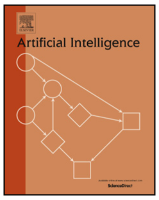
Artificial Intelligence 234 (2016) 1–25

Contents lists available at ScienceDirect

Artificial Intelligence

www.elsevier.com/locate/artint



Solving QBF with counterexample guided refinement

Mikoláš Janota [a,*], William Klieber [b], Joao Marques-Silva [a,c], Edmund Clarke [b]

[a] IST/INESC-ID, Technical University of Lisbon, Portugal
[b] Carnegie Mellon University, Pittsburgh, PA, USA
[c] CASL, University College Dublin, Ireland

A R T I C L E I N F O

Article history:
Received 12 July 2014
Received in revised form 2 January 2016
Accepted 8 January 2016
Available online 13 January 2016

Keywords:
QBF
Expansion
Counterexample guided abstraction refinement (CEGAR)

A B S T R A C T

This article puts forward the application of Counterexample Guided Abstraction Refinement (CEGAR) in solving the well-known PSPACE-complete problem of quantified Boolean formulas (QBF). The article studies the application of CEGAR in two scenarios. In the first scenario, CEGAR is used to expand quantifiers of the formula and subsequently a satisfiability (SAT) solver is applied. First it is shown how to do that for two levels of quantification and then it is generalized for arbitrary number of levels by recursion. It is also shown that these ideas can be generalized to non-prenex and non-CNF QBF solvers. In the second scenario, CEGAR is employed as an additional learning technique in an existing DPLL-based QBF solver. Experimental evaluation of the implemented prototypes shows that the CEGAR-driven solver outperforms existing solvers on a number of benchmark families and that the DPLL solver benefits from the additional type of learning.

1. Introduction

SAT solving has marked a considerable practical success by enabling efficient solving of large industrial NP-hard problems. This motivates the study of computationally harder problems. Quantified Boolean formulas (QBFs) [1] naturally extend the SAT problem by enabling quantification. The cost of this extension is that deciding QBF formulas is PSPACE-complete. However, this also means that a larger set of problems can be targeted [2–5].

While nonrandom SAT solving has been dominated by the DPLL procedure, it has proven to be far from being a silver bullet for QBF solving. Indeed, a number of solving techniques have been proposed for QBF [6–10], complemented by a variety of *preprocessing techniques* [11–16].

Currently, QBF solving can be divided into *search-based* and *expansion-based*. Search-based solvers apply conflict and solution-driven search throughout the formula's assignments [17]. In contrast, expansion-based solvers transform the formula into a propositional one by gradually rewriting quantifiers into the corresponding Boolean connectives [18,8,19,9]. This, however, may lead to exponential blowup in the size of the formula. This article provides a technique that mitigates this issue. Instead of always expanding quantifiers, they are expanded carefully, on demand. For such, we apply the well-known paradigm of counterexample guided abstraction refinement (CEGAR) [20].

The article shows that CEGAR can be applied in two significantly different ways. The first approach gradually *expands* the given formula into a propositional one. Once the formula is propositional, a SAT solver is applied in a blackbox-fashion. In the spirit of CEGAR, the algorithm partially expands the formula and tests whether such expansion is sufficient. If it is not,

[*] Corresponding author.
E-mail address: mikolas@sat.inesc-id.pt (M. Janota).

http://dx.doi.org/10.1016/j.artint.2016.01.004
0004-3702/© 2016 Elsevier B.V. All rights reserved.

the algorithm continues. The second approach employs CEGAR as an *additional learning technique* in an existing DPLL-based QBF solver. At the price of higher memory consumption, this learning technique enables more aggressive pruning of the search space than the existing techniques [17,21]. The experimental evaluation carried out demonstrates that CEGAR-based techniques are useful for a large number of benchmark families.

This article builds on a previous paper on 2-level QBF solving [22] and solving of QBF of arbitrary levels [23]. This article extends this work by showing that CEGAR-driven expansion can be also applied to QBF formulas that are both non-CNF and non-prenex.

This article is organized as follows. Section 2 introduces concepts and notation used throughout the article. Section 3 shows a CEGAR-based algorithm for solving QBF formulas with two levels of quantification. Section 4 shows a CEGAR-based algorithm for solving arbitrary formulas in prenex form; this is an extension of the previous section availing of recursion. Section 5 extends the previous sections by showing an algorithm for solving formulas that are not necessarily in the prenex form. Section 6 shows how CEGAR is integrated into an existing search-based solver as a form of learning. Section 7 provides experimental evaluation of the implemented prototypes. Section 8 overviews related work and finally, Section 9 concludes and provides pointers to future work.

2. Preliminaries

We overview basic notation and concepts used throughout the article; for further details see [1].

Throughout the paper we operate on Boolean variables (x, x_1, etc.), which are composed into formulas by logical connectives with their standard semantics ($\wedge$, $\vee$, $\neg$, $\Rightarrow$, $\Leftrightarrow$). A Boolean formula in *conjunctive normal form (CNF)* is a conjunction of *clauses*, where a clause is a disjunction of *literals*, and a literal is either a variable or its complement. Whenever convenient, a CNF formula is treated as a set of clauses. For a literal l, $\mathsf{var}(l)$ denotes the variable in l, i.e., $\mathsf{var}(\neg x) = \mathsf{var}(x) = x$. We write $|\phi|$ to denote the *size of a formula* ϕ, defined as the sum of the number of connective and variable occurrences in ϕ.

Variable assignments (τ, μ, etc.) are mappings from variables to the constants 0 and 1, represented as $x/1$, $y/0$ etc. The constant 1 represents true and 0 represents false.

Notation. We write $\mathcal{B}^Y$ for the set of assignments to the variables Y.

For a Boolean formula ϕ and an assignment τ we write $\phi[\tau]$ for the substitution of τ in ϕ. A substitution also performs basic simplifications, e.g. $(\neg x \vee y)[x/0] = (\neg 0 \vee y) = 1$. An assignment τ satisfies ϕ if $\phi[\tau] = 1$.

The set of *Quantified Boolean Formulas (QBF)* is the smallest set satisfying the following rules: any Boolean formula is a QBF; if Φ is a QBF, then $\exists x.\,\Phi$ and $\forall x.\,\Phi$ are also QBFs. For $\exists x.\,\Phi$ we say that x is *existentially bound* in Φ; likewise $\forall x.\,\Phi$ makes x *universally bound* in Φ. A variable not bound by any quantifier is called a *free variable*. A QBF with no free variables is called *closed*. Without loss of generality, we assume that for any given formula, once a variable is quantified over, it is never quantified over in the same formula nor it appears as free.

We extend the notion of substitution to QBF so that it first removes the quantifiers of substituted variables and then substitutes all occurrences with their assigned values. For instance, if τ is an assignment to a variable x, then $(Q\,y\,Q\,x\,Q\,z.\,\phi)\,[\tau]$ results in $(Q\,y\,Q\,z.\,\phi[\tau])$.

We define the semantics of QBF by assigning satisfying assignments to a formula. This is done inductively so that $\exists x.\,\Phi$ has the same satisfying assignments as the formula $\Phi[x/0] \vee \Phi[x/1]$ and $\forall x.\,\Phi$ has the same satisfying assignments as the formula $\Phi[x/0] \wedge \Phi[x/1]$.

Notation. For a quantifier type Q we write $\bar{Q}$ to denote the opposing quantifier type. In particular $\bar{Q}$ denotes $\forall$ if $Q = \exists$ and $\bar{Q}$ denotes $\exists$ if $Q = \forall$.

A QBF is in *prenex form* if it is in the form $Q_1 z_1 \ldots Q_n z_n.\phi$ where $Q_i \in \{\forall, \exists\}$, z_i are distinct variables, and ϕ is a propositional formula. The sequence of quantifiers is called *prefix* and the propositional formula *matrix*. The prefix is divided into *quantifier blocks*, $\forall x_1 \ldots \forall x_n$, resp. $\exists x_1 \ldots \exists x_n$, which we denote by $\forall X$, resp. $\exists X$, where $X = \{x_1, \ldots, x_n\}$. We assume that such blocks are maximal. Hence a prenex QBF has the form $Q_1 X_1 \ldots Q_k X_k.\,\phi$ with $Q_i \in \{\exists, \forall\}$, $Q_i \neq Q_{i+1}$. A block $Q_i X_i$ is referred to as *level* i.

Whenever convenient, parts of a prefix are denoted as P with possible subscripts, e.g., $P_1\,\forall X\,P_2.\,\phi$ denotes a QBF with the matrix ϕ and a prefix that contains $\forall X$. If the quantifier of a block Y occurs within the scope of the quantifier of another block X, we say that variables in X are *upstream* of variables in Y and that variables in Y are *downstream* of variables in X.

The pseudocode throughout the article uses the function $\mathtt{SAT}(\phi)$ to represent a call to a SAT solver on a propositional formula ϕ. The function returns a satisfying assignment for ϕ, if such exists, and returns NULL otherwise. In practice SAT solvers require formulas in CNF, which can always be guaranteed in linear time by standard techniques [24,25].

2.1. Game-centric view

An alternative view on QBF semantics is that a QBF is a *game* between the *universal player* and the *existential player* [26]. During the game, the existential player assigns values to the existentially quantified variables and the universal player assigns values to the universally quantified ones. A player can assign a value to a variable only if all variables upstream of it already have a value. The existential player wins if the formula evaluates to 1 and the universal player wins if it evaluates to 0.

Example 1. Consider the formula $\exists x \forall u \exists z. (u \Rightarrow (x \Leftrightarrow z)) \wedge (\neg u \Rightarrow (\neg x \Leftrightarrow z))$. The game $x = 0$, $u = 0$, $z = 0$ is losing for the existential player while $x = 0$, $u = 0$, $z = 1$ is losing for the universal player. In this formula the existential player can always ensure that he wins. One possible winning strategy for the existential player is to play $x = 1$ and $z = u$. This is easily verified, as once x is set to 1, the formula simplifies to $\forall u \exists z. (u \Rightarrow z) \wedge (\neg u \Rightarrow \neg z)$. Now if the universal player sets u to 1, then the formula simplifies to $\exists z. (z)$, where the existential player satisfies the matrix by setting z to 1. The case when the universal player sets u to 0 is analogous. ▲

We note that the order in which values are given to variables in the same block is unimportant. Hence, by a *move* we mean an assignment to variables in a certain block. A concept useful throughout the article are the *winning moves*.

Definition 1 *(Winning move).* Consider a (nonprenex) closed QBF $Q X. \Phi$ and an assignment τ to X. Then τ is called a *winning move* for $Q X$ in $Q X. \Phi$ if $Q = \exists$ and $\Phi[\tau]$ is true, or, $Q = \forall$ and $\Phi[\tau]$ is false.

Notation. We write $\mathcal{M}(Q X. \Phi)$ to denote the set of winning moves for $Q X. \Phi$.

Definition 2 *(Countermove).* Consider a (nonprenex) closed QBF $Q X \bar{Q} Y. \Phi$ and an assignment τ to X and an assignment μ to Y. We say that μ is a *countermove* to τ in $Q X \bar{Q} Y. \Phi$ if $Q = \exists$ and $\Phi[\tau][\mu]$ is false or if $Q = \forall$ and $\Phi[\tau][\mu]$ is true.

Observation 1. *A closed QBF $\exists X. \Phi$ is true iff there exists a winning move for $\exists X$. A closed QBF $\forall X. \Phi$ is false iff there exists a winning move for $\forall X$.*

Observation 2. *Let $Q X \bar{Q} Y. \Phi$ be a (nonprenex) closed QBF and τ be an assignment to X.*

1. *The assignment τ is a winning move for $Q X$ iff there does* not *exists a countermove to τ.*
2. *An assignment μ to Y is a countermove to τ iff μ is a winning move for $\bar{Q} Y$ in $\bar{Q} Y. \Phi[\tau]$.*

Example 2. Consider the formula $\Psi = \forall x \exists y. y \wedge (x \vee \bar{y})$, then the following holds. The assignment $\{x/0\}$ is a winning move for $\forall x$, hence the formula is false from Observation 1. The assignment $\{y/1\}$ is a winning move for $\exists y$ in $\exists y. (y \wedge (x \vee \bar{y}))[x/1]$. Hence, $\{y/1\}$ is a countermove to $\{x/1\}$ due to Observation 2(2). Since there exists a countermove to $\{x/1\}$, the assignment $\{x/1\}$ is *not* a winning move for $\forall x$ in Ψ due to Observation 2(1). ▲

3. 2-level QBF

This section focuses on prenex QBF with 2 levels of quantification, i.e., $\forall X \exists Y. \phi$ or $\exists X \forall Y. \phi$. First we take a look at an even simpler case and that is formulas with a single quantifier, i.e., $\forall X. \phi$ or $\exists X. \phi$. Such formulas represent a 1-move game, where only one of the players is allowed to make a single move upon which the game ends. The problem of deciding 1-quantifier formulas naturally translates to propositional satisfiability. In particular, there exists a winning move for the formula $\exists X. \phi$ if and only if ϕ is satisfiable. Analogously, there exists a winning move for the $\forall$-player for the formula $\forall X. \phi$ if and only if $\neg \phi$ is satisfiable. Further, the satisfying assignments of the respective formula are winning moves for the corresponding player.

This observation motivates the following approach to solving QBF. Start eliminating quantifiers until only one is left at which point invoke a SAT solver. The question is how to eliminate quantifiers. The approach we take here is by *expansion*, in particular we apply the equivalences $\forall x. \Phi = \Phi[x/0] \wedge \Phi[x/1]$ and $\exists x. \Phi = \Phi[x/0] \vee \Phi[x/1]$.

An observation, key to the CEGAR approach, is that in some cases a full expansion is not needed in order to decide the given formula. Instead, we consider *partial expansions* that consider only certain values of variables. The following example illustrates how partial expansions are useful.

Example 3. Let $\phi = (u \vee e_1) \wedge (\bar{u} \vee e_2) \wedge (\bar{e}_1 \vee \bar{e}_2)$ and consider $\forall u \exists e_1 e_2. \phi$. The formula is true and therefore there is no winning move for the $\forall$ player. One could expand e_1 and e_2 with the all 4 possible assignments to e_1 and e_2. However, we observe that considering only two is sufficient. In particular, considering the assignments $\{e_1/1, e_2/0\}$ and $\{e_1/0, e_2/1\}$ yields $(\forall u. \phi[e_1/1, e_2/0] \vee \phi[e_1/0, e_2/1]) = (\forall u. \bar{u} \vee u) = 1$. ▲

A partial expansion may be sufficient to decide a formula but there are also partial expansions that are *not* sufficient as illustrated by the following example.

Algorithm 1: CEGAR Algorithm for 2QBF.

```
input  : $Q X \bar{Q} Y. \phi$
output : $\nu$ if there exists a winning move $\nu$ for $Q X$,
         NULL otherwise
1 $\omega \leftarrow \emptyset$
2 while true do
3     $\alpha \leftarrow (Q = \exists)\,?\, \bigwedge_{\mu\in\omega} \phi[\mu] : \bigvee_{\mu\in\omega} \phi[\mu]$      // build abstraction
4     $\tau \leftarrow (Q = \exists)\,?\, \mathrm{SAT}(\alpha) : \mathrm{SAT}(\neg\alpha)$      // find a candidate
5     if $\tau =$ NULL then return NULL      // no winning move
6     $\mu \leftarrow (\bar{Q} = \exists)\,?\, \mathrm{SAT}(\phi[\tau]) : \mathrm{SAT}(\neg\phi[\tau])$      // find a countermove
7     if $\mu =$ NULL then return $\tau$      // no countermove
8     $\omega \leftarrow \omega \cup \{\mu\}$      // refine
```

Example 4. The formula $\forall u \exists e. (u \vee e) \wedge (\bar{u} \vee \bar{e})$ is true. Expanding e only by $\{e/0\}$ gives $\forall u. u$, which is false. Similarly, expanding e only by $\{e/1\}$ yields $\forall u. \bar{u}$, which is also false. ▲

The key question is, how to discover the right expansions? This is where CEGAR comes into play: Start by a small partial expansion and gradually enlarge it until it becomes sufficient. To this effect, an abstraction corresponds to a partial expansion parameterized by the sets of values which are used to carry out the expansion.

Definition 3 *(ω-abstraction).* Let X, Y be sets of variables and $\omega \subseteq \mathcal{B}^Y$.
The *ω-abstraction* of the closed formula $\forall X \exists Y. \phi$ is $\forall X. \bigvee_{\nu\in\omega} \phi[Y/\nu]$.
The *ω-abstraction* of the closed formula $\exists X \forall Y. \phi$ is $\exists X. \bigwedge_{\nu\in\omega} \phi[Y/\nu]$.

