which defines an amount of variables and clauses linear in n.

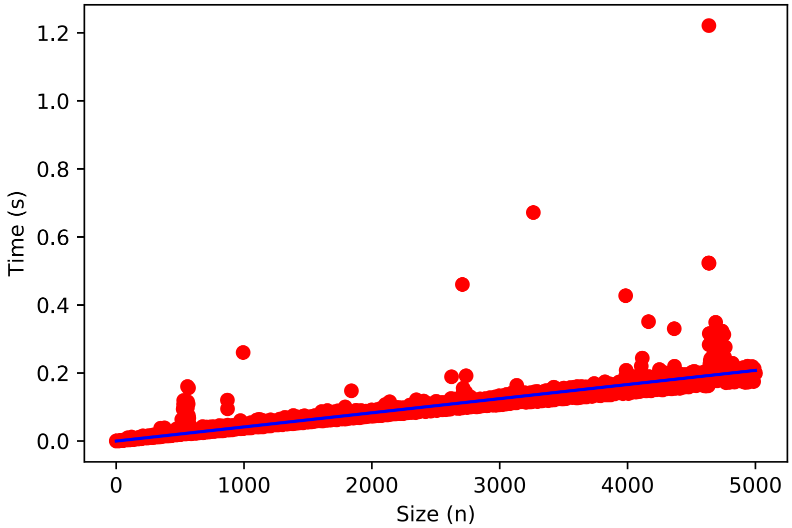

Figure 1: Running times for generating Type 1 formulas from $n = 1$ to $n = 5000$ in QDIMACS.

The linear regression performed on the data gives a slope of the order of 10^{-5} and, except for the outliers, we can build very big formulas in a very reasonable amount of time on a normal machine (less than a quarter of a second for $n = 5000$). We shall remember that the instance for size $n = 5000$ has 45004 variables and 60006 clauses.

The fundamental issue, however, is how long does it take for a QBF solver to find that these formulas are unsatisfiable? As explained above, we have performed tests on CAQE and depQBF. On each of them we have solved the instances for sizes $n = 1, 100, 200, 300, \ldots, 1000$. We have repeated each test three times and obtained the average running time for each instance size.

The two question we would like to answer are:

n	CAQE	depQBF
1	0.08226	0.04781
100	0.71697	0.06499
200	4.81839	0.16171
300	16.41930	0.51730
400	38.15976	1.17025
500	80.94273	1.67872
600	145.97795	2.56036
700	205.64134	3.78108
800	306.46030	5.34902
900	440.89564	6.09534
1000	582.56338	7.82560

Table 2: Type 1 formulas' running times in seconds on CAQE and depQBF

1. Do solvers achieve linear running times on Type 1 formulas, based on the fact that there exist linear size proofs for them in QU-resolution?
2. And, provided this is not the case, how fast can they solve them? Does the time grow polynomially or exponentially?

First and foremost, we present the running times in seconds for both solvers in Table 2, where we can already see that depQBF performs significantly faster.

In Figure 2 we show the running times compared, where we can clearly see how debQBF outperforms CAQE.

