Circuit-based Search Space Pruning in QBF

Mikoláš Janota

IST/INESC-ID,
University of Lisbon, Portugal
mikolas.janota@gmail.com

Abstract. This paper describes the algorithm implemented in the QBF solver CQESTO, which has placed second in the non-CNF track of the last year's QBF competition. The algorithm is inspired by the CNF-based solver QESTO. Just as QESTO, CQESTO invokes a SAT solver in a black-box fashion. However, it directly operates on the circuit representation of the formula. The paper analyzes the individual operations that the solver performs.

1 Introduction

Since the indisputable success of SAT and SMT, research has been trying to push the frontiers of automated logic-based solving. Reasoning with quantifiers represents a nontrivial challenge. Indeed, even in the Boolean case adding quantifiers bumps the complexity class from NP-complete to PSPACE-complete. Yet, quantifiers enable modeling a number of interesting problems [3].

This paper aims at the advancement of solving with quantifiers in the Boolean domain (QBF). Using CNF as input causes intrinsic issues in QBF solving [1,15, 28]. Consequently, there have been efforts towards solvers operating directly on a non-clausal representation [2,8,12,18,25,28]. This line of research is supported by the circuit-like QBF format QCIR [16].

This paper presents the solver CQESTO, which reads in a circuit-like representation of the problem and keeps on solving *directly* on this representation. For each quantification level, the solver creates a propositional formula that determines the possible assignments to the variables of that particular level. If one of these formulas becomes unsatisfiable, a formula in one of the preceding levels is to be strengthened. *The main focus of this paper is the analysis of the operations that take place during this strengthening.*

CQESTO extends the family of solvers that repeatedly call a SAT solver (exponentially many times in the worst case). The solver RAReQS [10, 12, 13] delegates to the SAT solver partial expansions of the QBF. QESTO [14] and CAQE [23] are CNF siblings of CQESTO. QuAbS [25] is similar to CQESTO but it operates on a *literal abstraction*, while CQESTO operates *directly* on the circuit. The workings of CQESTO is also similar to algorithms developed for theories in SMT [4, 7, 24].

The principle contributions of the paper are: 1) Description of the algorithm of the solver CQESTO. 2) Analysis of the operations used in the circuit. 3) Linking these operations to related solvers.

The rest of the paper is organized as follows. Section 2 introduces concepts and notation used throughout the paper; Section 3 describes the CQESTO algorithm; Section 4 reports on experimental evaluation. Finally, Section 5 summarizes the paper and outlines directions for future work.

2 Preliminaries

Standard concepts from propositional logic are assumed. Propositional formulas are built from variables, negation ($\neg$), and conjunction ($\wedge$). For convenience we also consider the constants $0, 1$ representing false and true, respectively. The results immediately extend to other connectives, e.g., $(\phi \Rightarrow \psi) = \neg(\phi \wedge \neg\psi)$, $(\phi \vee \psi) = \neg(\neg\phi \wedge \neg\psi)$. A *literal* is either a variable or its negation. An *assignment* is a mapping from variables to $\{0, 1\}$. Assignments are represented as sets of literals, i.e., $\{x, \neg y\}$ corresponds to $\{x \mapsto 1, y \mapsto 0\}$. For a formula ϕ and an assignment σ, the expression $\phi|_\sigma$ denotes *substitution*, i.e., the simultaneous replacement of variables with their corresponding value. With some abuse of notation, we treat a set of formulas $\mathcal{I}$ and $\bigwedge_{\phi \in \mathcal{I}} \phi$ interchangeably. The paper makes use of the well-established notion of *subformula polarity*. Intuitively, the polarity of a subformula is determined by whether the number of negations above it is odd or even. Formally, we defined priority as follows.

Definition 1 (polarity [19]). *The following rules annotate each occurrence of a subformula of a formula α with its* polarity $\in \{+, -\}$.

$$\begin{array}{lll} \alpha^{+} & & \textit{top rule, } \alpha \textit{ is positive} \\ (\neg\phi)^{\pi} & \rightsquigarrow \neg\phi^{-\pi} & \textit{negation flips polarity} \\ (\phi \wedge \psi)^{\pi} & \rightsquigarrow (\phi^{\pi} \wedge \psi^{\pi}) & \textit{conjunction maintains polarity} \end{array}$$