For an abstraction to be useful, it must in some sense approximate the original problem. This is indeed the case for ω-abstraction because for any ω, the set of winning moves of the ω-abstraction is a superset of the winning moves of the original formula. Adding more countermoves to ω decreases the abstraction's set of winning moves. Consequently, if ω contains *all* possible countermoves, the abstraction becomes equivalent to the formula.

Observation 3. *Let $\omega, \omega_1, \omega_2, \subseteq \mathcal{B}^Y$.*

1. $\mathcal{M}(\forall X \exists Y. \phi) = \mathcal{M}(\forall X. \bigvee_{\mu\in\mathcal{B}^Y} \phi[\mu])$
2. $\mathcal{M}(\exists X \forall Y. \phi) = \mathcal{M}(\exists X. \bigwedge_{\mu\in\mathcal{B}^Y} \phi[\mu])$
3. *If* $\omega_1 \subseteq \omega_2$ *then* $\mathcal{M}(\forall X. \bigvee_{\mu\in\omega_2} \phi[\mu]) \subseteq \mathcal{M}(\forall X. \bigvee_{\mu\in\omega_1} \phi[\mu])$
4. *If* $\omega_1 \subseteq \omega_2$ *then* $\mathcal{M}(\exists X. \bigwedge_{\mu\in\omega_2} \phi[\mu]) \subseteq \mathcal{M}(\exists X. \bigwedge_{\mu\in\omega_1} \phi[\mu])$
5. $\mathcal{M}(\forall X \exists Y. \phi) \subseteq \mathcal{M}(\forall X. \bigvee_{\mu\in\omega} \phi[\mu])$
6. $\mathcal{M}(\exists X \forall Y. \phi) \subseteq \mathcal{M}(\exists X. \bigwedge_{\mu\in\omega} \phi[\mu])$

Observation 3 motivates how to use ω-abstraction in a CEGAR loop. The loop first finds a winning move of the current ω-abstraction and then it tests whether that move is also a winning move of the formula being solved. If it is, we are done. If however the winning move for the abstraction is *not* a winning move of the original formula, the abstraction needs to be refined. To facilitate the discussion, a winning move of an abstraction is referred to as a *candidate*.

Two questions remain. How do we know that a candidate is a winning move of the original formula? How is the abstraction refined if the candidate is not a winning move? The first question is directly answered by Observation 2, i.e., a candidate τ is a winning move for $\forall X$ in $\forall X \exists Y. \phi$ iff there is *no* winning move for $\exists Y$ in $\exists Y. \phi[\tau]$. Analogously, a candidate τ is a winning move for $\exists X$ in $\exists X \forall Y. \phi$ iff there is *no* winning move for $\forall Y$ in $\forall Y. \phi[\tau]$. This also gives an answer to the second question, i.e., if there *is* a countermove μ to a candidate τ under some ω-abstraction, the countermove μ is added to ω.

Algorithm 1 shows the above-presented ideas in pseudocode. The algorithm maintains a set of countermoves ω, initialized to the empty set (line 1). In each iteration of the loop it first constructs an abstraction according to Definition 3 (line 3). Note that upon initialization $\omega = \emptyset$ and therefore we have $\alpha = 1$, if $Q = \exists$, since the empty conjunction is semantically equal to 1; analogously, the initial $\alpha = 0$ if $Q = \forall$. Subsequently, the algorithm tries to find a winning move for the abstraction (line 4). If no candidate is found, it means that there is no winning move for the given formula $Q X \bar{Q} Y. \phi$ due to Observation 3 and thus the algorithm terminates. If on the other hand a candidate τ was found,[1] a SAT solver is used to find a countermove for it (line 6). If there is no countermove, then τ is indeed a winning move and the algorithm terminates. If there is a countermove μ, this countermove is added to the set ω (line 8).

[1] Note that in the first iteration, τ is just a random assignment. It is possible to use heuristics to find initial assignment [27].

Example 5. Consider the formula $\exists e_1 e_2 \forall u_1 u_2 . \phi$, where $\phi = (e_1 \vee (u_1 \wedge u_2)) \wedge (e_2 \vee (\bar{u}_1 \wedge \bar{u}_2))$. The following is one possible run of Algorithm 1. Initial ω-abstraction for $\omega = \emptyset$ is $\alpha^1 = 1$. The first SAT call yields the candidate $\tau^1 = \{e_1/0, e_2/0\}$. Subsequently, another SAT call is issued to obtain a countermove $\texttt{SAT}\left(\neg\phi[\tau^1]\right)$, which returns $\mu^1 = \{u_1/0, u_2/0\}$. This countermove yields the refinement $\alpha^2 = (\alpha^1 \wedge e_1) = e_1$. The second iteration produces a candidate $\texttt{SAT}(\alpha^2) = \tau^2$ with $\tau^2 = \{e_1/1, e_2/0\}$ and a countermove $\texttt{SAT}\left(\neg\phi[\tau^2]\right) = \mu^2$ with $\mu^2 = \{u_1/0, u_2/1\}$. The corresponding refinement is $\alpha^3 = (\alpha^2 \wedge e_2) = (e_1 \wedge e_2)$. The candidate in the third iteration is inevitably $\tau^3 = \{e_1/1, e_2/1\}$, which is a winning move for $\exists e_1 e_2$ as there are no countermoves to it, i.e., $\texttt{SAT}(\neg\phi[\tau^3]) = \text{NULL}$. ▲

To conclude the description of this algorithm, we would like to make an important remark about how the expansion is done. Recall that the main motivation for expansion was to get rid of one of the quantifiers in order to enable the use of a SAT solver. Now observe that Algorithm 1 always expands the *innermost* quantifier. So a natural question would be, why not expand the outermost quantifier? While this would be sound, it would not be useful. To illustrate why not, let us consider the case when $\Phi = \exists X \forall Y . \phi$. Expanding the first quantifier by the set of assignments ω yields $\bigvee_{\tau \in \omega} (\forall Y . \phi[\tau])$. Observe that the disjuncts $(\forall Y . \phi[\tau])$ are independent from one another. So Φ is true iff there exists a τ for which $(\forall Y . \phi[\tau])$ is true. Consequently, if Φ is true, it is unnecessary to construct the expansion because it is sufficient to come upon the right τ. If, on the other hand, Φ is false, none of the disjuncts is true for any ω. Hence, in such case it would be necessary to consider the full expansion, i.e., $\omega = \mathcal{B}^X$.

3.1. Properties

Let us now discuss correctness and several other important properties of Algorithm 1. The behavior of the algorithm largely hinges on the following property of ω-abstraction. Once a countermove is included into an ω-abstraction, the abstraction prohibits this countermove. This is formalized by the following lemma.

Lemma 1. *Let $\Phi = Q X \bar{Q} Y . \phi$ and $\omega \subseteq \mathcal{B}^Y$. If τ is a winning move of the ω-abstraction and $\mu \in \omega$ then μ is* not *a countermove to τ in Φ.*

Proof. Consider the case $Q = \exists$. The ω-abstraction is equal to $\exists X . \left(\bigwedge_{\nu \in \omega} \phi[\nu]\right)$. Since $\mu \in \omega$, the abstraction is also equal to $\exists X . \left(\phi[\mu] \wedge \bigwedge_{\nu \in \omega} \phi[\nu]\right)$. Since τ is a winning move for the abstraction, then also $\phi[\mu][\tau]$ is true (note that ϕ does not have any other variables besides X and Y). For contradiction, let μ be a countermove to τ in Φ, i.e., the assignment μ is a winning move for $\forall Y$ in $\forall Y . \phi[\tau]$. From definition of a winning move, it holds $\neg\phi[\tau][\mu]$. This is an immediate contradiction because $\phi[\tau][\mu] = \phi[\mu][\tau]$ as τ and μ are assignments to disjoint sets of variables. The case $Q = \forall$ is shown analogously. □

Lemma 1 lets us derive that candidates in the CEGAR loop cannot repeat.

Proposition 1. *Consider a run of Algorithm 1 on a formula $\Phi = Q X \bar{Q} Y . \phi$. Let τ_i and τ_k be candidates found in the i-th and k-th iterations of the loop, respectively, where $i < k$. Then $\tau_i \neq \tau_k$.*

Proof. Let ω_k be the value of ω at the beginning of the k-th iteration. Let μ_i be a countermove found in the i-th iteration. Since $\mu_i \in \omega_k$, from Lemma 1, μ_i is *not* a countermove to τ_k but at the same time μ_i *is* a countermove to τ_i, hence $\tau_i \neq \tau_k$. □

Proposition 1 itself would be sufficient to show termination of the algorithm but Lemma 1 lets us derive another important property of Algorithm 1, which is that countermoves cannot repeat in the algorithm's loop.

Proposition 2. *Consider a run of Algorithm 1 on a formula $\Phi = Q X \bar{Q} Y . \phi$. Let μ_i and μ_k be countermoves found in the i-th and k-th iterations of the loop, respectively, where $i < k$. Then $\mu_i \neq \mu_k$.*

Proof. Let τ_k be a candidate found in the k-th step and ω_k be the value of ω at the beginning of k-th iteration. Since $\mu_i \in \omega_k$, due to Lemma 1, μ_i is *not* a countermove to τ_k in Φ. Since μ_k *is* a countermove to τ_k, it must be that $\mu_i \neq \mu_k$. □

Proposition 1 and Proposition 2 tell us that neither candidates nor countermoves can repeat in the iteration loop, which yields the following upper bound on the total number of iterations.

Proposition 3. *Consider a run of Algorithm 1 on a formula $Q X \bar{Q} Y . \phi$. Let $k = \min(|X|, |Y|)$, then Algorithm 1 performs at most 2^k iterations of the loop. Consequently, Algorithm 1 requires at most $2 \cdot 2^k$ SAT calls. Excluding the space required in SAT calls, the algorithm requires at most $O(|\phi| \cdot 2^k)$ space.*

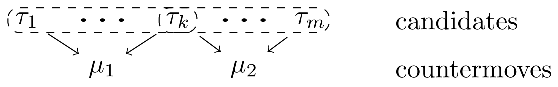


Fig. 1. Candidate–countermove relation.

Proof. There are $2^{|X|}$ possible candidates and $2^{|Y|}$ possible countermoves. Since due to Proposition 1 and Proposition 2 neither candidates nor countermoves repeat, the loop of Algorithm 1 can perform at most $2^{\min(|X|,|Y|)}$ iterations.

In each iteration of the loop of Algorithm 1 the current ω-abstraction is augmented by $\phi[\mu]$, with μ the countermove, and it holds that $|\phi[\mu]| \in O(|\phi|)$. □

While the theoretical upper bound given by Proposition 3 is rather crude, we observe that Proposition 1 and Proposition 2 give us some further insights. A refinement by a countermove μ prevents the algorithm from finding any candidates to which μ is also a countermove.

This is illustrated by Fig. 1, which depicts a situation where we have possible candidates $\tau_1, \ldots, \tau_k, \ldots, \tau_m$ such that μ_1 is a countermove to the candidates $\tau_1 \ldots, \tau_k$ and μ_2 is a countermove to the candidates $\tau_k, \ldots, \tau_m$. Now consider a scenario when Algorithm 1 finds a candidate τ_1 and subsequently the countermove μ_1. Upon refinement, due to Lemma 1, $\tau_1, \ldots, \tau_k$ are excluded from further search since they cannot be winning moves of the abstraction, which now contains μ_1. Hence, the next iteration of the algorithm must consider one of the candidates $\tau_{k+1}, \ldots, \tau_m$. Let's say it considers τ_m and subsequently it finds the countermove μ_2. Once μ_2 is included into the abstraction, the algorithm terminates because the abstraction does not have any winning moves.

What we observe is that the space of possible candidates is diminished more if the countermove just found is a countermove to many possible candidates. This is useful in both situations when there exists a winning move for the given formula or when there does not exist a winning move. If there does not exist a winning move, the algorithm needs to find such set ω of countermoves that covers all the possible candidates (the full set of assignments). If there exists a winning move, the algorithm is more likely to find the winning one if the possible space of candidates is small. The following example illustrates this idea.

Example 6. Let $\Phi = \exists xy \forall q. ((x \wedge q) \vee (x \wedge \neg q)) \wedge ((y \wedge q) \vee (y \wedge \neg q))$ and consider the following run of the algorithm. The first candidate is $\tau^1 = \{x/0, y/0\}$ and countermove $\mu^1 = \{q/1\}$ with the corresponding refinement $\alpha^2 = x \wedge y$. Inevitably, the second candidate $\tau^1 = \{x/1, y/1\}$ is a winning move. Observe that μ^1 is a countermove to all candidates that are not winning moves. ▲

3.2. Solving 2-level quantification with CNF matrices

So far we have only required formulas to be in prenex form and there were no restrictions imposed on the matrix. This section looks in more detail at solving formulas where the matrix is in CNF. CNF is a popular form in SAT and QBF since it enables simple and efficient data structures. An important property of QBF with CNF matrices is that it is sufficient to consider prefixes that end with an existential quantifier block. This is because if there are universal variables at the innermost level, the corresponding literals can be removed from the formula.[2]

Lemma 2. *Let x be a variable and ϕ a CNF. Define ϕ' as ϕ after removing all occurrences of literals containing x. The QBFs $\forall x. \phi$ and ϕ' have the same set of satisfying assignments.*

Proof. Let C be a clause not containing the variable x and let ϕ' be such that $\phi = \phi' \wedge (C \vee x)$. We show that x can be removed safely from the clause $C \vee x$. It holds that $\forall x. (C \vee x) \wedge \phi = ((C[x/0] \vee x[x/0]) \wedge \phi[x/0]) \wedge ((C[x/1] \vee x[x/1]) \wedge \phi[x/1]) = C \wedge \phi[x/0] \wedge \phi[x/1] = \forall x. C \wedge \phi$. □

In the light of Lemma 2, for two-level QPCNF it is only meaningful to consider formulas in the form $\forall X \exists Y. \phi$. Hence, we are always looking for a winning move for the universal quantifier. Note that this is equivalent to looking for such assignment τ to X that makes $\phi[X/\tau]$ unsatisfiable.

The pseudo-code is presented by Algorithm 2. The structure of the algorithm remains the same as Algorithm 1 but several important implementation improvements can be achieved. The workings of the algorithm is greatly influenced by the interface of modern SAT solvers. The vast majority of modern SAT solvers accept formulas in CNF and therefore our objective is to construct formulas passed to the SAT solver in this form. Further, modern SAT solvers enable *incremental* SAT solving; that is, the same SAT solver can be invoked multiple times and the input formula can be strengthened in between the calls. In this fashion incremental SAT solving enables the SAT solver to reuse any information that the solver has learned

[2] The rule of *universal reduction* [28] is a generalization of this property.

Algorithm 2: Two-level CNF solving.

```
input  : $\forall X \exists Y.\phi$, where $\phi$ in CNF
output : $\tau$ if there exists $\tau$ s.t. $\forall Y.\neg\phi[X/\tau]$, NULL otherwise
1  $\rho \leftarrow \{\}$
2  clauseMap $\leftarrow \{\}$
3  while true do
4      $\tau \leftarrow$ SAT($\rho$)                                  // find a candidate
5      if $\tau =$ NULL then return NULL                         // no candidate
6      $\mu \leftarrow$ SAT($\phi[X/\tau]$)                           // find a countermove
7      if $\mu =$ NULL then return $\tau$                        // candidate is winning
       // refine
8      $C_r \leftarrow 0$
9      foreach $C \in \phi$ do
10         $C' \leftarrow C[Y/\mu]$                               // substitute
11         if $C' = 1$ then continue
12         $(z_C, \rho, \text{clauseMap}) \leftarrow$ EncodeNeg($C', \rho$, clauseMap)
13         $C_r \leftarrow C_r \vee z_C$
14     $\rho \leftarrow \rho \cup \{C_r\}$
```

Algorithm 3: Clause negation with caching.

```
1  Function EncodeNeg($C, \phi$, clauseMap)
2  begin
3      if $C = (l)$ then return $\neg l$
4      if clauseMap[$C$] $= k$ then return $k$
5      $z_C \leftarrow$ fresh variable
6      $\phi \leftarrow \phi \wedge \bigwedge_{l \in C}(\neg z_C \vee \neg l)$
7      clauseMap[$C$] $\leftarrow z_C$
8      return $(z_C, \phi, \text{clauseMap})$
```

about the formula. CEGAR-based solving can take advantage of incremental SAT solving because the constructed abstraction is always being strengthened by refinement.

Let us look more closely and how refinement is realized in Algorithm 2. Recall that our objective is to obtain a winning move for $\forall X$. Consequently, to obtain a candidate from an abstraction α, we wish to issue the call SAT($\neg\alpha$). At the same time, each refinement corresponds to disjoining to the current abstraction the formula $\phi[\mu]$, where μ is the current countermove. In order to implement this behavior, we observe that the abstraction α does not need to be explicitly constructed and we will rather directly construct its *negation*.