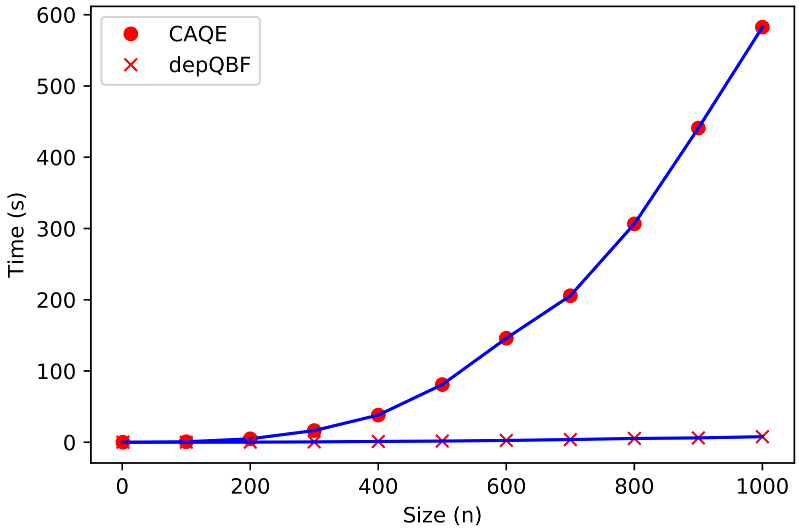

Figure 2: Type 2 formulas' running times on CAQE and depQBF

3.3.1 Results on depQBF

Figure 2, unfortunately, shows very little of how depQBF is actually doing. It could seem like the running time is linear, though let's look closer at running times on depQBF. We have obtained interpolation polynomials of degree 1, 2 and 3 for the data up to $n = 600$ and then extended these polynomials to see how they fitted the complete data set (we do not show the regressions done with fourth degree polynomials and exponential functions, as they are a clear wrong choice here).

In Figure 3 we can see the results for the mentioned regressions and we can see how the quadratic curve fits almost perfectly.

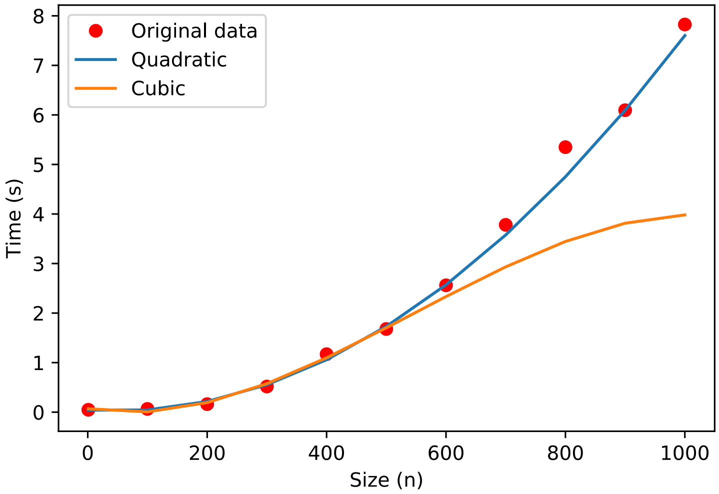

Figure 3: Quadratic and cubic regressions up to $n = 600$ against the original T1 data in depQBF

In fact, given how fast depQBF is, we have been able to perform tests up to $n = 5000$ and the quadratic regression fits tightly with the data. Compared to it, higher degree approximations either do not fit or have too small director coefficients, implying they are actually trying to approximate a polynomial of lower degree. Moreover, if we obtain a quadratic polynomial with the data up to $n = 2000$, the curve is almost perfect, as we can appreciate in Figure 4.

Although the quadratic approximation fits almost perfectly, we should note that this is still a very slow growing function (the obtained approximation is roughly $f(n) = 1.22 \cdot 10^{-5}n^2 - 6.45 \cdot 10^{-3}n + 1.34$). Thus, we could reasonably argue that the depQBF solver is definitely finding the proofs very fast, even if there is noise or implementation details that lead to strictly non-linear running times.

3.3.2 Results on CAQE

The results we can show for CAQE are rather limited. As we saw in Figure 3, running times grow significantly faster than in depQBF. We have again done polynomial regression on the data up to $n = 600$ and then extended the obtained functions to see how they fit the rest of the data. Linear functions are not displayed, as it is clear that they are not the case, and we omit exponential functions and polynomials of degree higher than three, as we have checked that these are either difficult to fit or have very small coefficients.