Quantified Boolean Formulas (QBF). QBFs [17] extend propositional logic by quantifiers over Boolean variables. Any propositional formula ϕ is also a QBF with all variables *free*. If Φ is a QBF with a free variable x, the formulas $\exists x.\, \Phi$ and $\forall x.\, \Phi$ are QBFs with x *bound*, i.e. not free. Note that we disallow expressions such as $\exists x. \exists x.\, x$. Whenever possible, we write $\exists x_1 \ldots x_k$ instead of $\exists x_1 \ldots \exists x_k$; analogously for $\forall$. For a QBF $\Phi = \forall x.\, \Psi$ we say that x is *universal* in Φ and is *existential* in $\exists x.\, \Psi$. Analogously, a literal l is universal (resp. existential) if $\mathsf{var}(l)$ is universal (resp. existential). Semantically a QBF corresponds to a compact representation of a propositional formula. In particular, the formula $\forall x.\, \Psi$ is satisfied by the same truth assignments as $\Psi|_{\{\neg x\}} \wedge \Psi|_{\{x\}}$ and $\exists x.\, \Psi$ by $\Psi|_{\{\neg x\}} \vee \Psi|_{\{x\}}$. Since $\forall x \forall y.\, \Phi$ and $\forall y \forall x.\, \Phi$ are semantically equivalent, we allow writing $\forall X$ for a set of variables X; analogously for $\exists$. A QBF with no free variables is *false* (resp. *true*), iff it is semantically equivalent to the constant 0 (resp. 1).

A QBF is *closed* if it does not contain any free variables. A QBF is in *prenex form* if it is of the form $Q_1 X_1 \ldots Q_k X_k.\, \phi$, where $Q_i \in \{\exists, \forall\}$, $Q_i \neq Q_{i+1}$, and ϕ is propositional. The propositional part ϕ is called the *matrix* and the rest the *prefix*. For a variable $x \in X_i$ we say that x is at *level* i. Unless specified otherwise, QBFs are assumed to be closed and in prenex form.

Algorithm 1: QBF solving with circuit-based pruning

```
input  : $Q_1X_1 \ldots\ldots Q_nX_nQ_{n+1}X_{n+1}.\,\phi$, where $X_{n+1}$ empty, $Q_i \neq Q_{i+1}$
output : truth value
1  $\alpha_i \leftarrow 1$, for $i \in 1..n-1$          // minimalistic initialization
2  $\alpha_n \leftarrow (Q_n = \exists)\,?\,\phi \,:\, \neg\phi$
3  $\alpha_{n+1} \leftarrow (Q_n = \exists)\,?\,\neg\phi \,:\, \phi$
4  $i \leftarrow 1$
5  while true do                                    // invariant $1 \leq i \leq n+1$
6      $\mathcal{I} \leftarrow$ proj$(\alpha_i, \bigcup_{j\in 1..i-1} \sigma_j)$
7      $(\sigma_i, \mathcal{C}) \leftarrow$ SAT$(\mathcal{I}, \alpha_i)$
8      if $\sigma_i = \bot$ then
9          if $i \leq 2$ then return $Q_i = \forall$       // nowhere to backtrack
10         $\xi_f \leftarrow$ forget$(X_i, \neg\mathcal{C})$             // eliminate $X_i$
11         $\xi_s \leftarrow \xi_f|_{\sigma_{i-1}}$          // eliminate $X_{i-1}$ by substitution
12         $\alpha_{i-2} \leftarrow \alpha_{i-2} \wedge \xi_s$               // strengthen
13         $i \leftarrow i-2$                               // backtrack
14     else
15         $i \leftarrow i+1$                               // move on
```

QBF as Games. A closed and prenex QBF $Q_1X_1 \ldots Q_kX_k.\,\phi$, represents a two-player game, c.f. [9,18]. The existential player tries to make the matrix true and conversely the universal player tries to make it false. Each player assigns a value only to a variable that belongs to the player and can only assign a variable once all preceding variables have already been assigned. Hence the two players assign values to variables following the order of the prefix alternating on a quantifier. A *play* is a sequence of assignments $\sigma_1, \ldots, \sigma_n$ where σ_i is an assignment to X_i. Within a play, the i^{th} assignment is referred to as the i^{th} *move*. The i^{th} move belongs to player Q_i. A QBF Φ is true iff there exists a winning strategy for $\exists$; it is false iff there exists a winning strategy for $\forall$. The game semantics enables treating $\exists$ and $\forall$ symmetrically, i.e. we are concerned with deciding which player has a winning strategy.

3 CQESTO Algorithm

The algorithm decides a closed, prenex QBF of n quantification levels. For the sake of uniformity we add a quantification level $n+1$ with no variables belonging to the player Q_{n-1}. So the formula being solved is $Q_1X_1 \ldots Q_nX_nQ_{n+1}X_{n+1}.\,\phi$, where X_{n+1} is empty and $Q_n \neq Q_{n+1}$. Like so it is guaranteed that eventually either of the player must lose as the play progresses, i.e. there's no need for handling especially a play where all blocks up till Q_n have value—if this happens, Q_{n+1} loses.