For such, Algorithm 2 maintains a set of clauses ρ, which is a CNF representation of the negation of the abstraction. The variable ρ is initialized to the empty set of clauses, i.e., semantically it is equal to the constant 1. Since the abstraction is *weakened* with $\phi[\mu]$ in each refinement, the negation is *strengthened* with $\neg\phi[\mu]$. However, because ϕ is in CNF, its negation is in DNF. In order to translate $\neg\phi[\mu]$ to CNF, the standard technique of introducing new variables, sometimes called *Tseitin variables*, is used [24]. Additionally, we take advantage of the formula's specific form, and we use the Plaisted–Greenbaum transformation [25], which introduces implications between the new variables and the encoded structures only in one direction.

Using this standard CNF encoding techniques, one could simply augment ρ with the encoding of $\neg\phi[\mu]$. But there is another improvement stemming from the fact that clauses in $\phi[\mu]$ may reappear in different refinements. This is realized in the function EncodeNeg (Algorithm 3). The function is invoked on each of the clauses of $\phi[\mu]$. During the course of the algorithm, the function maintains a mapping (clauseMap) from clauses to literals, where the mapped literal represents that the clause is false. If EncodeNeg is given a clause C that consists of a single literal, it simply returns the negation of this literal. If EncodeNeg is given a clause C that is already mapped to some Tseitin variable, this variable is returned. Otherwise, EncodeNeg creates a fresh variable z_C and adds to ρ the clauses forcing $\neg C$ whenever z_C is true. Consequently, constructing the clause $C_r = \bigvee_{C \in \phi[\mu]} \text{EncodeNeg}(C, \rho)$ represents that one of the clauses from $\phi[\mu]$ must be false, i.e., $\neg\phi[\mu]$ must be true. Note that any clauses that are reduced to true by the assignment μ are ignored.

Since the above-described refinement only *adds* clauses to ρ, the actual implementation maintains the clauses of ρ in a SAT solver and new clauses are added via the interface of the SAT solver.[3]

Incremental SAT interface can also be utilized for computing countermoves. Given a candidate τ, we wish to issue the SAT call SAT($\phi[\tau]$). For such, an implementation can avail of the *assumption*-based SAT call. In particular, an instance of a SAT solver is constructed at the beginning of the run of the algorithm and is populated with the clauses of ϕ. Whenever a countermove is needed, this SAT solver is called with assumptions representing the assignment τ.

[3] In the SAT solver minisat2.2 this is done by the method addClause.

3.3. Heuristics

The CEGAR loop relies on two calls to a SAT solver and either of these two calls may yield different models for the same abstraction or candidate, respectively. While the correctness of the algorithm is not affected by which of these models is returned, the overall efficiency of the algorithm may be affected. Here we propose two heuristics. One that discriminates between the possible candidates and one that discriminates between the possible counterexamples.

To formulate the heuristic we avail of the *partial MaxSAT* problem [29]. The partial MaxSAT problem is specified by two sets of clauses: a set of *hard* clauses ϕ_h, and a set of *soft* clauses ϕ_s. A solution to the problem is an assignment that satisfies all the hard clauses and maximizes the number of satisfied soft clauses.

Example 7. Consider the MaxSAT problem with hard clauses $\phi_h = \{(x \vee y \vee z), (\neg z \vee w)\}$ and the soft clauses $\phi_s = \{(\neg x), (\neg y), (\neg z), (\neg w)\}$. The assignment $\{x/1, y/0, z/0, w/0\}$ is a possible solution, which satisfies 3 soft clauses. In contrast, $\{x/0, y/0, z/1, w/1\}$, satisfies all the hard clauses but it is not a solution to the problem because it satisfies only 2 soft clauses. Note that solving this problem is equivalent to finding a satisfying assignment to ϕ_h and minimizing the sum $x + y + z + w$. ▲

Candidate heuristic The objective of the heuristic used in computing a candidate (the call $\mathtt{SAT}(\omega)$ in Algorithm 2) is to find such candidates that are likely to be winning moves for the universal player in the formula $\forall X \exists Y. \phi$. Recall that a complete assignment τ to X is a winning move for the universal player if $\phi[X/\tau]$ is unsatisfiable. Motivated by the intuition that larger formulas are more likely to be unsatisfiable than smaller ones, we pick such candidates that maximize the number of clauses of $\phi[X/\tau]$. In another words, we wish to compute such τ that does *not* satisfy as many clauses as possible. Hence, the call $\mathtt{SAT}(\omega)$ is replaced by the following MaxSAT problem:

$$\begin{aligned} \phi_h &= \omega \cup \{\neg z_C \vee \neg l \mid z_C \text{ is a fresh variable}, C \in \phi, l \in C, \text{var}(l) \in X\} \\ \phi_s &= \{z_C \mid C \in \phi\} \end{aligned}$$

Whenever z_C is true, all X literals of C are false. Hence, maximizing the number of variables z_C set to true achieves the objective.

Counterexample heuristic In the refinement step we consider only those clauses that are *not* satisfied by the counterexample μ, i.e., such $C \in \phi$ that $C[Y/\mu] \neq 1$. Hence, the clause $\bigvee z_C$, added in line 7, has less literals the more clauses of ϕ are satisfied by μ. Since, in general, short clauses represent stronger constraints than long clauses, we propose a heuristic that looks for those counterexamples that maximize the number of satisfied clauses in ϕ. Hence, the satisfiability problem $\mathtt{SAT}(\phi[X/\nu])$ is replaced by the following MaxSAT problem:

$$\begin{aligned} \phi_h &= \{C[X/\nu] \mid C \in \phi\} \\ \phi_s &= \left\{C' \mid C \in \phi,\ C' = \{l \mid l \in C, \text{var}(l) \in Y\}\right\} \end{aligned}$$

Implementing heuristics In both of the aforementioned heuristics the corresponding SAT problem is transformed into a MaxSAT problem. Solving these MaxSAT problems in each iteration of the CEGAR loop is not feasible because typically a large number of iterations is required (up to hundreds of thousands) and MaxSAT is significantly more time-consuming than SAT. Hence, in the implementation we compute an approximate solution to the MaxSAT problems by skewing the default decision polarity and variable activity of a SAT solver. Hard clauses are given to the SAT solver as standard clauses without any change. Each soft clause C is represented by the clause $r_C \vee C$ where r_C is a fresh variable. The polarity of the variable r_C is set to 0 and the activity increased. This instructs the SAT solver to set r_C to 0 as soon as possible in the search for a satisfying valuation, which then enforces C to be satisfied. While this approach does not guarantee the optimum, it is commonly used in modern MaxSAT and PB solvers and has been successfully applied to SAT solving with preference [30].

4. Prenex QBF with arbitrary number of quantification levels

This section generalizes algorithm from Section 3 for an arbitrary number of quantifier levels. This generalization follows the basic structure of Algorithm 1 and uses recursion to cope with the multiple levels. The recursion follows the prefix of the given formula starting with the most upstream variables progressing towards more downstream variables. It tries to find a winning move (Definition 1) for variables in a certain block by making recursive calls to obtain winning moves for the downstream variables. The base case of the recursion, i.e., a QBF with one quantifier, is handled by a SAT solver.

We begin by generalizing the observations and concepts introduced in the previous section. A quantifier can be expanded into a propositional operator (disjunction of conjunction) and a partial expansion lets us approximate a set of winning moves. An abstraction of a QBF with at least two quantifiers is obtained by partially expanding the second quantifier. Just as before, a *candidate* refers to a winning move of an abstraction.

Algorithm 4: Basic recursive CEGAR algorithm for QBF.

```
1  Function Solve ($QX.\Phi$)
   input   : $QX.\Phi$ is a closed QBF in prenex form with no adjacent blocks with the same quantifier
   output  : a winning move for $QX.\Phi$ if there is one, NULL otherwise
2  begin
3      if $\Phi$ has no quantifiers then return $(Q=\exists)\,?\,\mathtt{SAT}(\Phi) : \mathtt{SAT}(\neg\Phi)$
4      $\omega \leftarrow \emptyset$
5      while true do
6          $\alpha \leftarrow (Q=\exists)\,?\,\left(\exists X.\ \bigwedge_{\mu\in\omega}\Phi[\mu]\right) : \left(\forall X.\ \bigvee_{\mu\in\omega}\Phi[\mu]\right)$
7          $\tau' \leftarrow$ Solve(Prenex($QX.\alpha$))                    // find a candidate solution
8          if $\tau' =$ NULL then return NULL                            // no winning move
9          $\tau \leftarrow \{l \mid l\in\tau' \wedge \mathrm{var}(l)\in X\}$           // filter a move for $X$
10         $\mu \leftarrow$ Solve($\Phi[\tau]$)                           // find a counterexample
11         if $\mu =$ NULL then return $\tau$                             // no counterexample
12         $\omega \leftarrow \omega \cup \{\mu\}$                                   // refine
```

Observation 4. *Let Φ be QBF with free variables in $X\cup Y$ and ω_1,ω_2 the subset of $\mathcal{B}^Y$.*

1. $\mathcal{M}(\forall X\exists Y.\Phi) = \mathcal{M}(\forall X.\ \bigvee_{\mu\in\mathcal{B}^Y}\Phi[\mu])$
2. $\mathcal{M}(\exists X\forall Y.\Phi) = \mathcal{M}(\exists X.\ \bigwedge_{\mu\in\mathcal{B}^Y}\Phi[\mu])$
3. *If $\omega_1\subseteq\omega_2$ then $\mathcal{M}(\forall X.\ \bigvee_{\mu\in\omega_2}\Phi[\mu])\subseteq\mathcal{M}(\forall X.\ \bigvee_{\mu\in\omega_1}\Phi[\mu])$*
4. *If $\omega_1\subseteq\omega_2$ then $\mathcal{M}(\exists X.\ \bigwedge_{\mu\in\omega_2}\Phi[\mu])\subseteq\mathcal{M}(\exists X.\ \bigwedge_{\mu\in\omega_1}\Phi[\mu])$*

Definition 4 *(ω-abstraction).* Let ω be a subset of $\mathcal{B}^Y$.
The ω-abstraction of a closed QBF $\forall X\exists Y.\Phi$ is the formula $\forall X.\ \bigvee_{\mu\in\omega}\Phi[\mu]$.
The ω-abstraction of a closed QBF $\exists X\forall Y.\Phi$ is the formula $\exists X.\ \bigwedge_{\mu\in\omega}\Phi[\mu]$.

Note that the above observation and definition are generalizations of Observation 3 and Definition 3, which were aimed at formulas with a two-level prefix, while Observation 4 and Definition 4 hold for an arbitrarily long prefix.

Algorithm 4 presents a pseudocode utilizing the above-introduced concepts. The algorithm is presented as a recursive function Solve($QX.\Phi$) that accepts a QBF in prenex form with maximal quantifier blocks. The function returns a winning move for QX, if such exists, it returns NULL otherwise.

If Φ does not contain any quantifiers (it is a propositional formula), a SAT solver is used to find a winning move (line 3). In the general case, utilizing this CEGAR paradigm, it initializes the set of countermoves ω to the empty set (line 4) and grows it by encountered countermoves (line 12).

In contrast to the 2-level case, the ω-abstraction cannot be directly solved. This is due to the abstraction not being in prenex form. Hence, before invoking the recursive call to obtain a candidate (line 7), the algorithm must compute a prenex form of the abstraction. Consequently, the prenexed version of the abstraction contains some fresh variables. These need to be filtered out in order to obtain values for the X variables only.

Let us look at this process in more detail. Consider the case for $\exists X.\Phi$ (the case $Q=\forall$ is analogous) and consider that Φ has at least two levels of quantification. Hence, the input formula is of the form $\exists X\forall Y\exists Z\mathcal{P}.\phi$, where $\mathcal{P}$ is the rest of the prefix of the formula and ϕ is its matrix. Note that $\mathcal{P}$ is empty or it starts with the universal quantifier.

Further, let us assume that the CEGAR loop carried out k iterations and thus set ω contains k countermoves, i.e., $\omega=\{\mu_1,\ldots,\mu_k\}$. The abstraction constructed on line 6 is equal to $\exists X.\ \bigwedge_{i\in 1..k}\Phi[\mu_k]$. Prenexing this abstraction means introducing for each $i\in 1..k$ a fresh set of variables Z^i for the variables Z and introducing fresh variables for the variables appearing in the prefix $\mathcal{P}$. Hence, the prenexed form is equal to the following.

$$\exists X Z^1\ldots Z^k\mathcal{P}^1\ldots\mathcal{P}^k.\ \bigwedge_{i\in 1..k}\Phi[\mu_i, Z/Z^i, \mathcal{P}/\mathcal{P}^i] \tag{1}$$

where Z/Z^i stands for the substitution of each variable of the set Z with the corresponding (fresh) variable from Z^i. Similarly, $\mathcal{P}^i$ is the i-th fresh copy of the prefix $\mathcal{P}$ and $\mathcal{P}/\mathcal{P}^i$ stands for replacing the corresponding variables with their fresh copies.

If formula (1) has a winning move τ', it will contain values for the freshly introduced variables Z^i. Observe that once these values are filtered out from τ' (line 9), this is a winning move for the unprenexed version of the abstraction.

Example 8. Consider the QBF $\exists vw.\Phi$, where

$$\Phi = \forall u\exists xy.(v\vee w\vee x)\wedge(\bar{v}\vee y)\wedge(\bar{w}\vee y)\wedge(u\vee\bar{x})\wedge(\bar{u}\vee\bar{y}),$$

Algorithm 5: Recursive CEGAR algorithm for multi-games.

```
1  Function RAReQS ($Q\,X.\{\Phi_1,\ldots,\Phi_n\}$)
2  output: a winning move for $Q\,X.\{\Phi_1,\ldots,\Phi_n\}$ if there is one; NULL otherwise
3  begin
4      if $\Phi_i$ have no quantifiers then                          // base case
5          return $Q = \exists\,?\,\texttt{SAT}(\bigwedge_i \Phi_i) : \texttt{SAT}(\neg(\bigvee_i \Phi))$
6      $\alpha \leftarrow Q\,X.\{\}$
7      while true do
8          $\tau' \leftarrow$ RAReQS($\alpha$)                          // find a candidate
9          if $\tau' =$ NULL then return NULL
10         $\tau \leftarrow \{l \mid l \in \tau' \wedge \mathrm{var}(l) \in X\}$          // filter a move for X
11         for $i \leftarrow 1$ to $n$ do
12             $\mu_i \leftarrow$ RAReQS($\Phi_i[\tau]$)                // find a countermove
13         if ($\mu_i =$ NULL for all $i \in \{1..n\}$) then return $\tau$
14         let $l \in \{1..n\}$ be s.t. $\mu_l \neq$ NULL
15         $\alpha \leftarrow \texttt{Refine}(\alpha, \Phi_l, \mu_l)$                  // refine
```

and the candidates $\{v/1, w/1\}$ and $\{v/0, w/0\}$, and corresponding counterexamples $\{u/1\}$ and $\{u/1\}$. Refinement yields the abstraction $\exists v w.\, \Phi[u/1] \wedge \Phi[u/0]$, with the prenex form $\exists v w x y x' y'.\ (v \vee w \vee x) \wedge (\bar{v} \vee y) \wedge (\bar{w} \vee y) \wedge (\bar{y}) \wedge (v \vee w \vee x') \wedge (\bar{v} \vee y') \wedge (\bar{w} \vee y') \wedge (\bar{x}')$ with no winning move and the algorithm terminates with the return value NULL. ▲

4.1. Improving recursive CEGAR-based algorithm

The 2-level algorithm already has a significant memory consumption since in each iteration of the loop the abstraction is increased by the size of the input formula. We will show that recursive calls may further exponentially amplify this behavior. Consider the following formula.

$$\exists X_1 \forall Y_1 \exists Z \forall Y_2 \mathcal{P}.\, \phi \tag{2}$$

where $\mathcal{P}$ is the rest of the quantifier prefix and ϕ the matrix of the formula. Further consider an ω-abstraction of (2) for $\omega = \{\mu^1_1, \ldots, \mu^1_{n_1}\}$ corresponding to n_1 iterations of the CEGAR loop, which upon prenexing is equal to the following.

$$\exists X_1 Z^1, \ldots, Z^n \forall Y_2^1 \ldots, Y_2^n \mathcal{P}^1 \ldots, \mathcal{P}^n.\ \bigwedge\nolimits_{i \in 1..n_1} \phi[Z/Z^i, Y/Y^1, \mathcal{P}/\mathcal{P}^i] \tag{3}$$

The algorithm subsequently invokes the recursive call on formula (3) on line 7 in order to obtain a candidate. The recursive call operates on a matrix whose size is in $O(n_1|\phi|)$, i.e., n_1-times bigger than the original matrix. If the recursive call performs n_2 iterations of the CEGAR loop, it will produce an abstraction that will be $n_1 n_2$ bigger than the original matrix ϕ. In general, if the algorithm iterates n_i times at a recursion level i, the abstraction at level k is of the size $O(n_1 \cdots n_k \cdot |\phi|)$.