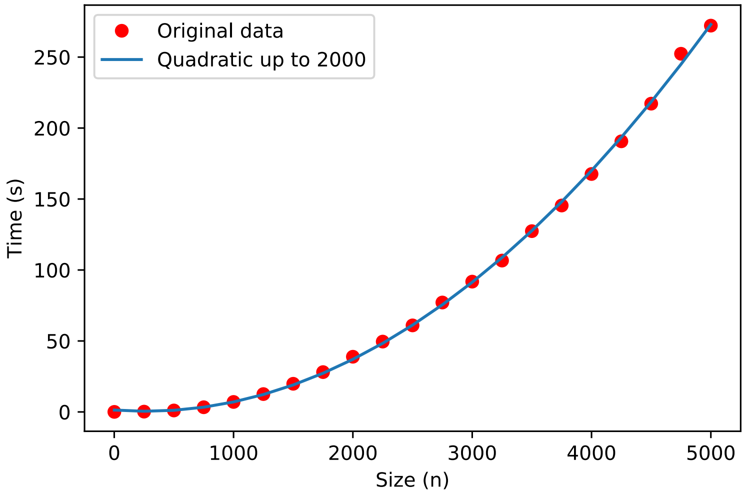

Figure 4: Quadratic regression up to $n = 2000$ against the original T1 data in depQBF

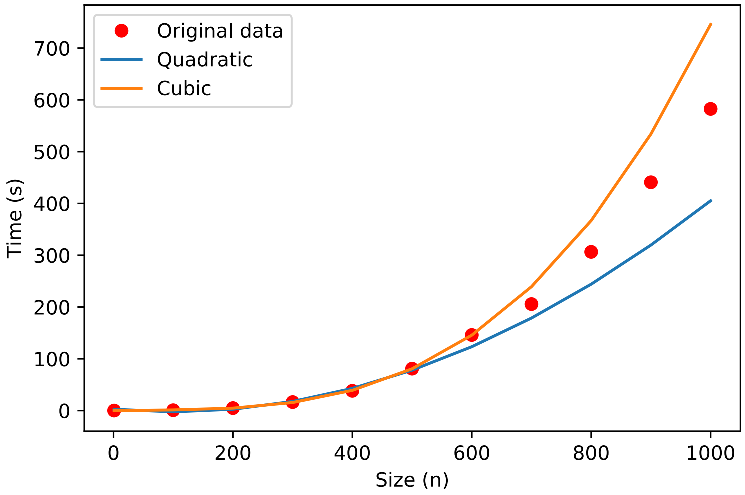

Figure 5: Quadratic and cubic regressions up to $n = 600$ against the original T1 data in CAQE.

In Figure 5 we can appreciate the comparison between actual running times and a quadratic and a cubic approximation, among which the actual running time is bounded. Given the slow times for large instances, it was not possible to obtain more data in this case, and this it is therefore difficult to confirm if the running times for Type 1 formulas on CAQE are $\Theta(n^2)$ or $\Theta(n^3)$. However, we can assure that they are not exponential.

3.3.3 Final comments on Type 1 experimental results

We should remind the reader that, in general, the task of extracting the asymptotic growth of an algorithm given its running times is a complex one. Firstly, because we tend to have little to no control over the noise introduced by the operating system, the language's internal compilation or interpretation or the machine in which we are running the code. This noise can provoke the data to not fit in a precise class of functions, even when the algorithm beneath does belong there. Secondly, the data we have in these cases is, as mentioned above, quite limited, and it is difficult to discern if the devised patterns extend to higher values of n.

With all this precaution in mind, though, we can be fairly sure in the fact that nor CAQE nor depQBF face an exponential blow-up when trying to solve Type 1 Chen formulas. In fact, depQBF seems to be finding short proofs very fast, while CAQE lies somewhere in between $\Omega(n^2)$ and $O(n^3)$.

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 $\exists x_1 \forall y_1 \dots \exists x_n \forall y_n$. 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 ($\vec{P_n} = \exists x_1 \forall y_1 \dots \exists x_n \forall y_n$), 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 circuit 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 $(x_1, y_1, \dots, x_n, y_n)$ 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 $\{0, \dots, 2n\}$ of possible values for S will be congruent with $n \pmod 3$ while others 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, $S = 2$. 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:

$$\begin{gathered}
(0,0,1,1) \rightarrow (x_1 \vee y_1 \vee \neg x_2 \vee \neg y_2) \\
(0,1,0,1) \rightarrow (x_1 \vee \neg y_1 \vee x_2 \vee \neg y_2) \\
(1,0,0,1) \rightarrow (\neg x_1 \vee y_1 \vee x_2 \vee \neg y_2) \\
(0,1,1,0) \rightarrow (x_1 \vee \neg y_1 \vee \neg x_2 \vee y_2) \\
(1,0,1,0) \rightarrow (\neg x_1 \vee y_1 \vee \neg x_2 \vee y_2) \\
(1,1,0,0) \rightarrow (\neg x_1 \vee \neg y_1 \vee x_2 \vee y_2)
\end{gathered}$$