The algorithm's pseudocode is presented as Algorithm 1 and its overall intuition is as follows. For each quantification level i there is a propositional formula α_i, which constrains the moves of player Q_i. The algorithm builds assignments $\sigma_1, \ldots, \sigma_k$ so that each σ_i represents the i^{th} move of player Q_i. A SAT solver is used to calculate a new σ_i from α_i. Backtracking occurs when α_i becomes unsatisfiable under the current assignments $\sigma_1, \ldots, \sigma_{i-1}$. Upon backtracking, player Q_i needs to change some moves that he had made earlier. Hence, the algorithm strengthens α_{i-2} and continues from there.[1] Note that if $i = 1$, the union $\bigcup_{j \in 1..i-1} \sigma_j$ is empty and the SAT call is on α_1 with empty assumptions.

The algorithm observes the following invariants regarding the constraints α_i.

Invariant 1 (syntactic) *Each α_i only contains variables $X_1 \cup \cdots \cup X_i$.*

Invariant 2 (semantic) *If player Q_i violates α_i, the player is bound to lose. More formally, if for a partial play $\sigma = \sigma_1, \ldots, \sigma_i$ it holds that $\sigma \vDash \neg\alpha_i$ then there is a winning strategy for the opponent from that position.*

The invariants are established upon initialization by setting all α_i to true except for α_n and α_{n+1}. The constraint α_n is set to the matrix or its negation depending on whether Q_i is existential or universal. The constraint α_{n+1} is set analogously. Note that since $\alpha_{n+1} = \neg\alpha_n$, once α_n is satisfied by $\sigma_1 \cup \cdots \cup \sigma_n$, the constraint α_{i+1} is immediately unsatisfiable since there are no further variables.

The algorithm uses several auxiliary functions. The function `SAT` models a SAT call on propositional formulas. The function `proj` is used to propagate information into the current α_i while the solver is moving forward. The function `forget` enables the solver to strengthen previous restrictions upon backtracking. Let us look at these mechanisms in turn.

3.1 Projection (`proj`)

The function $\texttt{proj}(\alpha_i, \sigma)$ produces a set of formulas $\mathcal{I}$ implied by the assignment $\sigma = \sigma_1 \cup \ldots \cup \sigma_{i-1}$ in α_i. The motivation for $\mathcal{I}$ is akin to the one for 1-UIP [20]. The set $\mathcal{I}$ may be envisioned as a cut in the circuit representing the formula α_i. In the context of the algorithm, `proj` enables generalizing the concrete variable assignment to subformulas. Rather than finding the move σ_i by satisfying $\alpha_i \wedge \bigwedge_{j \in 1..i-1} \sigma_j$, it must satisfy $\alpha_i \wedge \mathcal{I}$. Upon backtracking, $\mathcal{I}$ is used to strengthen α_{i-2}. This gives a better strengthening than a particular assignment.

As a motivational example, consider the formula $\exists xy \forall u \exists z.\, (x \vee y) \Rightarrow (z \wedge \neg z)$ and the assignment $\{x, \neg y\}$. In this case, the function `proj` returns $(x \vee y)$ because it is implied by the assignment and keeps forcing a contradiction. This yields the SAT call on $((x \vee y) \Rightarrow (z \wedge \neg z)) \wedge (x \vee y)$. The formula is unsatisfiable and the reason is that $(x \vee y)$ is true. This lets us conclude that at the first level, $\neg(x \vee y)$ must be true—the concrete assignment to x and y is not important.

The function `proj` operates in two phases, first it *propagates* the assignment σ in α_i and then it collects the most general sub-formulas of α_i propagated by σ. To formalize the definition we introduce an auxiliary concept of propagation $\sigma \vdash_p \phi$ meaning that ϕ follows from σ by propagation.

[1] The implementation enables jumping across multiple levels by backtracking to the maximum level of variables in the core belonging to Q_i.

Definition 2 **($\vdash_p$).** *For an assignment σ and formula ϕ, the relation $\sigma \vdash_p \phi$ is defined according to the following rules.*

$$\begin{array}{ll} \sigma \vdash_p 1 & \sigma \vdash_p \psi \wedge \phi, \textit{ if } \sigma \vdash_p \psi \textit{ and } \sigma \vdash_p \phi \\ \sigma \vdash_p l, \textit{ if } l \in \sigma & \sigma \vdash_p \neg(\psi \wedge \phi), \textit{ if } \sigma \vdash_p \neg\psi \textit{ or } \sigma \vdash_p \neg\phi \end{array}$$

The function `proj` operates recursively on subformulas. It first checks if a subformula or its negation is inferred by propagation. If so, it immediately returns the given subformula or its negation, respectively. Otherwise, it dives into the subformula's structure. Subformulas unaffected by the assignment are ignored. The definition follows (see also Examples 1–3).