Clearly, such blowup would prohibit any practical application of the algorithm for larger number of quantification levels. To cope with this issue, we exploit the form of the formulas that the algorithm handles. In the case of the existential quantifier, the abstraction is a conjunct and it is a disjunct in the case of the universal quantifier. For the sake of uniformity, we bridge these two forms by introducing the notion of a *multi-game* where a player tries to find a move that wins multiple formulas simultaneously.

Definition 5 *(Multi-game).* Let Q be a quantifier and X be a set of variables. Let $\Phi_1, \ldots, \Phi_n$ be a set of prenex QBFs such that each Φ_i starts with $\bar{Q}$ or has no quantifiers. Additionally, the free variables of each Φ_i must be in X and all Φ_i have the same number of quantifier blocks. A *multi-game* is the expression $Q\,X.\{\Phi_1, \ldots, \Phi_n\}$. We refer to the formulas Φ_i as *subgames* and $Q\,X$ as the *top-level prefix*.

A *winning move* for a multi-game is an assignment to the variables X such that it is a winning move for each of the formulas $Q\,X.\,\Phi_i$.

Observe that the set of winning moves of a multi-game $Q\,X.\{\Phi_1, \ldots, \Phi_n\}$ is the same as the set of winning moves of the QBF $\forall X.\,(\Phi_1 \vee \cdots \vee \Phi_n)$ for $Q = \forall$ and it is the same as $\exists X.\,(\Phi_1 \wedge \cdots \wedge \Phi_n)$ for $Q = \exists$. And, any prenex QBF $Q\,X.\,\Phi$ corresponds to a multi-game with a single subgame $Q\,X.\{\Phi\}$.

Algorithm 5 shows an algorithm to solve multi-games. Just as Algorithm 4, this algorithm is represented as a recursive function named RAReQS. As input the algorithm accepts a multi-game and the abstraction it constructs is again a multi-game.

To determine whether a candidate τ is a winning move, it tests whether it is a winning move for the subgames in turn. If it finds a subgame Φ_i such that the opponent $\bar{Q}$ wins $\Phi_i[\tau]$ by a move μ_i, then $\Phi_i[\mu_i]$ is used to refine the abstraction.

Since an abstraction is a multi-game, it seems natural to add $\Phi_i[\mu]$ to the set of its subgames. This, however, cannot be done right away because the formula $\Phi_i[\mu_i]$ is not in the right form. In particular, for a multi-game, all the subgames must start with the opposing quantifier with respect to the top-level prefix. Hence, if Φ_i is of the form $\bar{Q}\, Y\, Q\, X_1.\, \Psi_i$ and $\mu_i \in \mathcal{B}^Y$, then $\Phi_i[\mu_i] = Q\, X_1.\, \Psi_i[\mu_i]$. To bring the formula into the right form, we introduce fresh variables for the variables X_1 and move them into the top-level prefix. More precisely, the function $\texttt{Refine}(\alpha, \Phi_l, \mu_l)$ is defined as follows (observe that the subgames remain in prenex form).

$$\texttt{Refine}\big(Q\,X.\{\Psi_1,\ldots,\Psi_n\},\ \bar{Q}\,Y\,Q\,Z.\,\Psi,\ \mu\big) := Q\,X Z'.\{\Psi_1,\ldots,\Psi_n,\Psi[\mu, Z/Z']\}$$

where Z' are fresh duplicates of the variables Z

$$\texttt{Refine}\big(Q\,X.\{\psi_1,\ldots,\psi_n\},\ \bar{Q}\,Y.\,\psi,\ \mu\big) := Q\,X.\{\psi_1,\ldots,\psi_n,\psi[\mu]\}$$

where ψ is a propositional formula (and no variable duplicates are needed)

Similarly to Algorithm 4, after the refinement, the abstraction's top-level prefix contains additional variables besides the variables X. Hence, values for these variables are filtered out if a winning move for the abstraction is found.

4.2. Properties of the algorithms

Similar properties to the ones for 2QBF (see Section 3.1) can be derived for Algorithm 4 and Algorithm 5. Following the same argumentation as in Section 3.1, we can derive that once a counterexample μ is found in Algorithm 4, μ cannot be a countermove to any assignment that is a winning move for the future forms of the abstraction. Consequently, no candidate or counterexample repeats. From which follows that the loop is terminating and for a formula $Q\,X\bar{Q}\,Y.\,\Phi$ the number of its iterations is bounded by the number of possible assignments to the variables X and Y, i.e., $\min(2^{|X|}, 2^{|Y|})$. In the worst case, in each iteration the abstraction grows by the size of Φ.

For a multi-game $Q\,X.\,\{\Phi_1,\ldots,\Phi_n\}$ in the CEGAR loop of Algorithm 5 no candidates repeat but counterexamples may repeat. However, for a given $i \in 1..n$, a counterexample μ_i does not repeat. More precisely there are no two distinct iterations of the loop with the corresponding candidates and counterexamples $\tau_1, \mu_1, \tau_2, \mu_2$, such that $\mu_1 = \mu_2$ and μ_1 is a winning move for both $\Phi_i[\tau_1]$ and $\Phi_i[\tau_2]$ for some i. This demonstrates termination with the upper bound for the number of iterations as $\min(2^{|X|}, n \cdot 2^{|Y|})$. In the worst case, in each iteration the abstraction grows by the maximum of the sizes of the subgames $\Phi_1,\ldots,\Phi_n$. Soundness and completeness of the Algorithms 4 and 5 are direct consequences of Observation 4.

4.3. Implementation details

We have implemented a prototype[4] of RAReQS in C++, supporting the QDIMACS format, with the underlying SAT solver `minisat 2.2` [31].

The implementation has several distinctive features. In Algorithm 5, an abstraction computed within a sub-call is forgotten once the call returns. This may lead to repetition of work and hence the solver supports maintaining these abstractions and strengthening them gradually, similarly to the way SAT solvers provide *incremental* interface. This incremental approach, however, tends to lead to unwieldy memory consumption and therefore, it is used only when the given multigame's subgames have 2 or fewer quantification blocks.

If an assignment τ is a candidate for a winning move that turns out *not* to be a winning move, the refinement guarantees that τ is not a solution to the abstraction in the future iterations of the CEGAR loop. This knowledge enables us to make the subcall for solving the abstraction more efficient by explicitly disabling τ as a winning move for the abstraction. We refer to this technique as *blocking* and it is similar to the refinement used in certain SMT solvers [32,33].

Throughout its course, the algorithm may produce a large number of new formulas, either by substitution or refinement. Since these formulas tend to be simpler than the given one, they can be further simplified by standard QBF *preprocessing* techniques. The implementation uses unit propagation and *monotone* (pure) literal rule [34]. These simplifications introduce the complication that in a multi-game $Q\,X.\{\Phi_1,\ldots,\Phi_n\}$ the individual subgames might not necessarily have the same number of quantifier levels. In such case, all games with no quantifiers are immediately put into the abstraction before the loop starts.

5. Non-CNF, non-prenex QBF

In order to construct a non-CNF solver, we build on the ideas of multi-games, which were introduced in the previous section. Recall that a multi-game $\forall X.\{\exists Y.\,\Phi_1, \exists Y.\,\Phi_2\}$ corresponds to the formula $\forall X.\,(\exists Y.\,\Phi_1) \vee (\exists Y.\,\Phi_2)$. So in fact, multi-games are a special cases of non-CNF QBFs. In multi-games, we were able to play each subgame separately and also refine by one subgame at a time and thus mitigating space explosion of the abstractions being constructed (see introduction to Section 4.1).

[4] Available from http://sat.inesc-id.pt/~mikolas/sw/areqs.

$$
\begin{aligned}
F &::= \forall X.\, F^{\forall} \mid \exists X.\, F^{\exists} \\
F^{\forall} &::= F^{\forall} \vee F^{\forall} \mid F^{\forall} \wedge F^{\forall} \mid \exists X.\, F^{\exists} \mid x \mid \neg x \\
F^{\exists} &::= F^{\exists} \vee F^{\exists} \mid F^{\exists} \wedge F^{\exists} \mid \forall X.\, F^{\forall} \mid x \mid \neg x
\end{aligned}
$$

Fig. 2. Grammar for *Alternating form*.

Algorithm 6: Non-prenex and non-CNF solving.

```
1  Function RAReQS-NN ($Q\,X.\phi$)
2  begin
3      if $\phi$ has no quantifiers then return $(Q = \exists)\,?\,\mathrm{SAT}(\phi) : \mathrm{SAT}(\neg\phi)$
4      $\omega \leftarrow \emptyset$
5      while true do
6          $\alpha \leftarrow \mathtt{Abstract}(Q, \phi, \omega)$                  // build abstraction
7          $\tau' \leftarrow \mathtt{RAReQS\text{-}NN}(\mathtt{Prenex}(Q\,X.\alpha))$   // find a candidate
8          if $\tau' = \mathrm{NULL}$ then return NULL                // no winning move
9          $\tau \leftarrow \{l \mid l \in \tau' \wedge \mathrm{var}(l) \in X\}$      // filter a move for $X$
10         $R \leftarrow \mathtt{Test}(Q, \phi, \tau)$                     // test candidate
11         if $r = \mathrm{NULL}$ then return $\tau$                    // no counterexample
12         $\omega \leftarrow \omega \cup R$                               // refine
```

In the non-CNF case, we take this idea one step further. We consider a QBF $Q\,X.\,\Phi$ where Φ is a tree (or a directed acyclic graph) with internal nodes representing Boolean operators ($\wedge$, $\vee$) and leaves representing QBFs with the opposite quantifier ($\bar{Q}$). So for instance, we consider $\exists X.\,(\forall Y.\,\Phi_1) \wedge (\forall Z.\,\Phi_2)$ but not $\exists X.\,(\forall Y.\,\Phi_1) \wedge (\exists Z.\,\Phi_2)$. To construct an abstraction of a QBF in this form, we construct a tree with the same internal nodes but replace the leaves with their partial expansions.

Let us now look at the above presented ideas more precisely. We consider QBFs in the form given by the grammar in Fig. 2. As opposed to an arbitrary QBF, the grammar forces quantifiers to alternate, i.e., if some quantification $Q\,X_2$ is in the scope of quantification $Q\,X_1$, there must be another quantification $\bar{Q}\,Z$ in the middle of the two. Further, negations must be only used on variables[5] and we require the top-level expression to be a quantifier. Since this form requires quantifiers to alternate, we name it *alternating form*.

To convert an arbitrary QBF to alternating form, all negations are pushed inwards by using standard equivalences ($\neg\exists X.\,\Psi = \forall X.\,\neg\Psi$, etc.) and non-alternating quantifiers are brought together by prefixing. So for instance the formula $\exists x.\neg(\forall y.\, y \vee x) \wedge (\forall y.\, y \wedge x)$ becomes $\exists x y'.\,(\bar{y}' \wedge \bar{x}) \wedge (\forall y.\, y \wedge x)$. Note that alternating form also requires that the top-level expression is a quantifier which can always be guaranteed by adding a "dummy" existential quantification, i.e., $(\forall x.\, x) \wedge (\exists y.\, y)$ becomes $\exists z.\,(\forall x.\, x) \wedge (\exists y.\, y)$.

Having a quantifier at the top level simplifies presentation of the algorithm because the definition of a winning move still applies (Definition 1) and we can decide a given formula by determining whether there is a winning move for the top-level quantification. Algorithm 6 presents an algorithm for formulas in alternating form with several facets not yet explained. These will be discussed in the following sections. Algorithm 6 provides the big picture at this point.

The structure of the algorithm is almost identical to the algorithms we have seen so far. An abstraction α is constructed from the given formula and a parameter ω. The parameter ω is explained later but the intuition behind it is that it determines how the opponent's quantifiers are expanded (similarly as abstraction was parameterized by the set of countermoves in previous sections). Then the algorithm finds a winning move for the abstraction and subsequently tests whether this is really a winning move for the given problem. If the abstraction has no winning move, the given problem also does not have a winning move. If the winning move for the abstraction is a winning move for the given problem, the algorithm terminates. Otherwise, the algorithm refines the abstraction. To fully explain the algorithm the following facets need to be addressed:

1. form of the abstraction (function `Abstract`)
2. test that a given move is a winning move (function `Test`)
3. refinement

Before we look at these aspects of the algorithm, we introduce the following notation, which enables us to unify certain operations for the universal and existential quantifier.

Notation

$$
\begin{array}{lll}
\mathrm{ALL}^{\exists} = \wedge & \mathrm{SOME}^{\forall} = \wedge & \top^{\exists} = \bot^{\forall} = 1 \\
\mathrm{ALL}^{\forall} = \vee & \mathrm{SOME}^{\exists} = \vee & \top^{\forall} = \bot^{\exists} = 0
\end{array}
$$

[5] In the actual implementation we relax this condition by allowing negations on expressions containing no quantifiers.

Algorithm 7: Compute raw abstraction.

```
1  Function RawAbstract ($Q$, $\Phi$, $\omega$)
2  begin
3      if $\Phi = l$ then return $l$
4      if $\Phi = \mathrm{ALL}^{Q}(\Phi_1, \Phi_2)$ then
5          return $\mathrm{ALL}^{Q}(\mathtt{RawAbstract}(Q, \Phi_1, \omega), \mathtt{RawAbstract}(Q, \Phi_2, \omega))$
6      if $\Phi = \mathrm{SOME}^{Q}(\Phi_1, \Phi_2)$ then
7          return $\mathrm{SOME}^{Q}(\mathtt{RawAbstract}(Q, \Phi_1, \omega), \mathtt{RawAbstract}(Q, \Phi_2, \omega))$
8      if $\Phi = \bar{Q}Y.\Psi$ then
9          $\omega_Y \leftarrow \{\mu \mid (i, \mu) \in \omega,\ i = \mathtt{ID}(\Phi)\}$
10         return $\mathrm{ALL}^{Q}_{\mu \in \omega_Y} \Psi[\mu]$
```

The intuition behind the operator is that in the formula $QX.\mathrm{ALL}^{Q}(\Phi_1, \Phi_2)$ the player Q must find a winning move for both Φ_1 and Φ_2. In contrast, $\mathrm{SOME}^{Q}(\Phi_1, \Phi_2)$ means that Q must find a winning move for one of the Φ_1, Φ_2. In the same spirit, the top and the bottom constants are defined for the two players. These are Boolean constants that correspond to either of the players to win or lose, respectively. The operators ALL^{Q} and SOME^{Q} are treated as prefix operators. Note that the operators are commutative and associative.

Observation 5. *Let Φ_1 and Φ_2 be QBFs and X a set of variables and τ a total assignment to X. The assignment τ is a winning move for $\forall X.\mathrm{ALL}^{\forall}(\Phi_1, \Phi_2)$ iff $\Phi_1[\tau]$ and $\Phi_2[\tau]$ are both false; it is a winning move for $\exists X.\mathrm{ALL}^{\exists}(\Phi_1, \Phi_2)$ iff $\Phi_1[\tau]$ and $\Phi_2[\tau]$ are both true. The assignment τ is a winning move for $\forall X.\mathrm{SOME}^{\forall}(\Phi_1, \Phi_2)$ iff one of $\Phi_1[\tau]$, $\Phi_2[\tau]$ is false; it is a winning move for $\exists X.\mathrm{SOME}^{\exists}(\Phi_1, \Phi_2)$ iff one of $\Phi_1[\tau]$, $\Phi_2[\tau]$ is true.*

Observation 6. *The following equations hold.*

$$\begin{aligned} \mathrm{SOME}^{Q}(\phi, \top^{Q}) &= \top^{Q} & \qquad \mathrm{SOME}^{Q}(\phi, \bot^{Q}) &= \phi \\ \mathrm{ALL}^{Q}(\phi, \bot^{Q}) &= \bot^{Q} & \qquad \mathrm{ALL}^{Q}(\phi, \top^{Q}) &= \phi \end{aligned}$$

5.1. Abstraction

As hinted above, the crux of the abstraction is to replace the opponent's quantifiers by Boolean connectives. The following equations are the basis for these expansions.

$$\begin{aligned} \forall x.\,\Phi &= \Phi[x] \wedge \Phi[\bar{x}] = \mathrm{ALL}^{\exists}(\Phi[x], \Phi[\bar{x}]) \\ \exists x.\,\Phi &= \Phi[x] \vee \Phi[\bar{x}] = \mathrm{ALL}^{\forall}(\Phi[x], \Phi[\bar{x}]) \end{aligned} \tag{4}$$

Equations (4) tell us that the opposing quantifier is expanded by the ALL^{Q} operator. So in order to construct an abstraction of a formula $QX.\Phi$ we find each expression $\bar{Q}Y.\Psi$ and replace it by the expression $\mathrm{ALL}^{Q}_{\tau \in \omega_Y} \Psi[\tau]$ where ω_Y is the set of assignments that are considered for this expansion. Recall, however, that the algorithm operates on formulas in alternating form. And, performing these expansions might violate this form. Hence, we split the computation of an abstraction into two phases: the computation of a *raw abstraction* α' and a prenexing operation that brings α' to alternating form.