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 $\{0,\ldots,2n\}$ 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 $\lceil \frac{4^n}{3} \rceil$ 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 $\lceil \frac{4^n}{3} \rceil$:

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

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

Having $\lceil \frac{4^n}{3} \rceil$ clauses provokes an exponential blow-up 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 $k \in \{1,\ldots,n\}$ and every $m \in \{0,1,2\}$, we will now consider the auxiliary circuits μ_m^k over variables $(x_1, y_1, \ldots, x_k, y_k)$, 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 $k = n$:

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

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

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

Now, we can easily express our formula ϕ as

$$\phi = \neg \mu^n_{n \bmod 3}$$

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.

Nevertheless, we can still obtain these formulas in CNF avoiding the large number of clauses. Once we have built the circuit for a given n, we can use Tseitin variables (see [10]) and transform the circuit to a formula in clausal form. In order to do so we just add as many auxiliary variables as gates in our circuits. Since each μ-circuit has 4 gates inside of it and we have $3n$ μ-circuits, there will be $12n$ additional variables. Added to the original $2n$ variables we would end up with a CNF formula over $14n$ variables; still linear. Regarding the clauses, we would have one clause for the output gate and $12n$ clauses that would establish the equivalence between the new variable and their original gate. Since this equivalence can always be translated to a constant number of clauses, the final formula would be of size proportional to $3n$ and thus, linear.

In the generator presented in the next section we can output instances as both circuits and CNF formulas.

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(...)` and `generate_gates(...)`, 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 (introduced in [5]). In fact, when generated as circuits, the formulas are returned in this format.

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 $17n - 11$ gates.

As we mentioned earlier, apart from the QCIR format for circuits, we can also output Type 2 formulas in QDIMACS to test them on traditional solvers. These have $19n - 12$ variables and $54n - 38$ clauses.

Appendix B contains the relevant information regarding where to download the Python script and how to run it.

4.4 Experimental results

Using the generator described above, we have been able to construct Type 2 formulas for sizes from $n = 1$ up to $n = 5000$. We have built them both in QCIR (as circuits) and in QDIMACS (in CNF). Naturally, though, these are quite big instances, and not all of them have been tested.

In this case, we were interested in comparing the running times between the circuit-based solvers and the classic CNF solvers, as it is generally thought that CNF is not the ideal way to encode formulas for the QBF problem.

Regarding non-prenex solvers we have selected the top 3 from the 2018 QBFEVAL competition (see [7] for complete results). These are the *QuAbS* solver, the *CQESTO* solver and *GhostQ*. The GhostQ solver seems to use an internal format for circuits different from QCIR, and the converter provided by them was incorrect. This means we could not solve our QCIR formulas on their solver. Thus, the tests we present now have been performed on the first two solvers: QuAbS and CQESTO. The tests on CNF-solvers were done on the same two solvers used for Type 1 formulas: CAQE and depQBF.

The system used was again an Intel Core i5-8250U processor with 8 GB of RAM. This is a rather limited machine for performing large scale tests, and thus the experimental results gathered are a bit limited in some cases. More powerful computational resources would lead to more clarifying data.

Firstly, in Figure 6 we show the running times for the generator *per se*, from $n = 1$ to $n = 5000$. As expected, the running time is linear, as the generator preserves the construction presented in section 4.2.

