GhostQ
System Description

William Klieber weklieber@sei.cmu.edu
Carnegie Mellon University
Pittsburgh, Pennsylvania, USA

Abstract

GhostQ is a DPLL-based non-CNF QBF solver. This paper describes a noteworthy feature of GhostQ that has not yet been described in the peer-reviewed literature: support for Plaisted-Greenbaum encoding. For CNF inputs, GhostQ attempts to perform reverse engineering on the CNF formula to create an equivalent circuit representation. Support for reversing the Plaisted-Greenbaum transformation was added to the existing capability for reversing the Tseitin transformation.

Keywords: *QBF, DPLL, non-clausal, Plaisted-Greenbaum encoding*

Submitted November 2018; revised March 2019; published September 2019

1. Overview

GhostQ is a DPLL-based solver that uses *ghost variables* and *learned sequents* to achieve symmetry in handling of universal and existential variables. A description of GhostQ's ghost variables and sequents is in [4]. The original version of GhostQ [5] had support for non-prenex instances, but non-prenex support was dropped in 2012 to focus on prenex instances and add support for a limited form of CEGAR learning [3, 2]. A new feature added in 2017 (but not described in the literature until this paper) is an enhancement to the preprocessor to add support for the Plaisted-Greenbaum encoding. In addition, better support was added for XOR/ITE gates, including allowing them to be handled in instances where the order of clauses has been shuffled. Three configurations of GhostQ have been submitted to QBFEVAL'17 and QBFEVAL'18:

1. `ghostq-pg-cegar`: Plaisted-Greenbaum and CEGAR learning.
2. `ghostq-pg-plain`: Plaisted-Greenbaum, but without CEGAR learning.
3. `ghostq-cegar`: Unchanged from the 2016 version (no Plaisted-Greenbaum).

2. Preliminaries

We consider prenex QBF formulas of the form $Q_1 X_1 ... Q_n X_n. \phi$, where each $Q_i \in \{\exists, \forall\}$ and ϕ is quantifier-free. We say that $Q_1 X_1 ... Q_n X_n$ is the **quantifier prefix** and that ϕ is the **matrix**. A BNF grammar for the matrix ϕ is as follows (using v to denote a variable):

$$\phi \quad ::= \quad v \mid \neg\phi \mid \phi \wedge \ldots \wedge \phi \mid \phi \vee \ldots \vee \phi \mid \phi \Leftrightarrow \phi \mid \texttt{true} \mid \texttt{false}$$

©2019 IOS Press, SAT Association and the authors.

A *literal* is a variable or the negation of a variable. Given a literal ℓ, we write "var(ℓ)" to denote the variable v such that either $\ell = v$ or $\ell = \neg v$. If a literal is an unnegated variable, it is said to have positive polarity. If a literal is a negated variable, it is said to have negative polarity. Given a literal ℓ, we write "$\neg\ell$" to denote the literal with the same variable of ℓ but the opposite polarity. (That is, we remove double negations: if $\ell = \neg x$, then $\neg\ell = x$.) A *clause* is a disjunction of literals. The order of literals in a clause is immaterial; the clause $(x \vee y)$ is considered equivalent to $(y \vee x)$.

Definition (immediate subformula): Given a clause $(\ell_1 \vee ... \vee \ell_n)$, the set of *immediate subformulas* of the clause is $\{\ell_1, ..., \ell_n\}$. For example, x is not an immediate subformula of $(\neg x \vee y)$, but $\neg x$ is.

Definition (positive/negative occurrence): The terms "occur positively" and "occur negatively" are defined inductively as follows: (1) A formula occurs positively in itself. (2) If $\neg\psi$ occurs positively (or respectively negatively) in ϕ, then ψ occurs negatively (resp. positively) in ϕ. (3) If $\psi_1 \wedge ... \wedge \psi_n$ or $\psi_1 \vee ... \vee \psi_n$ occurs positively (resp. negatively) in ϕ, then all the formulas $\psi_1, ..., \psi_n$ occur positively (resp. negatively) in ϕ. (4) If $\psi_1 \Leftrightarrow \psi_2$ occurs (either positively or negatively) in ϕ, then ψ_1 and ψ_2 occur both positively and negatively in ϕ.

Definition (restricted circuit form): A QBF formula is in *restricted circuit form* iff its matrix consists only of variables, conjunction, disjunction, and negation (i.e., there are no occurrences of the biconditional ("$\Leftrightarrow$"), `true`, or `false`).

Gate variables. Given a formula in restricted circuit form, we label each conjunction and disjunction with a *gate variable.* If a subformula ϕ is labelled by a gate variable g, then $\neg\phi$ is labelled by $\neg g$. The variables originally in the formula are called "input variables", in distinction to gate variables.

QBF as a Game. A QBF formula Φ can be viewed as a game between an existential player (Player $\exists$) and a universal player (Player $\forall$):

- Existentially quantified variables are *owned* by Player $\exists$.
- Universally quantified variables are *owned* by Player $\forall$.
- The state of the game consists of an assignment to variables in the matrix of the QBF formula. At the start of the game, the assignment is empty.
- On each turn, the owner of the outermost-quantified unassigned variable assigns it a value (either `true` or `false`).
- The goal of Player $\exists$ is to make Φ be true.
- The goal of Player $\forall$ is to make Φ be false.