Throughout the computation, the algorithm remembers how the given formula has been expanded so far and expands it further in refinements (Algorithm 6). To do so, it maintains a variable ω, which determines for each subexpressions of the form $\bar{Q}Y.\Psi$ that is not within the scope of another quantifier how it should be expanded, i.e., the variable ω parameterizes the abstraction. Now let us look more closely at the contents of ω.

We assume that each node in a formula is uniquely identified by an identifier which can be obtained by the function ID. The variable ω comprises pairs (i, μ) where i is an identifier of some node $\bar{Q}Y.\Psi$ in the given formula and μ is a total assignment to Y. Each such pair means that Y should be expanded with the assignment μ. The function RawAbstract (Algorithm 7) realizes this idea by traversing recursively the given formula and expanding each quantified node with the pertaining assignments. As noted before, such transformation constructs raw abstraction because it might not necessarily be in alternating form. Hence, the function Abstract used in Algorithm 6 first calls RawAbstract and applies prefixing in order to bring it into alternating form.

Fig. 3(a) shows an example of an abstraction of the formula $\forall x.(x \vee \bar{y}) \wedge (\exists z.z)$. The node $\exists z.z$ is expanded by the assignments $z/0$ and $z/1$. Fig. 3(b) shows the same abstraction rewritten using the ALL^{Q} and SOME^{Q} operators.

5.2. Winning move test

The function Test serves two purposes: the first purpose is to test whether a given candidate is a winning move; the second purpose is provide how the current abstraction should be refined. More precisely, for a QBF $QX.\Phi$ and a total assignment τ to the variables X the function decides whether τ is a winning move for QX in $QX.\Phi$. The function either

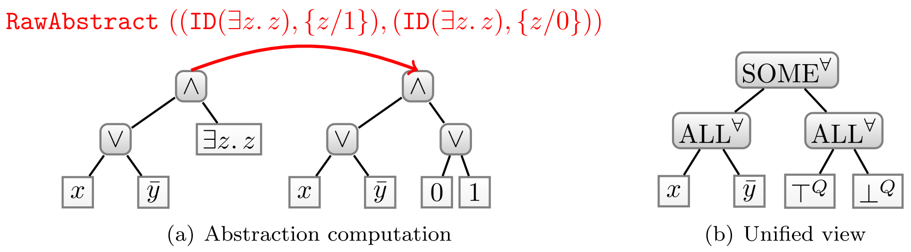

RawAbstract ((ID(∃z. z), {z/1}), (ID(∃z. z), {z/0}))
∧
∨
∃z. z
x
ȳ
∨
∨
x
ȳ
0
1
(a) Abstraction computation
SOME∀
ALL∀
ALL∀
x
ȳ
⊤Q
⊥Q
(b) Unified view

Fig. 3. Abstraction example.

Algorithm 8: Testing if a given move is a winning move.

```
1  Function Test ($Q$, $\phi$, $\tau$)
   input   : $\phi$ is a QBF with all its free variables assigned by $\tau$
   output  : refinement if $\tau$ is not winning move for $Q$ in $\phi$, NULL otherwise
2  begin
3      if $\phi = l$ then return $(\phi[\tau] = \top^Q)$ ? NULL : {}
4      if $\phi = \text{ALL}^Q(\phi_1, \phi_2)$ then
5          $\beta_1 \leftarrow$ Test($Q, \phi_1, \tau$)
6          if $\beta_1 \neq$ NULL then return $\beta_1$
7          $\beta_2 \leftarrow$ Test($Q, \phi_2, \tau$)
8          if $\beta_2 \neq$ NULL then return $\beta_2$
9          return NULL
10     if $\phi = \text{SOME}^Q(\phi_1, \phi_2)$ then
11         $\beta_1 \leftarrow$ Test($Q, \phi_1, \tau$)
12         if $\beta_1 =$ NULL then return NULL
13         $\beta_2 \leftarrow$ Test($Q, \phi_2, \tau$)
14         if $\beta_2 =$ NULL then return NULL
15         return $\beta_1 \cup \beta_2$
16     if $\phi = \bar{Q} Y. \psi$ then
17         $\mu \leftarrow$ Solve($\bar{Q} Y. \psi[\tau]$)
18         if $\mu =$ NULL then return NULL
19         return {(ID($\phi$), $\mu$)}
```

returns NULL if τ *is* a winning move or it returns a set of pairs (i, μ), which are pairs that are eventually added to the variable ω (see Algorithm 6). Effectively, if a pair (i, μ) is added to the return value, the node with identifier i is expanded by the assignment μ.

Algorithm 8 shows pseudocode for the function Test. The function recursively traverses the given formula until it reaches a quantified expression where it invokes the solving procedure to test whether this subexpression is won by τ. Note that any quantified expression found must be of the opposing quantifier due to the formula being in alternating form. If τ leads to a loss in some subexpression $\bar{Q} Y. \Psi$, there is a winning move μ for $\bar{Q} Y$ in $\bar{Q}. \Psi[\tau]$. This move μ is used to expand this subexpression in the upcoming abstractions. This is very much similar to countermoves were used in the algorithms for prenex QBFs.

Now let us look at how the function Test aggregates computed expansions in the internal notes of the formula. If the function Test operates on a ALL^Q node, it only needs to check whether one of the children needs to be refined. If τ leads to a loss in one of the children, that child is expanded. Otherwise, NULL is returned (no refinement needed). However, in the case of nodes of type SOME^Q, computing refinements is slightly more complicated. If τ yields a win for one of the children, no expansion is needed in that subtree. In the opposite case, when τ loses for all the children, *all* these children are expanded.

Observe that when children of ALL^Q and SOME^Q nodes are treated, the operations are short-circuited. So for instance once the function discovers that τ leads to a loss in some child of a ALL^Q node, it does not evaluate the other children. Analogously, if τ leads to a win in some child of a SOME^Q node, the function stops without producing an expansion. This is also a motivation for why the aggregations of expansions are different for the two types of nodes. In the case of a ALL^Q node, there is a single child that calls for expansion. In the case of a SOME^Q node, the recursive calls have produced expansions for all the children. It would be correct to randomly pick just one of the children for expansion. Like so, however, the computed expansions would go to waste.

Remark When the given formula is given a single literal, τ might lead to a loss without generating any expansion to be added to the abstraction. ▲

6. CEGAR as a learning technique in DPLL

The previous section shows that CEGAR can give rise to a complete and sound algorithm for QBF. In this section we show that CEGAR enables us to extend existing DPLL solvers with an additional learning technique. We will employ the *ghost variables* and *game sequents* techniques to more easily present this technique.

6.1. Ghost variables and sequent learning

In this subsection, we briefly describe the techniques of ghost variables and sequent learning. A more complete treatment may be found in [35].

We employ *ghost variables* to provide a modification of the Tseitin transformation that is symmetric between the two players. The idea of using a symmetric transformation was first explored in [36], which performed the Tseitin transformation twice: once on the input formula, and once on its negation.

For each subformula of the original input formula, we introduce two *ghost variables*: an existentially quantified variable $g^\exists$ and a universally quantified variable $g^\forall$. We say that $g^\exists$ and $g^\forall$ *represent* the labeled subformula. Variables that occur in the original input formula are called *input variables*, in distinction to ghost variables. Ghost variables are always downstream of all input variables.

We now introduce a semantics with ghost variables for the game formulation of QBF. As in the Tseitin transformation, the existential player should lose if an existential ghost variable $g^\exists$ is assigned a different value than the subformula that it represents. Additionally, the universal player should lose if an universal ghost variable $g^\forall$ is assigned a different value than the subformula that it represents.

As noted in Section 2, we write "$\phi[\pi]$" to denote the result of substituting assignment ϕ into formula ϕ. The assignment π may contain ghost variables, but a subformula labeled by a ghost variable is not replaced by the assigned value of the ghost variable. For example, if g_1^Q represents $x \wedge y$ and $\pi_1 = \{g_1^Q/1\}$, then $(x \wedge y)[\pi_1]$ evaluates to $(x \wedge y)$, and $g_1^Q[\pi_1]$ evaluates to true.

Definition 6 *(Consistent assignment to ghost literal).* Let $Q \in \{\exists, \forall\}$; let π be an assignment; let g^Q be a ghost literal; and let ξ be the formula represented by g^Q. We say that g^Q is assigned **consistently** under π iff $g^Q[\pi] = \xi[\pi]$. We say g^Q is assigned **inconsistently** under π iff $g^Q[\pi] = \neg\xi[\pi]$.

For example, if g_1^Q represents $x \wedge y$, then g_1^Q is assigned consistently under $\{g_1^Q/0, x/0\}$, while it assigned inconsistently under $\{g_1^Q/0, x/1, y/1\}$. Under $\{g_1^Q/0\}$, g_1^Q is not said to be either consistently or inconsistently assigned.

Definition 7 *(Winning under a total assignment).* Given a formula Φ, a quantifier type $Q \in \{\exists, \forall\}$, and an assignment π to all the input variables and a subset of the ghost variables, we say "Player Q **wins** Φ under π" iff both of the following conditions hold true:

1. $\Phi[\pi] = \begin{cases} \mathsf{true} & \text{if } Q \text{ is } \exists \\ \mathsf{false} & \text{if } Q \text{ is } \forall \end{cases}$
2. Every ghost variable owned by Q in $\text{vars}(\pi)$ is assigned consistently.
   (Intuitively, a winning player's ghost vars must "respect the encoding".)

For example, if $\Phi = \exists e \forall u. (e \wedge u)$ and g labels $(e \wedge u)$ then neither player wins Φ under $\{e/0, u/1, g^\forall/1, g^\exists/0\}$. The existential player fails to win because $\Phi[\pi] = \mathsf{false}$, and the universal player fails to win because the ghost variable $g^\forall$ is assigned inconsistently, since $g^\forall[\pi] = \mathsf{true}$ but the formula represented by $g^\forall$ (i.e., the conjunction $e \wedge u$) evaluates to false.

Definition 8 *(Losing under an assignment).* Given a formula Φ and an assignment π, we define the phrase "Player Q **loses** Φ under π" recursively. We say "Player Q **loses** Φ under π" iff either:

1. Player Q does not win Φ under π and every input variable is assigned by π, or
2. there is an outermost unassigned input variable x such that either:
   (a) Player Q loses Φ under both $\pi \cup \{x/1\}$ and $\pi \cup \{x/0\}$, or
   (b) Q's opponent owns x and Player Q loses Φ under either $\pi \cup \{x/1\}$ or $\pi \cup \{x/0\}$.

For example, consider a formula $\Phi = \forall u \exists e. u \wedge e$. Then:

- Player $\exists$ loses Φ under $\{u/0, e/0\}$, by subpart 1 of Definition 8.
- Player $\exists$ loses Φ under $\{u/0\}$, by subpart 2(a) of Definition 8.
- Player $\forall$ loses Φ under $\{u/1\}$, by subpart 2(b) of Definition 8.

Definition 9 *(Game-state specifier, match).* A **game-state specifier** is a pair $\langle L^{\text{now}}, L^{\text{fut}}\rangle$ consisting of two sets of literals, L^{now} and L^{fut}. We say that $\langle L^{\text{now}}, L^{\text{fut}}\rangle$ **matches** an assignment π iff:

1. for every literal ℓ in L^{now}, $\ell[\pi] = \text{true}$, and
2. for every literal ℓ in L^{fut}, $\ell[\pi] \neq \text{false}$ (i.e., either $\ell[\pi] = \text{true}$ or $\text{var}(\ell) \notin \text{vars}(\pi)$).

For example, $\langle\{u\}, \{e\}\rangle$ matches the assignments $\{u/1\}$ and $\{u/1, e/1\}$ (because both conditions in Definition 9 are satisfied), but does not match the empty assignment (because condition 1 fails) or $\{u/1, e/0\}$ (because condition 2 fails).

Note that, for any literal ℓ, if $\{\ell, \neg\ell\} \subseteq L^{\text{fut}}$, then $\langle L^{\text{now}}, L^{\text{fut}}\rangle$ matches an assignment π only if π does not assign ℓ. The intuition behind the names "L^{now}" and "L^{fut}" is as follows: Under the game formulation of QBF, the assignment π can be thought of as a state of the game, and π matches $\langle L^{\text{now}}, L^{\text{fut}}\rangle$ iff every literal in L^{now} is already true in the game and, for every literal ℓ in L^{fut}, it is possible that ℓ can be true in a future state of the game.

Definition 10 *(Game sequent).* The sequent $\langle L^{\text{now}}, L^{\text{fut}}\rangle \models (Q \text{ loses } \Phi)$ means "Player Q loses Φ under all assignments that match $\langle L^{\text{now}}, L^{\text{fut}}\rangle$."

As an example, let Φ be the following formula:

$$\forall u \exists e.\, (e \vee \neg u) \wedge (u \vee \neg e) \tag{5}$$

Note that sequent $\langle\{u\}, \{e\}\rangle \models (\forall \text{ loses } \Phi)$ holds true: in any assignment π that matches it, $\Phi[\pi] = \text{true}$. However, $\langle\{u\}, \varnothing\rangle \models (\forall \text{ loses } \Phi)$ does not hold true: it matches the assignment $\{u/1, e/0\}$, under which Player $\forall$ does not lose Φ.

For closed prenex instances, game sequents are isomorphic to the learned clauses/cubes; the differences are merely cosmetic. With the learning algorithm in [35], whenever a new game sequent is learned for a closed prenex instance, the literals owned by the winner all go in L^{fut}, and the literals owned by the loser go in L^{now}. The relationship between game-state sequents and learned clauses/cubes (for prenex instances) is as follows.

A learned clause $(\ell_1 \vee \ldots \vee \ell_n)$ is equivalent to the game sequent $\langle L^{\text{now}}, L^{\text{fut}}\rangle \models (\exists \text{ loses } \Phi_{\text{in}})$ where L^{now} contains all the existential literals from $\{\neg\ell_1, \ldots, \neg\ell_n\}$, and L^{fut} contains all the universal literals from $\{\neg\ell_1, \ldots, \neg\ell_n\}$. Note that the literals from the clause occur negated in the sequent.

Likewise, a learned cube $(\ell_1 \wedge \ldots \wedge \ell_n)$ is equivalent to the game sequent $\langle L^{\text{now}}, L^{\text{fut}}\rangle \models (\forall \text{ loses } \Phi_{\text{in}})$ where L^{now} contains all the universal literals from $\{\ell_1, \ldots, \ell_n\}$, and L^{fut} contains all the existential literals from $\{\ell_1, \ldots, \ell_n\}$. Unlike the case for clauses, the literals in a cube do not get negated for the corresponding sequent.

6.2. DPLL for 2-level QBF

Notation. Given an assignment π, let $prop(\pi)$ be the assignment produced by adding literals that would be forced in boolean constraint propagation (BCP) using the solver's sequent database. For example, if the input formula contains a subformula $(x \vee y)$ labeled by ghost variables $g^{\exists}$ and $g^{\forall}$, then $prop(\{x/1\})$ would contain x, $g^{\exists}$, and $g^{\forall}$.

To illustrate the basic idea of the CEGAR-in-DPLL technique, let Φ be a QBF of the form $\forall X \exists Y.\phi$. Let π_{cand} be an assignment to the variables in X such that $prop(\pi_{\text{cand}})$ does not match any sequent in the solver's sequent database. Let π_{cex} be a counterexample to π_{cand}; i.e., let π_{cex} be an assignment to the variables in Y such that $\phi[\pi_{\text{cand}} \cup \pi_{\text{cex}}] = \text{true}$. The goal of the CEGAR learning is to produce a set of sequents such that, if these sequents are added to the sequent database, then for every assignment π'_{cand} to X for which π_{cex} is a counterexample, some sequent in the database would match $prop(\pi'_{\text{cand}})$. This goal is accomplished as follows:

1. Substitute the assignment π_{cex} into ϕ, yielding the formula $\phi[\pi_{\text{cex}}]$.
2. Introduce ghost variables for any subformulas in $\phi[\pi_{\text{cex}}]$ that are not already labeled by ghost variables. Add sequents that relate these ghost variables to the subformulas that they represent, as described in Sec. 2.4.1 of [35].
3. Let $g_*^{\forall}$ be the universal ghost variable that labels the formula $\phi[\pi_{\text{cex}}]$.
4. Learn the new sequent $\langle\{g_*^{\forall}\}, \pi_{\text{cex}}\rangle \models (\forall \text{ loses } \Phi)$.