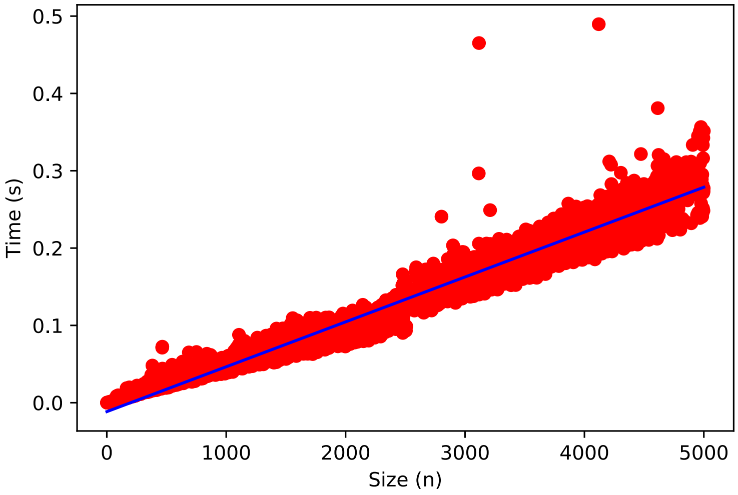

Figure 6: Running times for generating Type 2 formulas from $n = 1$ to $n = 5000$ in QCIR.

This time, the two questions we would like to answer are:

1. How long does it take for a solver to find the unsatisfiability of Type 2 formulas? That is, how hard are they for them?
2. Is there a significant difference in running time between solvers designed for circuits and solvers designed for CNF?

We answer the first question for circuit-based solvers on the next section and postpone the second one until section 4.4.2, where we will find some interesting results.

4.4.1 Results for circuit-based solvers

We have performed tests on QuAbS and CQESTO and solved the instances for sizes $n = 1, 100, 200, 300, \ldots, 600$ (for QuAbS we have data up to 1000, but CQESTO was significatly slower and we had to stop at 600).

n	**QuAbS**	**CQESTO**
1	0.05061	0.12992
100	1.39349	3.48476
200	6.45269	23.03770
300	18.52676	137.92092
400	40.95977	362.74532
500	76.81426	1564.41455
600	131.73557	4153.90999
700	206.60374	-
800	316.54247	-
900	480.91833	-
1000	619.57376	-

Table 3: Type 2 ciruits' running times in seconds on QuAbS and CQESTO

We have repeated each test three times and obtained the average running time for each instance size. The running times are shown in Figure 7 as well as in Table 3.

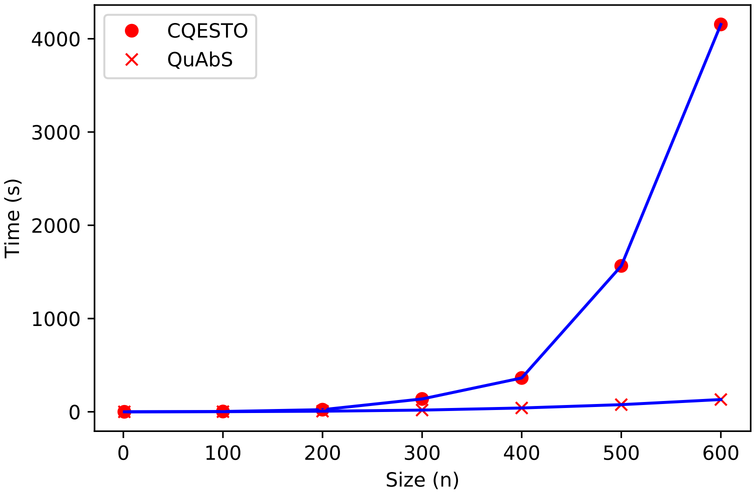

Figure 7: Type 2 circuits' running times on QuAbS and CQESTO

This time, QuAbS outperforms CQESTO by far.

Results from QuAbS present a pattern similar to the one we have seen on CAQE with Type 1 formulas. Exponential regression does not fit (it does, but with a base that tends to 1), and thus in Figure 8 we present polynomial regressions for degrees 1 to 4 done with values from $n = 1$ to $n = 600$ and then extended up to 1000 to see how they fit.

The linear and quadratic functions are easily discarded. The cubic and the fourth degree ones are both close, but when we compare the latter's director coefficient we find that it is too small (10^{-7} versus 10^{-5}). Thus, we could conclude that the running time on QuAbS seems to grow at a cubic rate.