Definition 3 **(`proj`).** *For a formula ϕ and assignment σ, `proj`(ϕ, σ) is defined by the following equalities.*

$$\begin{array}{ll} \texttt{proj}(\phi, \sigma) = \{\phi\} & \textit{if } \sigma \vdash_p \phi \\ \texttt{proj}(\phi, \sigma) = \{\neg\phi\} & \textit{if } \sigma \vdash_p \neg\phi \\ \texttt{proj}(\psi \wedge \phi, \sigma) = \texttt{proj}(\psi, \sigma) \cup \texttt{proj}(\phi, \sigma) & \textit{if above does not apply} \\ \texttt{proj}(\neg\psi, \sigma) = \texttt{proj}(\psi, \sigma) & \textit{if above does not apply} \\ \texttt{proj}(l, \sigma) = \emptyset & \textit{if above does not apply} \end{array}$$

Note that `proj` is well defined also for an empty $\sigma = \emptyset$. Since we only have $\emptyset \vdash_p 1$ the projection `proj`$(\phi, \emptyset)$ will give the empty set, except for the special cases `proj`$(1, \emptyset) =$ `proj`$(0, \emptyset) = \{1\}$.

3.2 SAT calls

SAT calls are used to obtain a move σ_i at position i in a straightforward fashion (line 7). If the SAT calls deem the query unsatisfiable, it provides a core that is used to inform backtracking. In a call `SAT`$(\mathcal{I}, \alpha_i)$, $\mathcal{I}$ is a set of propositional formulas modeling assumptions. The function returns a pair $(\sigma_i, \mathcal{C})$, where σ_i is a satisfying assignment to $\alpha_i \wedge \mathcal{I}$ if such exists and it is $\perp$ otherwise. If there is no satisfying assignment, $\mathcal{C} \subseteq \mathcal{I}$ is a *core*, i.e. $\phi \wedge \mathcal{C}$ is also unsatisfiable. Since modern SAT solvers typically only accept formulas in CNF, standard translation from formulas to CNF may be used via fresh variables [26].

3.3 Backtracking

The backtracking mechanism is triggered once the SAT call deems $\alpha_i \wedge \mathcal{I}$ unsatisfiable. Only a core $\mathcal{C} \subseteq \mathcal{I}$ is used, which gives a stronger constraint than using the whole of $\mathcal{I}$. The SAT call guarantees that $\alpha_i \wedge \mathcal{C}$ is unsatisfiable—and therefore losing for player Q_i. The objective is to derive a strengthening for α_{i-2}. *To that end we remove the sets of variables X_i and X_{i-1} from the core $\mathcal{C}$.*

Variables X_{i-1} are removed by substituting the opponent's move σ_{i-1}. The intuition is that the opponent can always play that same move σ_{i-1} in the future. In another words, player Q_i must account for *any* move of the opponent and in this case Q_i prepares for σ_{i-1}. This is best illustrated by an example that already does not contain any of the variables X_i — so we only need to worry about X_{i-1}.

Example 1. Consider $\exists x_1 x_2 \forall y \exists z.\ (x_1 \wedge z) \wedge (x_2 \vee y)$ with $\sigma_1 = \{x_1, \neg x_2\}$, $\sigma_2 = \{\neg y\}$. Propagation gives $\sigma_1 \cup \sigma_2 \vdash_p x_1$ and $\sigma_1 \cup \sigma_2 \vdash_p \neg(x_2 \vee y)$. SAT gives the core $\mathcal{C} = \neg(x_2 \vee y)$. Negating the core and substituting σ_2 gives $\xi_f = (x_2 \vee y)|_{\{\neg y\}} = x_2$ leading to the strengthening $\alpha_1 \leftarrow \alpha_1 \wedge x_2$.

The removal of the variables X_i relies on their polarity (see Definition 1). Each positive occurrence of a variable is replaced by 1 and each negative occurrence by 0. This operation guarantees that the resulting formula is weaker than the derived core (see Lemma 2 for justification).

Definition 4 (`forget`). *For a set of variables X and a formula ϕ, the transformation* `forget`(X, ϕ) *is defined as follows. The definition uses an auxiliary function* `pol`(ψ, X, c) *where ψ is a formula and $c \in \{0, 1\}$ a constant. The constant c is determined by the polarity of ψ within ϕ (see Definition 1). If ψ is annotated positively (ψ^+), c is 1; if ψ is annotated negatively (ψ^-), c is 0.*

$$\begin{array}{ll}
\texttt{forget}(X, \phi) = \texttt{pol}(\phi, X, 1) & \\
\texttt{pol}(x, X, c) = c, \text{ if } x \in X & \texttt{pol}(\phi \wedge \psi, X, c) = \texttt{pol}(\phi, X, c) \wedge \texttt{pol}(\psi, X, c) \\
\texttt{pol}(x, X, c) = x, \text{ if } x \notin X & \texttt{pol}(\neg\phi, X, c) = \neg\texttt{pol}(\phi, X, \neg c)
\end{array}$$