Consider an arbitrary assignment π'_{cand} to X to which π_{cex} is a counterexample. Then $\phi[\pi_{\text{cand}} \cup \pi_{\text{cex}}] = \text{true}$. To prove that $prop(\pi'_{\text{cand}})$ matches $\langle\{g_*^{\forall}\}, \pi_{\text{cex}}\rangle$, we must prove (1) $g_*^{\forall} \in prop(\pi'_{\text{cand}})$ and (2) $prop(\pi'_{\text{cand}})$ does not contain the negation of any literal in π_{cex}:

1. Since all the variables in the formula labeled by $g_*^{\forall}$ are assigned by π'_{cand}, it follows that either the variable $g_*^{\forall}$ or its negation must be a forced literal under π'_{cand}. And since $g_*^{\forall}$ labels $\phi[\pi_{\text{cex}}]$, and $(\phi[\pi_{\text{cex}}])[\pi_{\text{cand}}] = \text{true}$, it follows that the positive literal $g_*^{\forall}$ is forced, i.e., $g_*^{\forall} \in prop(\pi'_{\text{cand}})$.
2. A literal ℓ is forced under an assignment π only if the owner of ℓ is doomed to lose under $\pi \cup \{\ell/0\}$. Since Player $\exists$ owns π_{cex} and wins under $\pi'_{\text{cand}} \cup \pi_{\text{cex}}$, it follows that no literals from π_{cex} appear negated in $prop(\pi'_{\text{cand}})$.

Algorithm 9: DPLL algorithm with CEGAR learning.

```
global $\pi_{\text{cur}} \leftarrow \varnothing$
Function DPLL-Solve($\phi_{\text{in}}$)
begin
    while true do
        while ($\pi_{\text{cur}}$ doesn't match any database sequent) do
            DecideLit()
            Propagate()
        DPLL-Learn()
        if (learned seq has form $\langle \varnothing, L^{\text{fut}} \rangle \models (\Phi_{\text{in}} \Leftrightarrow \psi)$) then return $\psi$
        if (last decision literal is owned by winner) then
            CEGAR-Learn()
        Backtrack()
        Propagate()                                   // learned information forces a literal
```

For example, consider the formula $\Phi = \forall X \exists Y . \phi$ where ϕ is:

$$\phi = (\neg u_1 \vee \neg e_3) \wedge (\neg u_2 \vee \neg e_4) \wedge (u_1 \vee e_3) \wedge (u_2 \vee e_4) \tag{6}$$

Suppose that $\pi_{\text{cand}} = \{u_1/1, u_2/1\}$ and $\pi_{\text{cex}} = \{e_3/0, e_4/0\}$. Then $\phi[\pi_{\text{cex}}] = u_1 \wedge u_2$. Let $g_6^{\forall}$ be the universal ghost variable for $u_1 \wedge u_2$. The solver learns the sequent $\langle \{g_6^{\forall}\}, \{\neg e_3, \neg e_4\} \rangle \models (\forall \text{ loses } \Phi)$, as well as sequents relating $g_6^{\forall}$ to the subformula which it represents.

To add CEGAR learning to the DPLL-based solver GhostQ, we insert a call to a new CEGAR-learning procedure after standard DPLL learning, as shown in Algorithm 9. As shown in Algorithm 9, CEGAR learning is performed only if the last decision literal in π_{cur} is owned by the winner. (The case where the last decision literal is owned by the losing player corresponds to the conflicts that take place *within* the underlying SAT solver in RAReQS.)

6.3. DPLL for QBF with arbitrary quantification levels

Consider a QBF $Q_1 Z_1 \ldots . Q_n Z_n . \phi$. Suppose that the last decision literal belongs to the winner and is in the block Z_i. Then CEGAR learning would proceed as follows:

1. Let π_{cex} be a total assignment to the variables in Z_i. If a variable in Z_i is assigned by π_{cur}, it should have the same value in π_{cex}; if it does not appear in π_{cur}, it can be assigned an arbitrary value in π_{cex}.
2. Let *guard* be a subset of π_{cur} that assigns a subset of variables in $Z_1, \ldots, Z_{i-2}$. The choice of *guard* is heuristicly picked.
3. Let $Z'_{i+1}, \ldots, Z'_n$ be fresh variables corresponding to $Z_{i+1}, \ldots, Z_n$, respectively.
4. Let ϕ' be the result of substituting the assignment $\pi_{\text{cex}} \cup guard$ into ϕ and replacing all occurrences of variables in $Z_{i+1}, \ldots, Z_n$ with $Z'_{i+1}, \ldots, Z'_n$, respectively.
5. Introduce ghost variables for any formulas in ϕ' not already labeled by ghost variables. Add sequents that relate these ghost variables to the subformulas that they represent.
6. Let Q^* be $\bar{Q}_i$. Let $g_*^{Q^*}$ be the ghost variable that labels the formula ϕ' (if Q^* is $\forall$) or the negation of this ghost variable (if Q^* is $\exists$).
7. Learn the new sequent $\langle guard \cup \{g_*^{Q^*}\}, \pi_{\text{cex}} \rangle \models (Q^* \text{ loses } \Phi)$.

7. Experimental results

This section shows experimental evaluation of the presented algorithms. All experiments were carried out on Intel Xeon 5160 3 GHz machines, with 4 GB of memory.

7.1. Experimental results for 2-level QCNF

Two versions of AReQS were evaluated: one that does not use any heuristics (denoted *AReQS*) and the second that uses the heuristics described in Section 3.3 (denoted *AReQS-H*). The solvers were compared to the CDCL solver DepQBF [37] and the expansion-based solvers Nenofex [9] and Quantor [8].

A variety of benchmarks were chosen for the empirical evaluation. The sources for the benchmarks were: QBF library [38], QBF evaluation [39], and two well-known Σ_2^P and Π_2^P complete problems. From the QBF library [38] we chose the RobotsD2 benchmarks, from QBF evaluation the set of problems used in 2010 2QBF track. Since the family of RobotsD2 is disproportionally large, only a random sample of 150 instances was considered. Entailment in propositional circumscription (circ) is a well-known Π_2^P problem and instances from product configuration were used [40]. Implicate core (icore) is the problem of deciding for a given clause C, a constant k, and a CNF ϕ whether there exists a clause $C' \subseteq C$, s.t. $|C'| < k$

Table 1
Numbers of solved instances.

Family	AReQS	AReQS-H	DepQBF	Quantor	Nenofex
2QBF ’10 (65)	**52**	51	30	9	0
icore (75)	**39**	38	26	19	20
RobotsD2 (samp.) (98)	**98**	**98**	93	0	3
circ (38)	**38**	**38**	30	8	34
total	**227**	225	179	36	57

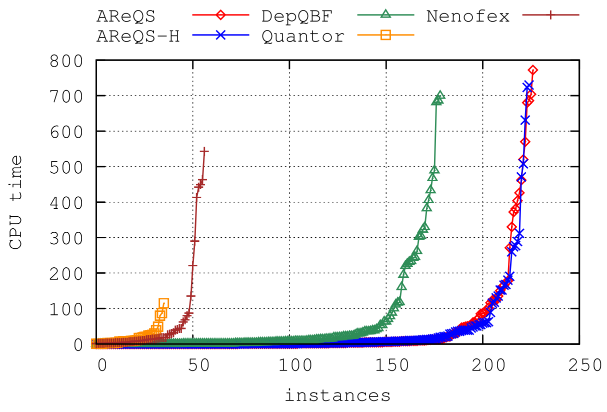
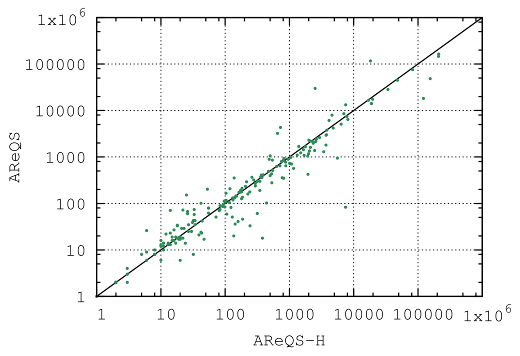
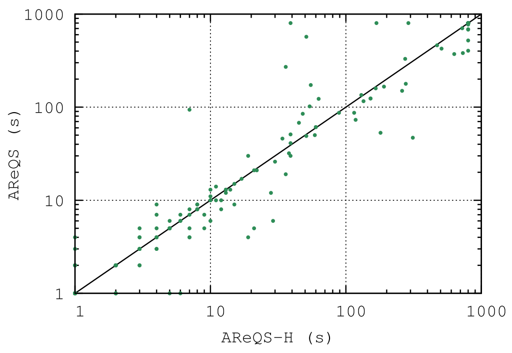


Fig. 4. Cactus plot for the overall results.



(a) Number of iterations comparison (log scale)



(b) Time comparison (log scale)

Fig. 5. Overview of the experimental results.

and $\phi \Rightarrow C'$; the problem is well known to be Σ_2^P-complete [41].[6] Only problems of the form ∀∃ were considered from the QBF library (this was true for all the problems in the 2QBF track of the QBF-Evaluation); the implicate core problem was directly generated in its negated form (again producing the ∀∃ form). All benchmarks were preprocessed by the preprocessor bloqqer [16] and instances solved by the preprocessor alone were excluded from further analysis. The experiments were obtained with a 800 s time limit and 2 GB memory limit.

Table 1 shows the number of solved instances for each set of benchmarks and solver. Fig. 4 shows a cactus plot for all of the instances. Both versions of AReQS dominate the results and are followed by DepQBF. The expansion-based solvers Nenofex and Quantor perform poorly.

The heuristics in AReQS-H do not seem to give a clear benefit. In fact, the heuristics implementation solves two instances fewer. Figs. 5(a) and 5(b) provide a more detailed view. Fig. 5(a) compares the number of iterations that the solver performed per instance. Fig. 5(b) compares the running times of the implementations. These figures do not show a clear advantage of either of the versions. However, it seems that the heuristics pay off more in harder instances, i.e., in instances where both of the solvers need a substantial amount of time or iterations. Hence, it would be interesting to explore a hybrid approach of invoking the heuristics. Another observation to be made is that there are instances where AReQS performs a large number of

[6] The problem is usually presented for an implicant rather than implicate, which is easily convertible to the implicate problem by negating the input formula.

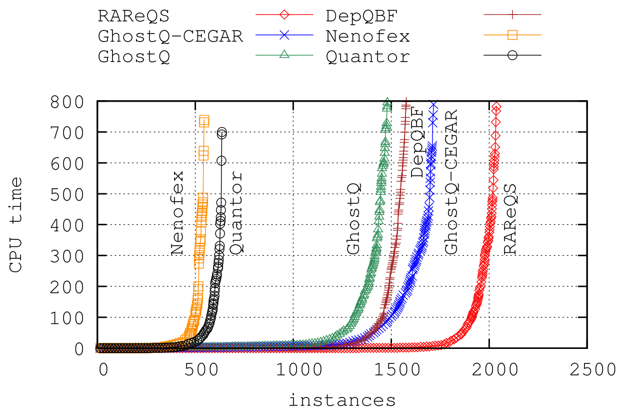


Fig. 6. Cactus plot of the overall results.

Table 2
N. of instances solved by RAReQS but not by a competing solver, and *vice versa*.

	GhostQ	GhostQ-CEGAR	DepQBF	Nenofex	Quantor
Only RAReQS	832	632	561	1512	1437
Only competitor	275	311	101	21	36

iterations (order of 10^6); this suggests that the complexity of the investigated problems is not the underlying SAT problems but the number of iterations overall.

7.2. Experimental results for PQCNF

Our objective was to analyze the effect of CEGAR on the different families of available benchmarks. Due to the large number of families in QBF-LIB [38], we have targeted families from *formal verification* and *planning* as two prominent applications of QBF. We have also included the family of benchmarks used for the 2012 evaluation.[7] Several large and hard families were randomly sampled (`terminator`, `tipfixpoint`, `Strat. Companies`, `traffic-controller`, `RobotsD3`); the area of planning contains four classes for robot planning, each counting 1000 instances with similar characteristics and thus only one of these classes was selected (`RobotsD3`).

The solvers were compared to `DepQBF` [37] and the expansion-based solvers Nenofex [9] and Quantor [8]. GhostQ was evaluated in two versions. One, the original version GhostQ [42,35] and second, with the CEGAR learning—GhostQ-CEGAR (see Section 6). The time limit was 800 s and the memory limit 2 GB.

All the instances were preprocessed by the preprocessor bloqqer [16] and instances solved by the preprocessor alone were excluded from further analysis. An exception was made for the family `Debug` where preprocessing turned out to be infeasible and the family was considered in its unpreprocessed form.

Unlike the other solvers, GhostQ's input format is not clause-based (QDIMACS) but it is circuit-based. To enable running GhostQ on the targeted instances, the solver was prepended with a reverse-engineering front-end. This reverse-engineering performs poorly on the bloqqer's output, since the original structure is "scrambled" by preprocessing. Hence, GhostQ was run directly on the instances without preprocessing. The other solvers were run on the preprocessed instances.

The relation between solving times and instances is presented by a cactus plot in Fig. 6; number of solved instances per family are shown in Table 3; a comparison of RAReQS with other solvers is presented in Table 2.

On the considered benchmarks, RAReQS solved the most instances, approximately 29% more than the solver `DepQBF`. RAReQS also turned out to be the best solver for most of the types of the considered instances. Table 2 further shows that for each of the other solvers, there is only a small portion of instances that the other solver can solve and RAReQS cannot.

In several families the addition of CEGAR learning to GhostQ worsened its performance. However, with the exception of `RobotsD3`, the performance was worse only slightly. Overall, GhostQ benefited from the additional CEGAR learning and in particular for certain families. A family worth noting is `irqlkeapclte`, where no instances were solved by any of the solvers except for GhostQ-CEGAR.

The usefulness of CEGAR was in particular demonstrated by the families `incrementer-enc.`, `conformant-planning`, `trafficlight-controller`, `Sorting-networks`, and `BMC` where RAReQS solved significantly more instances than the existing solvers and GhostQ-CEGAR improved significantly over GhostQ. Most notably, for `incrementer-encoder` only one instance was not solved by RAReQS, and for `RobotsD3`, `blackbox-01X-QBF` and `trafficlight-contr.` RAReQS solved *all* instances.

[7] Also used for the QBF Gallery 2014 http://qbf.satisfiability.org/gallery/.

Table 3
Number of instances solved within 800 seconds by each solver. "Lev" indicates the number of quantifier blocks (min–max) in the family of instances, post-bloqqer.

Family	Lev.	RAReQS	GhostQ	GhostQ-Cegar	DepQBF	Quantor	Nenofex
Adder (28)	3–7	**11**	2	2	7	5	9
BMC (85)	1–3	**73**	26	46	45	65	64
Blocks (7)	3–3	**7**	6	**7**	**7**	**7**	**7**
Counter (58)	1–125	30	15	11	23	**33**	15
Debug (38)	3–5	3	0	0	0	**24**	6
Gent-Rowley (205)	7–81	52	68	68	**82**	2	0
Lin. Bitvec. Rank. Fun. (60)	3–3	**9**	0	0	2	0	0
Ling (8)	1–3	**8**	6	**8**	**8**	**8**	**8**
Logn (2)	3–3	**2**	**2**	**2**	**2**	**2**	**2**
Mneimneh-Sakallah (163)	1–3	110	**150**	**150**	69	3	22
RankingFunctions (4)	2–2	**3**	0	0	**3**	0	0
RobotsD3 sample (100)	2–2	**100**	41	29	97	3	2
Scholl-Becker (55)	1–29	37	**43**	40	35	32	27
Sorting networks (84)	1–3	**72**	25	33	52	38	38
Strat. Comp. (samp.) (150)	1–2	107	12	20	**108**	18	12
blackbox-01X-QBF (320)	2–21	**320**	142	123	232	3	4
blackbox design (27)	5–9	**27**	**27**	**27**	18	0	0
circuits (63)	1–3	8	4	6	5	**9**	8
conformant plan. (23)	1–3	**17**	8	15	11	13	12
evader-pursuer (15)	5–19	**10**	9	9	**10**	2	2
fpu (6)	1–3	**6**	**6**	**6**	**6**	**6**	**6**
incrementer-encoder (484)	3–119	**483**	275	446	285	51	27
irqlkeapclte (45)	2–2	0	0	**44**	0	0	0
jmc quant (10)	3–3	2	0	0	**5**	0	1
jmc quant squaring (10)	3–9	0	0	0	**4**	0	1
terminator sample (150)	2–2	98	**104**	101	35	25	0
tipdiam (121)	1–3	55	**101**	93	56	21	14
tipfixpoint sample (150)	1–3	26	**130**	124	27	5	6
toilet all (136)	1–1	134	133	132	132	**135**	133
traff.-contr. (samp.) (100)	1–263	**100**	30	36	83	60	57
uclid (3)	4–6	0	**2**	**2**	0	0	0
Eval 2012 (276)	1–142	128	114	**137**	129	67	64
total		**2038**	1481	1717	1578	637	547

Table 4
Number of instances solved for non-cnf, non-prenex benchmarks.