Results on CQESTO are much more difficult to read due to the fact that we have too few data. Figure

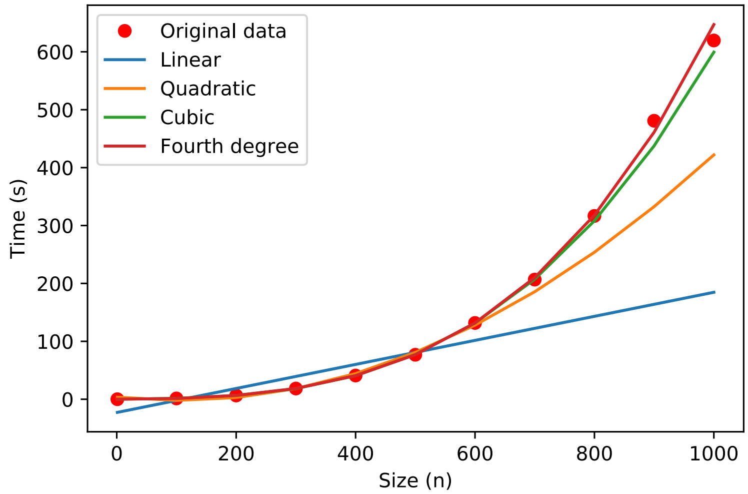

Figure 8: Polynomial regressions up to $n = 600$ against T2 original data on QuAbS.

9 shows all the possible approximations. These have been obtained using the whole data set minus the last value, so we cannot see what happens for higher values of n. The exponential could be discarded on the grounds that its base is very small to be real ($f(n) = 1.015^n + 12.788$). Between the cubic and the fourth degree approximation we are still in much the same situation as before, but nothing definitive can be concluded.

4.4.2 Results on CNF-based solvers

As we have explained in section 4.2, using Tseitin variables we can build Type 2 formulas in size linear in n. Figure 10 shows the running times for the generator outputting the QDIMACS formulas up to $n = 5000$. It is obviously linear.

These format leads to a surprising result, that can easily be appreciated in Table 4: the running times follow a clearly exponential pattern. So much so, that we have been unable to collect date above $n = 15$.

As we see, it gets out of hand very quickly. When plotting the data, we see how an exponential curve feats neatly (see Figure 11). For CAQE this curve is approximately $f(n) = 1.8538^n - 259.0903$, while for depQBF it goes up a bit higher up to $f(n) = 1.9016^n - 393.3636$

4.4.3 Final comments on Type 2 experimental results

We should remind the reader that, in general, the task of extracting the asymptotic growth of an algorithm given its running times is a complex one. This is particularly clear in the case of CQESTO, where there was not much to be said.

The general conclusion of Type 2 formulas is that, as expected, encoding them as circuits results in a more efficient solving. When written in QDIMACS, even if the size of the formulas grows linearly the running times grow exponentially, thus showing the generally accepted notion that CNF is not the ideal format for QBF solvers.

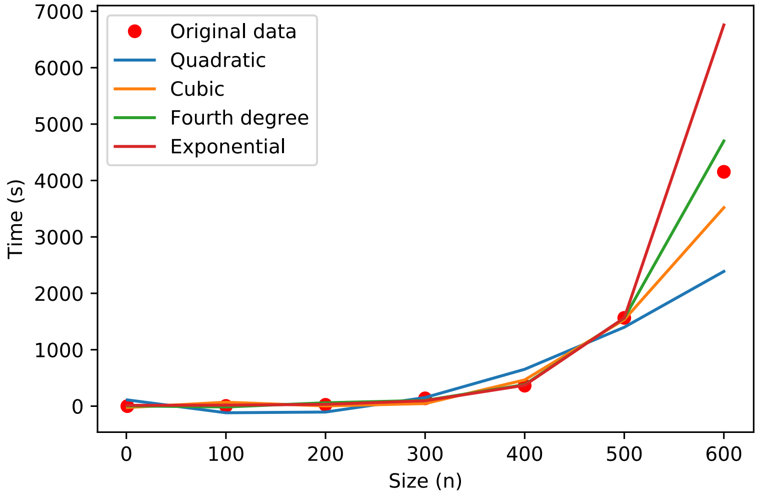

Figure 9: Polynomial and exponential regressions on all the original T2 data on CQESTO minus the last point.