Example 2. Consider $\exists x_1 x_2 \forall y \exists z.\ (x_1 \wedge z) \wedge (x_2 \vee y)$ and $\sigma_1 = \{\neg x_1, x_2\}$, $\sigma_2 = \{\neg y\}$. By propagation obtain $\sigma_1 \cup \sigma_2 \vdash_p \neg(x_1 \wedge z)$ and $\sigma_1 \cup \sigma_2 \vdash_p (x_2 \vee y)$. Yielding core $\mathcal{C} = \neg(x_1 \wedge z)$. Negating the core gives $(x_1 \wedge z)$, applying `forget`$(\{z\}, (x_1 \wedge z)) = x_1$. Hence we obtain the strengthening step $\alpha_1 \leftarrow \alpha_1 \wedge x_1$.

We conclude by an example where both X_i, X_{i-1} are removed at the same time.

Example 3. $\exists x_1 x_2 x_3 \forall y \exists z w.\ \big((x_1 \wedge x_2 \vee w) \Rightarrow \neg z\big) \wedge \big((x_3 \wedge y \vee \neg w) \Rightarrow z\big)$ for $\sigma_1 = \{x_1, x_2, x_3\}$, $\sigma_2 = \{y\}$ obtain $\sigma_1 \cup \sigma_2 \vdash_p (x_1 \wedge x_2 \vee w)$ and $\sigma_1 \cup \sigma_2 \vdash_p (x_3 \wedge y \vee \neg w)$. SAT giving the core $\mathcal{C} = \{(x_1 \wedge x_2 \vee w), (x_3 \wedge y \vee \neg w)\}$. Negate: $\neg(x_1 \wedge x_2 \vee w) \vee \neg(x_3 \wedge y \vee \neg w)$; apply `forget`: $\neg(x_1 \wedge x_2) \vee \neg(x_3 \wedge y)$; substitute $\{y\}$: $\neg(x_1 \wedge x_2) \vee \neg x_3$.

3.4 Discussion

CQESTO hinges on two operations: projection and forgetting. Arguably, any backtracking QBF solver must perform these operations in some form, while observing the properties outlined in the following section (Section 3.5). Here I remark that the implementation of CQESTO enables deviations from the current presentation. In particular, Invariant 1 may not be strictly observed: upon initialization all α_i are initialized with the original (negated) matrix. Like so, downstream variables also appear in α_i.

Several QBF solvers are characterized by repeated SAT calls. RAReQS [10, 12,13] performs a heavy-handed expansion of quantifiers requiring recursive calls: this may turn unwieldy in formulas with many quantification levels. The operation `forget` in RAReQS corresponds to the creation of new copies of the variables in the refinement. QELL [27] can be seen as a variant of RAReQS where variables are removed by greedy elimination.

Both QESTO [14] and CAQE [23] can be seen as specializations of CQESTO for CNF input. A similar approach has also been used in SMT [4].

QuAbS [25] is similar to CQESTO but with some important differences. Conceptually, CQESTO *directly* works on the given circuit. In contrast, QuAbS encodes the circuit into clauses and then it operates on the literals representing subformulas. The representing literals are effectively Tseitin variables (accommodating for semantics of quantification).[2] In this sense, CQESTO is more flexible because QuAbS loses the information about the circuit upon translation to clauses. Observe that for instance that Examples 1 and 2 operate on the same formula but `proj` gives different sub-formulas.

One important consequence of this flexibility is that CQESTO calls the SAT solver with fewer assumptions. As an example consider a sub-circuit ϕ that is set to 1 by propagation of values on previous quantification levels. Further, there are some sub-circuits of ϕ, also set to 1 by previous levels, let's say $\psi_1, \ldots, \psi_k$. To communicate to the current level that these are already set to true, QuAbS invokes the SAT solver with the assumptions $t_\phi, t_{\psi_1}, \ldots, t_{\psi_k}$, where t_γ is the representing literal. Consequently, in QuAbS, any of these assumptions may appear in the core. However, t_ϕ is the more desirable core because it "covers" its sub-circuits. In CQESTO, it is guaranteed to obtain such core because only ϕ will be using the assumptions (thanks to the function `proj`).

3.5 Correctness and Termination

This section shows correctness and termination by showing specific properties of the used operations. We begin by a lemma that intuitively shows that $(\alpha_i \wedge \sigma)$ is roughly the same as $(\alpha_i \wedge \texttt{proj}(\alpha_i, \sigma))$. This is relevant to the SAT call on line 7.

Lemma 1. *For* $\sigma = \bigcup_{j \in 1..i-1} \sigma_j$ *and* $\mathcal{I} = \texttt{proj}(\alpha_i, \sigma)$ *it holds that*

1. $\sigma \Rightarrow \mathcal{I}$
2. *Models restricted to* X_i *of* $\alpha_i \wedge \sigma$ *and* $\alpha_i \wedge \mathcal{I}$ *are the same.*

Proof (sketch). (1) By induction on expression depth. If for a literal l, it holds that $\sigma \vdash_p l$, then by Definition 2 also $l \in \sigma$ and therefore $\sigma \Rightarrow l$. For composite expressions, the implication holds by standard semantics of $\wedge$ and $\neg$.