	RAReQS	CirQit	GhostQ
semaphore (16)	**16**	**16**	**16**
ring (20)	**20**	**20**	**20**
possibility (120)	**106**	54	37
assertion (120)	**108**	56	47
counter (45)	43	42	**45**
dme (11)	6	**11**	**11**
consistency (10)	**10**	7	2
total (342)	**319**	206	178

CEGAR learning was also useful in GhostQ on the `Evaluation 2012` benchmarks, where GhostQ-CEGAR solved significantly more instances than other solvers—this was confirmed by the QBF Gallery 2014 [43].

7.3. Experimental results for non-prenex non-CNF QBF

A prototype of `RAReQS-NN` (Algorithm 6) was implemented in C++, supporting the *qpro format* [44], with the underlying SAT solver `minisat 2.2` [31]. The whole set of qpro benchmarks from the QBF-LIB was used. The solvers CirQit [10,45] and GhostQ were used for comparison. A timeout of 800 s was used along with a 2 GB memory limit. Table 4 shows how many instances were solved by each solver for each of the families of benchmarks. Fig. 7 provides a cactus plot for all the benchmarks. A more detailed overview can be found on the authors' website.[8] The solvers GhostQ and CirQit behave rather similarly on the considered benchmarks. `RAReQS-NN` turned out to be very successful on families assertion and possibility. In contrast, for the dme family, both CirQit and GhostQ perform significantly better.

[8] http://sat.inesc-id.pt/~mikolas/sw/areqs/qpro.html.

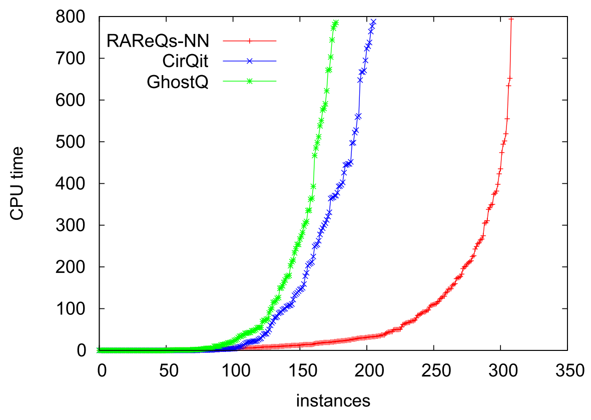


Fig. 7. Cactus plot of non-cnf, non-prenex benchmarks.

8. Related work

CEGAR has proven useful in number of areas, most notably in model checking [20] and SMT solving [32,33]; more recently it has been applied to handle quantification in SMT [46–48]. Special cases of QBF, with limited number of quantifiers, have been targeted by CEGAR: computing vertex eccentricity [49], and nonmonotonic reasoning [50,51].

A SAT solver was used in [52] to guide DPLL search of a QBF solver and to cut out unsatisfiable branches. A notion of abstraction was also used in QBF preprocessing [13]. This notion, however, differs from the one used in RAReQS as it means treating universally quantified variables as existentially quantified.

An important feature of RAReQS is the expansion of the given QBF into a propositional formula, which is then solved by a SAT solver. Expansion of quantifiers appear in number of solvers. In SAT and QBF preprocessing the technique of *variable elimination* [18,12] replaces all clauses containing a certain variable with all their possible resolvents on that variable. This is in fact existential expansion in disguise. Indeed, expanding $\exists x. (\phi_1 \vee x) \wedge (\phi_2 \vee \neg x)$ gives us $\phi_1 \vee \phi_2$ (assuming that ϕ_1, ϕ_2 do not contain x). If the original formula is in CNF, distributing $\vee$ gives us the result of all resolvents on x. Expansion of the universal quantifier is also used in preprocessing [11,53].

Several existing solvers tackle QBF solving by expansion. Most notably QUBOS [54], Quantor [8], sKizzo [7] and Nenofex [9]. Quantor and Nenofex apply expansion of quantifiers inside out and then invoke a SAT solver. While Quantor maintains a CNF matrix by variable elimination, Nenofex operates on an non-CNF representation. sKizzo considers a *Skolemization* of the original QBF, which in fact corresponds to expansion of all universal quantifiers and introducing fresh variable for the existential ones. Specialized techniques are used in order to avoid space explosion. Just as RAReQS uses multi-games, the above-mentioned solvers employ various techniques to mitigate the blowup of the expansion (besides preprocessing). QUBOS uses *miniscoping*, Quantor *tree-like prefixes*, and Nenofex uses *negation normal form*.

The way the expansion is carried out in RAReQS is significantly different. The expansion in RAReQS is *careful*. In the aforementioned solvers, once a variable is scheduled for expansion, both of its values are considered in the expansion.[9] In contrast, RAReQS only expands by an assignment to a block of variables at a time. This is an important factor for both time and space complexity. For large formulas, the traditional expansion-based solvers are bound to generate unwieldy formulas but the use of abstraction in RAReQS enables the solver to stop before this expansion is reached. This leads to generating easier formulas for the underlying SAT solver and dramatically mitigates the problems with memory blowup. Similar careful quantifier expansion also appears in SMT solving [48,47] and first order logic [55].

CNF search-based solvers can support non-CNF formulas by working on two representations of the formula: one in DNF and one in CNF [36,56]. But dedicated solvers also exists [10,35].

Since the publication of the 2-level QBF algorithm (AReQS), several similar approaches were used in specific domains [57–62].

8.1. Comparison to existing approaches

While the experimental results show that CEGAR-based solving gives us an approach that is often more efficient than the existing ones, it is natural to further explore how the algorithms differ, and, ask if there is a fundamental difference between them.

One approach how to compare solving algorithms is through *proof systems*. *Q-resolution* [28] is the proof system that corresponds to conflict/solution driven (CSDCL) solving, e.g. as in DepQBF. Expansion was formalized as the proof system $\forall$Exp + Res [63]; this corresponds directly to RAReQS. Further generalizations of this proof system exist [64]. It has been

[9] In Quantor, eliminating a single variable gives worst-case quadratic space increase.

shown that Q-resolution as well as its variants [65,66] are fundamentally different from expansion-based proof systems. More particularly, separations were shown in both directions, i.e., there are families of formulas where Q-resolution gives exponentially larger proofs than $\forall\text{Exp}+\text{Res}$ but also the other way around [67,68]. Hence, even from a theoretical point of view, CSDCL solving is fundamentally different from expansion-based solving.

In the remainder of this subsection, we present two example QBF formulas that showcase strengths of RAReQS. In the first example, we compare RAReQS to the full-expansion algorithm used in Quantor. In the second example, we compare RAReQS to a generic CSDCL solver.

8.1.1. *Comparison to Quantor on a simple example*

RAReQS may seem quite similar to expansion-based solvers like Quantor and Nenofex since RAReQS also performs expansion of quantifiers and subsequently calls a SAT solver. However, there is an important difference, RAReQS performs this expansion gradually and tests whether such expansion is already sufficient. Note that in the worst case, however, it may also perform a full expansion of the formula. The following example shows how gradual expansion as done by RAReQS can be beneficial.

Consider the QBF formula (where $\oplus$ is the XOR function)

$$\exists\underbrace{x_1\ldots x_N}_{X}\,\forall\underbrace{z_1\ldots z_N}_{Z}\,\exists\underbrace{s\,t_2\ldots t_N}_{T}.\,\phi$$

where

$$\phi = t_N \wedge \left(s \Leftrightarrow \bigvee\nolimits_{i=1\ldots N} x_i\right) \wedge \left(t_2 \Leftrightarrow ((z_1 \oplus z_2) \vee s)\right) \wedge \bigwedge\nolimits_{i=3\ldots N} \left(t_i \Leftrightarrow ((t_{i-1} \oplus z_i) \vee s)\right)$$

Note that ϕ can be efficiently converted to CNF without introducing any new variables. The algorithm used by Quantor [8] has difficulty with this problem. At each step, this algorithm can take one of two actions: eliminate an innermost existential variable by resolution, or eliminate a universal variable (of the second-to-innermost block) by expansion.

There are two end-game scenarios: (1) all the innermost T variables are eliminated, at which point all Z variables can be dropped from all clauses by universal reduction, and the result given to a SAT solver, or (2) all universal variables Z are expanded, at which point the formula is handed off to a SAT solver. Resolving on all the innermost T variables (without expanding any of the universal variables) requires representing $z_1 \oplus \ldots \oplus z_N$ in CNF (without introducing new variables), which requires space exponential in N. Now let us examine what happens when universal variables get expanded.

Consider what the formula looks like after variables $V_Z \subset Z$ have been expanded and variables $V_T \subset T$ have been resolved. Let $\text{Res}(\psi, V_T)$ denote the result of eliminating existential variables in V_T by resolution, as described in Section 3.1 of [8]. Then the formula has the following form:

$$\exists X.\forall(Z \smallsetminus V_Z).\ \bigwedge\nolimits_{\pi\in\mathcal{B}^{V_Z}} \exists(T \smallsetminus V_T).\,\text{Res}\big(\phi[\pi], V_T\big)$$

except that it must be converted to prenex form, which entails making $2^{|V_Z|}$ copies of the variables in $T \smallsetminus V_T$. Note that, for every assignment π to V_Z, the formula $\text{Res}(\phi[\pi], V_T)$ has an actual dependence on $Z \smallsetminus V_Z$ and $T \smallsetminus V_Z$ i.e., it does not evaluate to true or false or an expression involving only the X variables. Thus, the size of the formula is at least $O(2^{|V_Z|})$. So, eliminating all the universal variables would require space $O(2^N)$.

On the other hand, to eliminate all the innermost T variables, the algorithm must, as an intermediate step, construct a formula where $|V_T| = |T| - 1$. In this case, each $\text{Res}(\phi[\pi], V_T)$ must represent an XOR of at least $\frac{1}{2}(|V_T| - |V_Z|)$ variables (either $\{z_1, \ldots, z_i\} \smallsetminus V_Z$ or $\{z_{i+1}, \ldots, z_N\} \smallsetminus V_Z$, for some i), so it must have size $O(2^{(|V_T|-|V_Z|)/2})$, and there are $2^{|V_Z|}$ of them, so the total size is

$$O\big(2^{|V_Z|}\cdot 2^{(|V_T|-|V_Z|)/2}\big) = O\big(2^{(|V_T|+|V_Z|)/2}\big) \subseteq O\big(2^{N/2}\big)$$

So, regardless of whether all universal variables Z are eliminated or all innermost existential variables T are eliminated, the full-expansion algorithm requires time and space exponential in N.

In contrast to Quantor, RAReQS solves this QBF quickly. One possible run of the CEGAR algorithm proceeds as follows:

1. Suppose we pick the candidate $x_1 = \ldots = x_N = \text{false}$ for the outermost existential block. Under this assignment, ϕ simplifies to

   $$t_n \wedge (s \Leftrightarrow \text{false}) \wedge \big(t_2 \Leftrightarrow (z_1 \oplus z_2)\big) \wedge \bigwedge\nolimits_{i=3\ldots N}\big(t_i \Leftrightarrow (t_{i-1} \oplus z_i)\big)$$

   This formula can evaluate to true only if $t_n = z_1 \oplus z_2 \oplus \ldots \oplus z_N$ and $t_n = \text{true}$. Accordingly, any assignment to $z_1 \ldots z_N$ in which an even number of z variables are true is a winning move for the universal player and a counterexample to the existential candidate $x_1 = \ldots = x_N = \text{false}$.
2. Now the existential player must pick a candidate for the outermost block in which at least one x_i is true. Under this assignment, ϕ simplifies to

   $$t_n \wedge (s \Leftrightarrow \text{true}) \wedge \big(t_2 \Leftrightarrow \text{true}\big) \wedge \bigwedge\nolimits_{i=3\ldots N}\big(t_i \Leftrightarrow \text{true}\big)$$

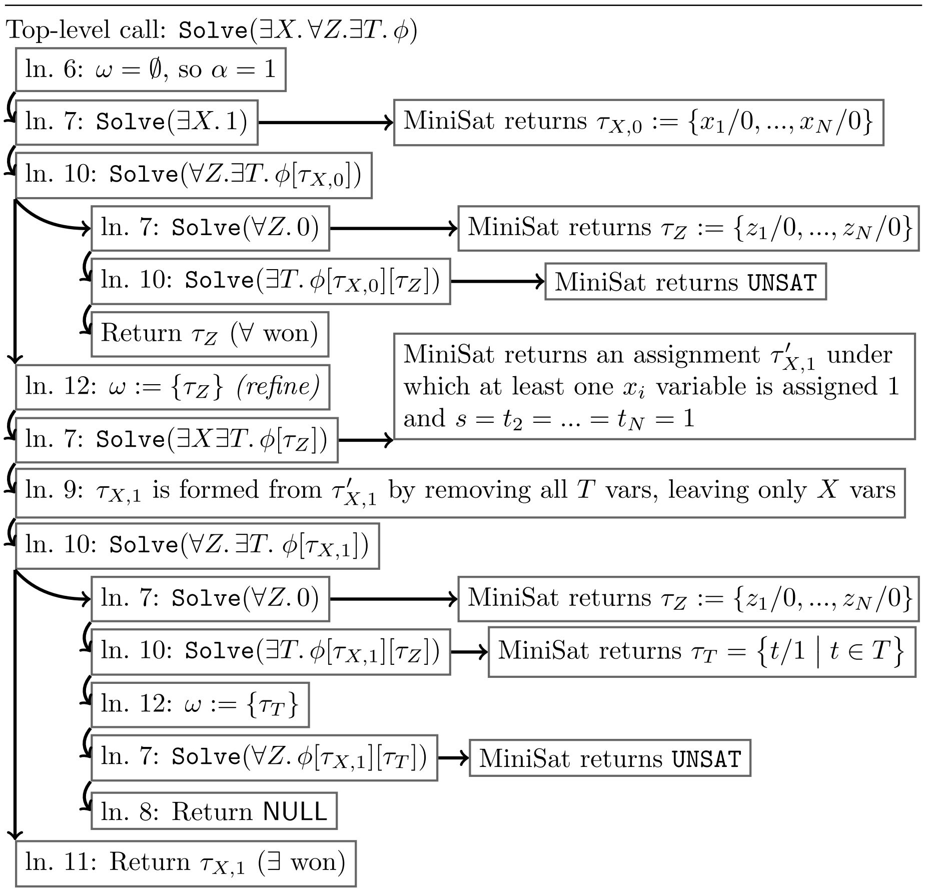

Top-level call: Solve($\exists X. \forall Z. \exists T. \phi$)
ln. 6: $\omega = \emptyset$, so $\alpha = 1$
ln. 7: Solve($\exists X. 1$)
MiniSat returns $\tau_{X,0} := \{x_1/0, ..., x_N/0\}$
ln. 10: Solve($\forall Z. \exists T. \phi[\tau_{X,0}]$)
ln. 7: Solve($\forall Z. 0$)
MiniSat returns $\tau_Z := \{z_1/0, ..., z_N/0\}$
ln. 10: Solve($\exists T. \phi[\tau_{X,0}][\tau_Z]$)
MiniSat returns UNSAT
Return τ_Z ($\forall$ won)
ln. 12: $\omega := \{\tau_Z\}$ *(refine)*
MiniSat returns an assignment $\tau'_{X,1}$ under which at least one x_i variable is assigned 1 and $s = t_2 = ... = t_N = 1$
ln. 7: Solve($\exists X \exists T. \phi[\tau_Z]$)
ln. 9: $\tau_{X,1}$ is formed from $\tau'_{X,1}$ by removing all T vars, leaving only X vars
ln. 10: Solve($\forall Z. \exists T. \phi[\tau_{X,1}]$)
ln. 7: Solve($\forall Z. 0$)
MiniSat returns $\tau_Z := \{z_1/0, ..., z_N/0\}$
ln. 10: Solve($\exists T. \phi[\tau_{X,1}][\tau_Z]$)
MiniSat returns $\tau_T = \{t/1 \mid t \in T\}$
ln. 12: $\omega := \{\tau_T\}$
ln. 7: Solve($\forall Z. \phi[\tau_{X,1}][\tau_T]$)
MiniSat returns UNSAT
ln. 8: Return NULL
ln. 11: Return $\tau_{X,1}$ ($\exists$ won)

Fig. 8. Walk-through of Algorithm 4. Indentation corresponds to depth in the call stack.

No universal variables occur in the above formula, so regardless of how the universal z_i variables are assigned, a winning move for the existential player is simply to assign true to all the variables in the innermost block.

Fig. 8 shows in greater detail how the recursive nature of Algorithm 4 (page 9) plays out for this example.