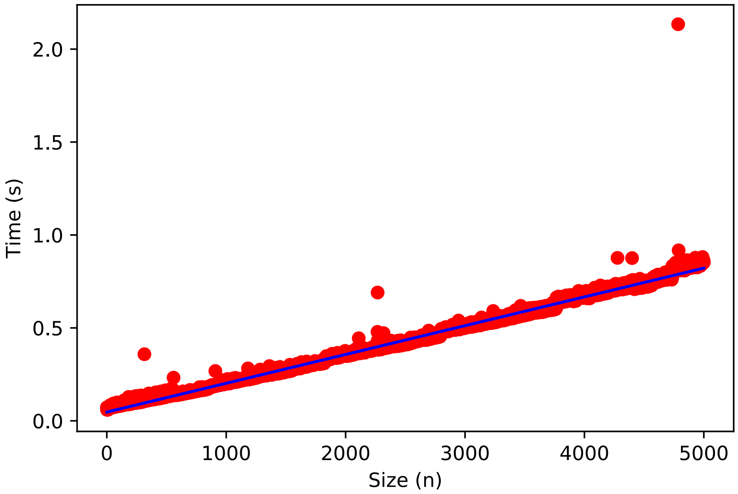

Figure 10: Running times for generating Type 2 formulas from $n = 1$ to $n = 5000$ in QDIMACS.

n	**CAQE**	**depQBF**
1	0.10295	0.03602
2	0.04303	0.02145
3	0.04304	0.02372
4	0.04890	0.02330
5	0.05341	0.02648
6	0.08515	0.03009
7	0.22742	0.04562
8	0.62194	0.12410
9	1.87319	0.48458
10	7.42670	2.09502
11	31.11183	9.75327
12	108.41917	48.84733
13	355.90740	266.93340
14	1541.80189	1539.74230
15	6360.79237	9288.23431

Table 4: Type 2 CNF formulas' running times in seconds on CAQE and depQBF

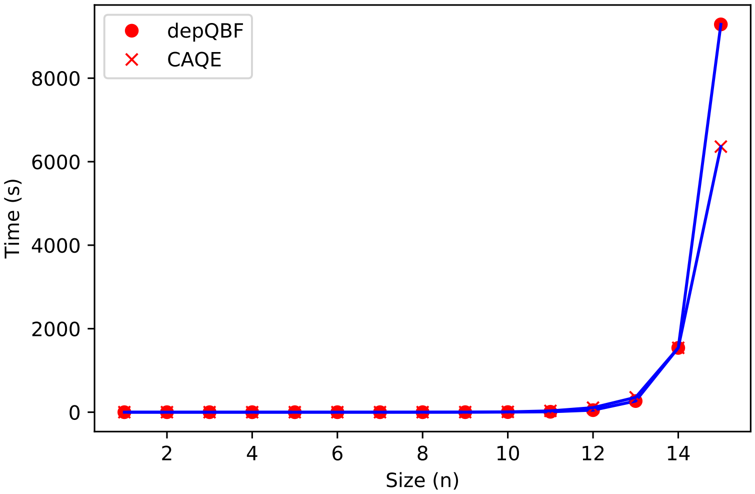

Figure 11: Type 2 formulas' running times on CAQE and depQBF.

5 Conclusion and future work

The results presented so far can be briefly summarized as follows.

Regarding Type 1 formulas, we have seen how solvers can handle them with certain ease, particularly depQBF, which, even if seems like it is not finding the liner size proofs immediately, can solve the formulas very fast. CAQE is substantially slower, though its growth rate is not exaggerated. In fact, it may have the same behaviour as depQBF but with bigger hidden constants.

Regarding Type 2 formulas, we have seen how, as expected by the community, circuit representation is clearly beneficial for QBF solvers. This was accentuated by the surprising result of exponential running times on formulas converted to linear size CNF. There is, however, room for improvement in the efficiency of circuit-based solvers.

Regarding our own work, we have presented two working generators available online that can be used for future research in this area.

Looking into the future, we would like to continue our research focusing on the implementation of the solvers discussed. So far, we have treated them as black boxes, which has made difficult the task of determining asymptotic complexities. A more through research of available algorithms for QBF solvers could lead to interesting results, regarding how the actual proofs are obtained, as well as how other sets of hard formulas relate to ours.