(2) The models restricted to X_i of $\alpha_i \wedge \sigma$ are the same as of $\alpha_i|_\sigma$. Hence, instead of conjoining σ to α_i we imagine we substitute it into α_i and then apply standard simplification, e.g. $0 \wedge \phi = 0$. This results into the same formula as if we substituted directly 1 for $\phi \in \mathcal{I}$ with and 0 for $\neg\phi \in \mathcal{I}$. E.g. for $\sigma = \{x, y\}$ and $\alpha_i = (x \vee z) \wedge (y \vee q) \wedge o$, we obtain $\sigma \vdash_p (x \vee z)$ and $\sigma \vdash_p (y \vee q)$, and $\alpha_i|_\sigma$ gives o, which is equivalent replacing $(x \vee z)$ and $(x \vee z)$ with 1.

We continue by inspecting the operation $\texttt{forget}(X_i, \psi)$, important in abstraction strengthening. We show that the operation is a weakening of ψ, i.e. it

[2] This was also done similarly in Z3 when implementing [4].

does not rule out permissible moves. At the same time, however, we need to show that the result is not too weak. In particular, that the performed strengthening on lines 10–12 does not allow repeating a play that was once already lost.

Lemma 2. *Let ψ be a formula, $\sigma = \bigcup_{j\in 1..i-1} \sigma_j$ and $\mathcal{C} \subseteq \texttt{proj}(\alpha_i, \sigma)$ s.t. $\alpha_i \wedge \mathcal{C}$ is unsatisfiable.*

1. $\psi \Rightarrow \texttt{forget}(X_i, \psi)$
2. $\sigma \wedge \texttt{forget}(X_i, \neg\mathcal{C})$ *is unsatisfiable*
3. $\big(\sigma_1 \wedge \cdots \wedge \sigma_{i-2} \wedge \texttt{forget}(X_i, \neg\mathcal{C})\big)|_{\sigma_{i-1}}$ *is unsatisfiable*

Proof (sketch). (1) A positive occurrence of a formula with a weaker one, or replacing a negative occurrence of a formula with a stronger one leads to a weaker formula [19, The Polarity Proposition]. The operation `forget` is a special case of this because a positive occurrence of a variable is replaced by 1 (trivially weaker) and a negative occurrence by 0 (trivially stronger).
(2) Since the elements of the core $\mathcal{C}$ must have been obtained by `proj` (see ln. 7), we have $\sigma \vdash_p \phi$ for $\phi \in \mathcal{C}$ where all variables X_i are unassigned in σ. Hence, replacing the X_i variables with arbitrary expressions preserves the $\vdash_p$ relation, e.g. $\{x\} \vdash_p (x \vee z)$ but also $\{x\} \vdash_p (x \vee 0)$. Bullet (3) is a immediate consequence of (2).

Lemma 3. *Algorithm 1 preserves Invariants 1 and 2.*

Proof (sketch). Invariant 1 is trivially satisfied upon initialization and is preserved by backtracking. Invariant 2 is trivially satisfied upon initialization.

Since $\mathcal{C}$ is a core, $\alpha_i \wedge \mathcal{C}$ is unsatisfiable and therefore $\alpha_i \Rightarrow \neg\mathcal{C}$. This means that player Q_i must satisfy $\neg\mathcal{C}$ because he must satisfy α_i. The operation `forget` is a weakening (Lemma 2(1)) and therefore the player also must satisfy $\texttt{forget}(\neg\mathcal{C})$. Since the opponent can always decide to play σ_{i-1}, player Q_i must also satisfy $\big(\sigma_1 \wedge \cdots \wedge \sigma_{i-2} \wedge \texttt{forget}(X_i, \neg\mathcal{C})\big)|_{\sigma_{i-1}}$ at level $i-2$. Therefore the backtracking operation preserves the invariant.

Theorem 1. *The algorithm is correct, i.e. it returns the validity of the formula.*

Proof (sketch). The algorithm terminates only if α_i becomes unsatisfiable for $i \in 1..2$. From Invariant 2, the opponent of Q_i has a winning strategy for whatever move Q_i plays. Since Q_i has no previous moves to alter, there's a winning strategy for the opponent for the whole game. The algorithm returns true iff the losing player is $\forall$, i.e. iff there is a winning strategy for $\exists$.