8.1.2. Comparison to CSDCL solver on a simple example

In contrast to RAReQS or Quantor, CSDCL solvers do not expand variables at all. As a consequence they can operate in polynomial space whereas expansion-based solvers operate in exponential space. The following example exemplifies that expansion can be beneficial ([67, Sec. 7] gives another example). Consider the QBF formula

$$\exists X \, \forall z \, \exists Y. \phi, \quad \text{where} \quad \phi = \mathit{hard}(X) \wedge \big(z \,?\, \mathit{easy}_1(Y) : \mathit{easy}_2(Y)\big)$$

where $\mathit{hard}(X)$ is a hard problem that evaluates to false (such as the pigeon-hole problem with n pigeons and $n-1$ holes) over the X variables and each $\mathit{easy}_i(Y)$ is an easy (but not trivial) problem over the Y variables that evaluates to false.

In a traditional CSDCL solver, all outermost existential variables (X) must be assigned a value before the solver can choose a decision variable from $\{z\}$ or Y. Furthermore, it is easy to see that no variables from $\{z\}$ or Y will get forced during boolean constraint propagation (BCP) by a decision to an X variable. Thus, a traditional CSDCL solver will not finish until it solves the hard problem.

In contrast, RAReQS, following Algorithm 4, quickly determines that the formula is false by solving the easy problem without solving the hard problem.

1. RAReQS picks an assignment to X (line 7, with $\omega = \emptyset$, so $\alpha = 1$).
2. RAReQS finds a counterexample for z, say $z = 0$.
3. RAReQS tries to find another candidate for X. Substituting $z = 0$ into ϕ yields $\mathit{hard}(X) \wedge \mathit{easy}_2(Y)$, which RAReQS gives to a SAT solver. The SAT solver is able to solve this by just solving the easy problem, and it quickly returns UNSAT.
4. Since the SAT solver returned UNSAT, there is no possible winning move for X, so RAReQS returns NULL, indicating that the universal player wins.

9. Conclusions and future work

This article studies applications of the CEGAR paradigm in the context of QBF solving. It introduces the algorithm RAReQS, which gradually expands the given formula into a propositional one and applies a SAT solver on it. RAReQS is a recursive extension of the 2QBF algorithm AReQS, which has since its original publication inspired specialized algorithms for number of problems in the second level of polynomial hierarchy.

In its workings, RAReQS is close to expansion-based solvers (e.g. Quantor, Nenofex) but with the important difference that the expansion is done step-by-step, driven by counterexamples. Thus, the solver builds an abstraction of the given formula by constructing a partial expansion. The downside of this approach is that if in the end a full expansion is needed, then RAReQS performs the same expansion as a traditional expansion-based solver but with the overhead of intermediate tests.

However, the approach has important advantages. Whenever there is no winning move for the partial expansion, then there is no winning move for the given formula. This enables RAReQS to quickly stop for formulas with no winning moves. For formulas for which there *is* a winning move, RAReQS only needs to build a strong-enough partial expansion whose winning moves are also likely to be winning moves for the given formula. The experimental results demonstrate the ability of RAReQS to avoid the inherent memory blowup of expansion solvers, and, that careful expansion outperforms a traditional DPLL-based approach on a large number of practical instances.

We have shown that abstraction-refinement as used in RAReQS is also applicable within DPLL solvers as an additional learning mechanism. This provides a more powerful learning technique than standard clause/cube learning, although it requires more memory. Experimental evaluation indicates that this type of learning is indeed useful for DPLL-based solvers.

In the future we plan to further develop our DPLL solver so that it supports the full range of CEGAR learning exploited by RAReQS and to investigate how to fine-tune this learning in order to mitigate the speed penalty for the cases where the learning provides little information over the traditional learning. This can not only be done by better engineering of the solver but also devising schemata that disable the learning once deemed too costly. Conversely, in the future we plan to integrate learning techniques into RAReQS. This is likely to be important in formulas with large number of quantification levels. Techniques used in other solvers should also be considered. In particular, more aggressive preprocessing as used in Quantor and techniques for finding commonalities in formulas used in Nenofex, and dependency detection as in DepQBF [37].

Acknowledgements

This work is partially supported by SFI PI grant BEACON (09/IN.1/I2618), CMU-Portugal grant AMOS (CMUP-EPB/TIC/0049/2013) and the FCT grant POLARIS (PTDC/EIA-CCO/123051/2010), and multiannual PIDDAC program funds PEst-OE/EEI/LA0021/2013.

References

[1] H. Kleine Büning, U. Bubeck, Theory of quantified Boolean formulas, in: [71], 2009, pp. 735–760.
[2] M. Benedetti, H. Mangassarian, QBF-based formal verification: experience and perspectives, J. Satisf. Boolean Model. Comput. 5 (1–4) (2008) 133–191.
[3] J. Rintanen, Asymptotically optimal encodings of conformant planning in QBF, in: AAAI Conference on Artificial Intelligence, AAAI Press, 2007, pp. 1045–1050.
[4] M. Schaefer, C. Umans, Completeness in the polynomial-time hierarchy: a compendium, SIGACT News 33 (3) (2002) 32–49.
[5] E. Giunchiglia, P. Marin, M. Narizzano, Reasoning with quantified Boolean formulas, in: [71], 2009, pp. 761–780.
[6] E. Giunchiglia, P. Marin, M. Narizzano, QuBE 7.0 system description, J. Satisf. Boolean Model. Comput. 7 (2010) 83–88.
[7] M. Benedetti, Evaluating QBFs via symbolic skolemization, in: International Conference on Logic for Programming, Artificial Intelligence, and Reasoning, LPAR, Springer, 2004, pp. 285–300.
[8] A. Biere, Resolve and expand, in: International Conference on Theory and Applications of Satisfiability Testing, SAT, 2004, pp. 238–246.
[9] F. Lonsing, A. Biere, Nenofex: expanding NNF for QBF solving, in: International Conference on Theory and Applications of Satisfiability Testing, SAT, Springer, 2008, pp. 196–210.
[10] A. Goultiaeva, F. Bacchus, Exploiting QBF duality on a circuit representation, in: AAAI Conference on Artificial Intelligence, AAAI Press, 2010, pp. 71–76.
[11] U. Bubeck, H. Kleine Büning, Bounded universal expansion for preprocessing QBF, in: International Conference on Theory and Applications of Satisfiability Testing, SAT, 2007, pp. 244–257.
[12] E. Giunchiglia, P. Marin, M. Narizzano, sQueezeBF: an effective preprocessor for QBFs based on equivalence reasoning, in: [69], 2010, pp. 85–98.
[13] F. Lonsing, A. Biere, Failed literal detection for QBF, in: [72], 2011, pp. 259–272.
[14] H. Samulowitz, J. Davies, F. Bacchus, Preprocessing QBF, in: International Conference on Principles and Practice of Constraint Programming, CP, Springer, 2006, pp. 514–529.
[15] H. Samulowitz, F. Bacchus, Binary clause reasoning in QBF, in: International Conference on Theory and Applications of Satisfiability Testing, SAT, Springer, 2006, pp. 353–367.
[16] A. Biere, F. Lonsing, M. Seidl, Blocked clause elimination for QBF, in: International Conference on Automated Deduction, CADE, 2011, pp. 101–115.
[17] L. Zhang, S. Malik, Conflict driven learning in a quantified Boolean satisfiability solver, in: International Conference on Computer-Aided Design, ICCAD, 2002, pp. 442–449.
[18] G. Pan, M.Y. Vardi, Symbolic decision procedures for QBF, in: International Conference on Principles and Practice of Constraint Programming, CP, Springer, 2004, pp. 453–467.
[19] M. Benedetti, sKizzo: a suite to evaluate and certify QBFs, in: International Conference on Automated Deduction, CADE, 2005, pp. 369–376.
[20] E.M. Clarke, O. Grumberg, S. Jha, Y. Lu, H. Veith, Counterexample-guided abstraction refinement for symbolic model checking, J. ACM 50 (5) (2003) 752–794.
[21] R. Letz, Lemma and model caching in decision procedures for quantified Boolean formulas, in: Automated Reasoning with Analytic Tableaux and Related Methods, TABLEAUX, 2002, pp. 160–175.

[22] M. Janota, J. Marques-Silva, Abstraction-based algorithm for 2QBF, in: [72], 2011, pp. 230–244.
[23] M. Janota, W. Klieber, J. Marques-Silva, E.M. Clarke, Solving QBF with counterexample guided refinement, in: International Conference on Theory and Applications of Satisfiability Testing, SAT, 2012, pp. 114–128.
[24] G.S. Tseitin, On the complexity of derivation in propositional calculus, Stud. Constr. Math. Math. Log. 2 (1970) 115–125.
[25] D.A. Plaisted, S. Greenbaum, A structure-preserving clause form translation, J. Symb. Comput. 2 (3) (1986) 293–304.
[26] C.H. Papadimitriou, Computational Complexity, Addison–Wesley, 1994.
[27] N. Narodytska, A. Legg, F. Bacchus, L. Ryzhyk, A. Walker, Solving games without controllable predecessor, in: Computer Aided Verification, CAV, Springer, 2014, pp. 533–540.
[28] H. Kleine Büning, M. Karpinski, A. Flögel, Resolution for quantified Boolean formulas, Inf. Comput. 117 (1) (1995) 12–18.
[29] C.M. Li, F. Manyà, MaxSAT, hard and soft constraints, in: [71], 2009, pp. 613–631.
[30] E.D. Rosa, E. Giunchiglia, M. Maratea, Solving satisfiability problems with preferences, Constraints 15 (4) (2010) 485–515.
[31] N. Eén, N. Sörensson, An extensible SAT-solver, in: [73], 2003, pp. 502–518.
[32] L.M. de Moura, H. Rueß, M. Sorea, Lazy theorem proving for bounded model checking over infinite domains, in: International Conference on Automated Deduction, CADE, Springer, 2002, pp. 438–455.
[33] C.W. Barrett, D.L. Dill, A. Stump, Checking satisfiability of first-order formulas by incremental translation to SAT, in: Computer Aided Verification, CAV, Springer-Verlag, 2002, pp. 236–249.
[34] M. Cadoli, A. Giovanardi, M. Schaerf, An algorithm to evaluate quantified Boolean formulae, in: National Conference on Artificial Intelligence, John Wiley & Sons Ltd, 1998, pp. 262–267.
[35] W. Klieber, Formal verification using quantified Boolean formulas (QBF), Ph.D. thesis, Carnegie Mellon University, 2014, http://www.cs.cmu.edu/~wklieber/thesis.pdf.
[36] L. Zhang, Solving QBF by combining conjunctive and disjunctive normal forms, in: AAAI Conference on Artificial Intelligence, AAAI Press, 2006, pp. 143–150.
[37] F. Lonsing, A. Biere, DepQBF: a dependency-aware QBF solver, J. Satisf. Boolean Model. Comput. 7 (2–3) (2010) 71–76.
[38] QBF-library, The quantified Boolean formulas satisfiability library, http://www.qbflib.org/, 2010.
[39] QBF-evaluation, QBF solver evaluation portal, http://www.qbflib.org/index_eval.php, 2010.
[40] M. Janota, G. Botterweck, R. Grigore, J. Marques-Silva, How to complete an interactive configuration process?, in: International Conference on Current Trends in Theory and Practice of Computer Science, SOFSEM, Springer, 2010, pp. 528–539.
[41] C. Umans, The minimum equivalent DNF problem and shortest implicants, J. Comput. Syst. Sci. 63 (4) (2001) 597–611.
[42] W. Klieber, S. Sapra, S. Gao, E.M. Clarke, A non-prenex, non-clausal QBF solver with game-state learning, in: [69], 2010, pp. 128–142.
[43] F. Lonsing, M. Seidl, A. Van Gelder, The QBF gallery: behind the scenes, CoRR, arXiv:1508.01045.
[44] M. Seidl, The qpro input format, available at http://qbf.satisfiability.org/gallery/qpro.pdf, 2009.
[45] A. Goultiaeva, F. Bacchus, Exploiting circuit representations in QBF solving, in: [69], 2010, pp. 333–339.
[46] C.M. Wintersteiger, Y. Hamadi, L.M. de Moura, Efficiently solving quantified bit-vector formulas, in: Formal Methods in Computer-Aided Design, FMCAD, IEEE, 2010, pp. 239–246.
[47] C.M. Wintersteiger, Y. Hamadi, L.M. de Moura, Efficiently solving quantified bit-vector formulas, Form. Methods Syst. Des. 42 (1) (2013) 3–23.
[48] D. Monniaux, Quantifier elimination by lazy model enumeration, in: Computer Aided Verification, CAV, 2010, pp. 585–599.
[49] M.N. Mneimneh, K.A. Sakallah, Computing vertex eccentricity in exponentially large graphs: QBF formulation and solution, in: [73], 2003, pp. 411–425.
[50] B. Browning, A. Remshagen, A SAT-based solver for Q-ALL SAT, in: ACM Southeast Regional Conference, ACM, 2006, pp. 30–33.
[51] M. Janota, R. Grigore, J. Marques-Silva, Counterexample guided abstraction refinement algorithm for propositional circumscription, in: European Conference on Logics in Artificial Intelligence, JELIA, 2010, pp. 195–207.
[52] H. Samulowitz, F. Bacchus, Using SAT in QBF, in: International Conference on Principles and Practice of Constraint Programming, CP, Springer, 2005, pp. 578–592.
[53] U. Bubeck, Model-based transformations for quantified Boolean formulas, Ph.D. thesis, University of Paderborn, 2010.
[54] A. Ayari, D.A. Basin, QUBOS: deciding quantified Boolean logic using propositional satisfiability solvers, in: Formal Methods in Computer-Aided Design, FMCAD, Springer, 2002, pp. 187–201.
[55] K. Korovin, Instantiation-based automated reasoning: from theory to practice, in: International Conference on Automated Deduction, CADE, Springer, 2009, pp. 163–166.
[56] A. Goultiaeva, M. Seidl, A. Biere, Bridging the gap between dual propagation and CNF-based QBF solving, in: Design, Automation & Test in Europe, DATE, EDA Consortium, 2013, pp. 811–814.
[57] W. Dvořák, M. Järvisalo, J.P. Wallner, S. Woltran, Complexity-sensitive decision procedures for abstract argumentation, in: International Conference on Principles of Knowledge Representation and Reasoning, KR, AAAI Press, 2012, pp. 54–64.
[58] H. Chen, M. Janota, J. Marques-Silva, QBF-based Boolean function bi-decomposition, in: Design, Automation & Test in Europe, DATE, IEEE, 2012, pp. 816–819.
[59] A. Morgenstern, M. Gesell, K. Schneider, Solving games using incremental induction, in: Integrated Formal Methods, IFM, Springer, 2013, pp. 177–191.
[60] C. Jordan, Ł. Kaiser, Experiments with reduction finding, in: [70], 2013, pp. 192–207.
[61] A. Ignatiev, M. Janota, J. Marques-Silva, Quantified maximum satisfiability: a core-guided approach, in: [70], 2013, pp. 250–266.
[62] S. Jo, T. Matsumoto, M. Fujita, SAT-based automatic rectification and debugging of combinational circuits with LUT insertions, in: Asian Test Symposium, IEEE Computer Society, 2012, pp. 19–24.
[63] M. Janota, J. Marques-Silva, On propositional QBF expansions and Q-resolution, in: [70], 2013, pp. 67–82.
[64] O. Beyersdorff, L. Chew, M. Janota, On unification of QBF resolution-based calculi, in: Mathematical Foundations of Computer Science, MFCS, Springer, 2014, pp. 81–93.
[65] A. Van Gelder, Contributions to the theory of practical quantified Boolean formula solving, in: International Conference on Principles and Practice of Constraint Programming, CP, Springer, 2012, pp. 647–663.
[66] V. Balabanov, J.-H.R. Jiang, Unified QBF certification and its applications, Form. Methods Syst. Des. 41 (1) (2012) 45–65.
[67] M. Janota, J. Marques-Silva, Expansion-based QBF solving versus Q-resolution, Theor. Comput. Sci. 577 (2015) 25–42.
[68] O. Beyersdorff, L. Chew, M. Janota, Proof complexity of resolution-based QBF calculi, in: International Symposium on Theoretical Aspects of Computer Science, STACS, 2015, pp. 76–89.
[69] SAT10, International Conference on Theory and Applications of Satisfiability Testing, SAT, Springer, 2010.
[70] SAT13, International Conference on Theory and Applications of Satisfiability Testing, SAT, Springer, 2013.
[71] Handbook of Satisfiability, vol. 185, IOS Press, 2009.
[72] SAT11, International Conference on Theory and Applications of Satisfiability Testing, SAT, Springer, 2011.
[73] SAT03, International Conference on Theory and Applications of Satisfiability Testing, SAT, Springer, 2003.