Acknowledgements

Thanks to Montserrat Hermo in her continuous supervision of the worked done so far.

Thanks to Hubie Chen for proposing the project as well as for his helpful feedback and comments during its development.

Thanks to Edu Vallejo, whose weekly comments helped in the experimental side of the project regarding runnning times, regressions and approximations.

Special thanks to Robert Israel, Emeritus Associate Professor of Mathematics at University of British Columbia, who willfully helped with the idea of μ-circuits, as well as to Mikoláš Janota, for his technical support with the CQESTO solver.

References

[1] Valeriy Balabanov, Magdalena Widl, and Jie-Hong R. Jiang. QBF Resolution Systems and Their Proof Complexities. In Carsten Sinz and Uwe Egly, editors, *Theory and Applications of Satisfiability Testing – SAT 2014*, pages 154–169, Cham, 2014. Springer International Publishing.

[2] O Beyersdorff, L Chew, and Mikolas Janota. Proof Complexity of Resolution-based QBF Calculi. *Leibniz International Proceedings in Informatics, LIPIcs*, 30:76–89, 02 2015.

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

[4] H. Chen and Y. Interian. A Model for Generating Random Quantified Boolean formulas. *Proceedings of the Nineteenth International Joint Conference on Artificial Intelligence*, 2005.

[5] C. Jordan, W. Klieber, and M. Seidl. Non-CNF QBF Solving with QCIR. *AAAI Workshop: Beyond NP*, 2016.

[6] A. R. Meyer and L. J. Stockmeyer. The equivalence problem for regular expressions with squaring requires exponential space. pages 125–129, 1972.

[7] Luca Pulina and Martina Seidl. Results of the 11th QBF Solvers Evaluation (QBFEVAL 18). *21st International Conference on Theory and Applications of Satisfiability Testing*.

[8] J. A. Robinson. A Machine-Oriented Logic Based on the Resolution Principle. *J. ACM*, 12(1):23–41, January 1965.

[9] Larry J. Stockmeyer. The Polynomial-Time Hierarchy. *Theoretical Computer Science*, 3(1):1 – 22, 1976.

[10] G. S. Tseitin. On the Complexity of Derivation in Propositional Calculus. 1983.

[11] Allen Van Gelder. Contributions to the Theory of Practical Quantified Bolean Formula Solving. In Michela Milano, editor, *Principles and Practice of Constraint Programming*, pages 647–663, Berlin, Heidelberg, 2012. Springer Berlin Heidelberg.

Appendix A - Repository with available formulas

During the development of the project and for testing purposes we generated all instances for both Type 1 and Type 2 formulas from $n = 1$ up to $n = 5000$. The Type 1 formulas were written in QDIMACS, while Type 2 formulas were built in both QCIR and QDIMACS.

The files containing them are available for download on the GitHub repository of this project. The folder containing them can be found at:

`https://github.com/alephnoell/ChenGenerator/tree/master/formulas`

If somebody wants to build bigger instances, the generator itself can be downloaded and run. This is explained in Appendix B.

Appendix B - How to download and run the generator

The GitHub repository for this project contains the developed generators as two Python scripts that can be run to generate the formulas.

These can be found here:

`https://github.com/alephnoell/ChenGenerator/tree/master/generators`

To run the Type 1 generator, one must run the `generate_T1.py` script. The command is the following:

```
python3 generate_T1.py n [filename]
```

The first argument is the value of n. The second argument is the name of a file. If not given, it will print the generated formula on the standard output, though it will not print formulas with more than 100 variables or clauses.

To run the Type 2 generator, one must run the `generate_T2.py` script. The command is one of the following:

```
python3 generate_T2.py n -QCIR [filename]
python3 generate_T2.py n -QDIMACS [filename]
```

The first argument is the value of n. The second argument is either `-QCIR` or `-QDIMACS`, and it specifies the format in which the formula has to be written. The third argument is the name of a file. If not given, it will print the generated formula on the standard output, though it will not print formulas with more than 100 variables or clauses.