Theorem 2. *The algorithm is terminating.*

Proof (sketch). We show that if the solver backtracks upon an assignment $\sigma = \bigcup_{j\in 1..i-1} \sigma_j$, the same assignment will not appear again. For contradiction let us assume that σ appears in a future run. This means that σ_{i-2} was obtained from $\texttt{SAT}(\mathcal{I}, \alpha_{i-2})$ with $\mathcal{I} = \texttt{proj}(\alpha_{i-2}, \bigcup_{j\in 1..i-3} \sigma_j)$. From Lemma 2(3) we have that $\bigcup_{j\in 1..i-2} \sigma_j$ is not a model of α_{i-2}. From Lemma 1(2) $\alpha_{i-2} \wedge \mathcal{I}$ have the same models as $\alpha_{i-2} \wedge \bigcup_{j\in 1..i-3} \sigma_j$, which gives a contradiction.

Solver	Solved (320)
QFUN	**118**
CQESTO	112
QuAbS	103
GQ	87
Qute	83

Table 1: Result summary.

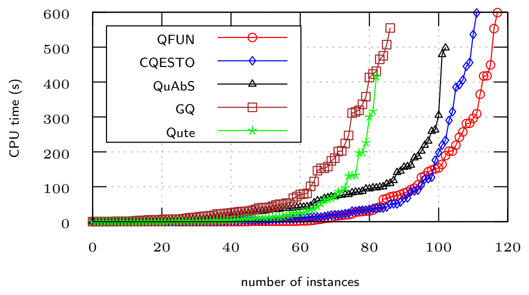

Fig. 1: Cactus plot. A point at (x, y) means that the solver solved x instances each within y sec.

4 Experimental Evaluation

The prototype CQESTO was implemented in C++ where logical gates are hash-consed as to avoid redundant sub-circuits. SAT calls are delegated to minisat 2.2 [6]. It differs from the Algorithm 1 by starting with stronger α_i: An α_i is initialized by ϕ where opponent's moves are fixed to a constant value.

CQESTO is compared to the solvers QFUN [11], QuAbS [25], Qute [21], and GhostQ [18] on the QBF Eval '17 instances [22] on Intel Xeon E5-2630 2.60GHz with 64GB memory; the limits were set to 32GB and 600s.

The results are summarized in Table 1 and Figure 1; detailed results can be found online [5]. There's a clear division between SAT-based solvers (QFUN, CQESTO, QuAbS) and resolution-based solvers (GhostQ, Qute). QFUN is in the lead closely followed by CQESTO. QuAbS is only 9 instances behind CQESTO but the cactus plot shows a notable slow-down early on. I remark that detailed inspection reveals that Qute fares much better on instances with high number of quantification levels. It is a subject of future work to better understand if the difference between QuAbS and CQESTO is due to implementation or the different calculation of strengthening.

5 Summary and Future Work

This paper contributes to the understanding of QBF solving by studying the algorithm CQESTO, which is characterized by maintaining propositional restrictions, in a circuit form, on the possible moves of the corresponding player at each quantification level. *Projection* is used to propagate the current assignment into the circuit. Once the SAT solver provides a contradiction at the current level, this needs to be transferred to the level to which we backtrack. Upon backtracking, CQESTO performs two operations: *substitution* of the opponent's move, *forgetting* of variables belonging to the player. Identifying these operations helps us making a link to other solvers, such as RAReQS.

The presented operations open several avenues for future work. They may enable connecting CNF-based learning [29] and the circuit-based approach by extending propagation (Definition 2). The discussed connection between CQESTO and RAReQS also opens the possibility of combining the two methods, which would in particularly be beneficial in formulas with high number of quantifiers, where RAReQS may be too heavy-handed. The recently proposed use of machine learning for RAReQS implemented in QFUN [11] could also be used in CQESTO as a look-ahead for future moves of the opponent.

Acknowledgments.

This work was supported by national funds through Fundação para a Ciência e a Tecnologia (FCT) with reference UID/CEC/50021/2013. The author would like to thank Nikolaj Bjørner and João Marques-Silva for the helpful discussions on the topic.

References

1. Ansótegui, C., Gomes, C.P., Selman, B.: The Achilles' heel of QBF. In: National Conference on Artificial Intelligence and the Seventeenth Innovative Applications of Artificial Intelligence Conference (AAAI). pp. 275–281 (2005)
2. Balabanov, V., Jiang, J.R., Scholl, C., Mishchenko, A., Brayton, R.K.: 2QBF: Challenges and solutions. In: Theory and Applications of Satisfiability Testing (SAT). pp. 453–469 (2016), https://doi.org/10.1007/978-3-319-40970-2_28
3. Benedetti, M., Mangassarian, H.: QBF-based formal verification: Experience and perspectives. Journal on Satisfiability, Boolean Modeling and Computation (JSAT) 5(1-4), 133–191 (2008)
4. Bjørner, N., Janota, M.: Playing with quantified satisfaction. In: International Conferences on Logic for Programming LPAR-20, Short Presentations. vol. 35, pp. 15–27. EasyChair (2015)
5. CQESTO website, http://sat.inesc-id.pt/~mikolas/sw/cqesto/res.html
6. Eén, N., Sörensson, N.: An extensible SAT-solver. In: Theory and Applications of Satisfiability Testing (SAT) (2003)
7. Farzan, A., Kincaid, Z.: Strategy synthesis for linear arithmetic games. Proc. ACM Program. Lang. 2(POPL), 61:1–61:30 (Dec 2017), http://doi.acm.org/10.1145/3158149
8. Goultiaeva, A., Seidl, M., Biere, A.: Bridging the gap between dual propagation and CNF-based QBF solving. In: Design, Automation & Test in Europe (DATE). pp. 811–814 (2013)
9. Goultiaeva, A., Van Gelder, A., Bacchus, F.: A uniform approach for generating proofs and strategies for both true and false QBF formulas. In: International Joint Conference on Artificial Intelligence (IJCAI). pp. 546–553 (2011)
10. Janota, M., Klieber, W., Marques-Silva, J., Clarke, E.M.: Solving QBF with counterexample guided refinement. In: Theory and Applications of Satisfiability Testing (SAT). pp. 114–128 (2012)
11. Janota, M.: Towards generalization in QBF solving via machine learning. In: AAAI Conference on Artificial Intelligence (2018)
12. Janota, M., Klieber, W., Marques-Silva, J., Clarke, E.: Solving QBF with counterexample guided refinement. Artificial Intelligence 234, 1–25 (2016)

13. Janota, M., Marques-Silva, J.: Abstraction-based algorithm for 2QBF. In: Sakallah, K.A., Simon, L. (eds.) SAT. pp. 230–244. Springer (2011)
14. Janota, M., Marques-Silva, J.: Solving QBF by clause selection. In: International Joint Conference on Artificial Intelligence (IJCAI) (2015)
15. Janota, M., Marques-Silva, J.: An Achilles' heel of term-resolution. In: Conference on Artificial Intelligence (EPIA). pp. 670–680 (2017)
16. Jordan, C., Klieber, W., Seidl, M.: Non-CNF QBF solving with QCIR. In: Proceedings of BNP (Workshop) (2016)
17. Kleine Büning, H., Bubeck, U.: Theory of quantified boolean formulas. In: Biere, A., Heule, M., van Maaren, H., Walsh, T. (eds.) Handbook of Satisfiability, Frontiers in Artificial Intelligence and Applications, vol. 185, pp. 735–760. IOS Press (2009)
18. Klieber, W., Sapra, S., Gao, S., Clarke, E.M.: A non-prenex, non-clausal QBF solver with game-state learning. In: SAT. pp. 128–142 (2010)
19. Manna, Z., Waldinger, R.: The Logical Basis for Computer Programming, Volume 2. Addison-Wesley (1985)
20. Marques Silva, J.P., Sakallah, K.A.: GRASP: A search algorithm for propositional satisfiability. IEEE Trans. Computers 48(5), 506–521 (1999)
21. Peitl, T., Slivovsky, F., Szeider, S.: Dependency learning for QBF. In: Theory and Applications of Satisfiability Testing (SAT). pp. 298–313 (2017), https://doi.org/10.1007/978-3-319-66263-3_19
22. QBF Eval 2017, http://www.qbflib.org/event_page.php?year=2017
23. Rabe, M.N., Tentrup, L.: CAQE: A certifying QBF solver. In: Formal Methods in Computer-Aided Design, FMCAD. pp. 136–143 (2015)
24. Reynolds, A., King, T., Kuncak, V.: Solving quantified linear arithmetic by counterexample-guided instantiation. Formal Methods in System Design 51(3), 500–532 (2017), https://doi.org/10.1007/s10703-017-0290-y
25. Tentrup, L.: Non-prenex QBF solving using abstraction. In: Theory and Applications of Satisfiability Testing (SAT). pp. 393–401 (2016)
26. Tseitin, G.S.: On the complexity of derivations in the propositional calculus. Studies in Constructive Mathematics and Mathematical Logic Part II, ed. A.O. Slisenko (1968)
27. Tu, K., Hsu, T., Jiang, J.R.: QELL: QBF reasoning with extended clause learning and levelized SAT solving. In: Theory and Applications of Satisfiability Testing - (SAT). pp. 343–359 (2015), https://doi.org/10.1007/978-3-319-24318-4_25
28. Zhang, L.: Solving QBF by combining conjunctive and disjunctive normal forms. In: National Conference on Artificial Intelligence and the Eighteenth Innovative Applications of Artificial Intelligence Conference (AAAI) (2006)
29. Zhang, L., Malik, S.: Conflict driven learning in a quantified Boolean satisfiability solver. In: International Conference On Computer Aided Design (ICCAD). pp. 442–449 (2002)