2.1 Tseitin transformation

The Tseitin transformation [6] is the usual way of converting a formula (in restricted circuit form) into CNF. In the Tseitin transformation, all the gate variables (also called *Tseitin variables*) are existentially quantified in the innermost quantification block and clauses are added to equate each gate variable with the subformula that it represents. For example, consider the following formula:

$$\Phi \;:=\; \exists e.\, \forall u.\; \underbrace{(e \wedge u)}_{g_1} \vee \underbrace{(\neg e \wedge \neg u)}_{g_2}$$

The subformula $(e \wedge u)$ is labelled by the gate variable g_1, and the subformula $(\neg e \wedge \neg u)$ is labelled by the gate variable g_2. This formula is converted to:

$$\Phi' \;=\; \exists e.\, \forall u.\; \exists g_1 \exists g_2.\; (g_1 \vee g_2) \wedge (g_1 \Leftrightarrow (e \wedge u)) \wedge (g_2 \Leftrightarrow (\neg e \wedge \neg u)) \tag{1}$$

The biconditionals defining the gate variables are converted to clauses as follows:

$$(g_1 \Leftrightarrow (e \wedge u)) \;=\; (\neg e \vee \neg u \vee g_1) \;\wedge\; (\neg g_1 \vee e) \wedge (\neg g_1 \vee u)$$

3. Reverse Engineering for Plaisted-Greenbaum

In the Tseitin encoding, the gate definition $g \Leftrightarrow (x_1 \vee ... \vee x_n)$ is encoded by the following clauses: $(\neg g \vee x_1 \vee ... \vee x_n)$, $(g \vee \neg x_1)$, $\ldots$, $(g \vee \neg x_n)$. During unit propagation, the binary clauses (i.e., clauses with exactly two literals) will force g to be assigned `true` if any x_i gets assigned `true`. The single non-binary clause will force g to be assigned `false` if all the x_i get assigned `false`. The Plaisted-Greenbaum encoding for g may omit either all the binary clauses or the single non-binary clause, depending on how g occurs in the formula. (If g occurs both positively and negatively, then no clauses can be omitted.)

One approach for reversing the Plaisted-Greenbaum encoding is presented in [1]. Here, we present a different approach, which was easier to correctly implement for GhostQ.

Consider a QBF formula Φ of the form $P.\phi$ where P is the quantifier prefix and ϕ is a conjunction of clauses and a non-CNF formula[1.]: $\phi = (clause_1 \wedge ... \wedge clause_m) \wedge \phi_{\text{non-CNF}}$. Consider a literal g, where var(g) is existentially quantified in the innermost quantification block. We define *binary half-def* as follows:

> If (1) g does not occur as an immediate subformula of any non-binary clause and (2) the set of clauses in which g does occur (as an immediate subformula) is the set $\{(g \vee \neg x_1), ..., (g \vee \neg x_n)\}$, then $(g \vee x_1 \vee ... \vee x_n)$ is a *binary half-def* of g in Φ, and $\{(g \vee \neg x_1), ..., (g \vee \neg x_n)\}$ is the set of clauses *associated* with the half-def.

Intuition: With the game semantics of QBF, the set of clauses $\{(g \vee \neg x_1), ..., (g \vee \neg x_n)\}$ forces the existential player to assign g `true` if any of the x_i literals have been assigned `true`. It is safe[2.] to individually replace any occurrence of g (other than in the clauses associated with its half-def) with its half-def because if the half-def evaluates to a different value than g evaluates to, then the matrix is going to evaluate to `false` anyway because one of the associated binary clauses will evaluate to `false`. When all occurrences of g (outside the associated binary clauses) have been replaced, the associated binary clauses can be safely

1. Motivation: We start with a formula in CNF. When we discover clauses that constitute a gate definition, we delete the clauses and insert an equivalence for the gate definition. E.g., $(x \vee y) \wedge (\neg x \vee \neg y) \wedge C_3 \wedge ... \wedge C_n$ might become $(C_3 \wedge ... \wedge C_n) \wedge (x \Leftrightarrow \neg y)$.
2. By saying that the operation is 'safe', it is meant that the operation doesn't change the truth value that the QBF formula evaluates to.

removed. (Alternatively, all occurrences of g, including in the associated binary clauses, can be simultaneously substituted for.) At that point, g should either occur only positively or occur only negatively, allowing it to be substituted with either `true` or `false`.

We define *non-binary half-def* as follows:

> If $\neg g$ occurs (as an immediate subformula) in exactly one clause, and that clause is a non-binary clause $(\neg g \vee x_1 \vee ... \vee x_n)$, then $g \wedge (x_1 \vee ... \vee x_n)$ is a *non-binary half-def* of g in Φ.

Notation: Given two formulas ϕ and ψ and a variable v, let "$\phi[v:=\psi]$" denote the result of taking ϕ and substituting all occurrences of v with ψ. Given a negative literal $\ell = \neg v$, let "$\phi[\ell:=\psi]$" denote $\phi[v:=\neg\psi]$.

Notation: Given a formula ϕ and an assignment $\pi = \langle x_1{:}c_1, ..., x_n{:}c_n\rangle$ (where each c_i is a boolean constant (`true` or `false`)), let "$\phi|_\pi$" denote the substitution of π in ϕ, i.e., $\phi|_\pi = \phi[x_1:=c_1][x_2:=c_2]\cdots[x_n:=c_n]$.

Theorem 1 If a formula ψ of the form $(g \vee x_1 \vee ... \vee x_n)$ is a binary half-def of a literal g in $P.\,\phi$, then $P.\,\phi[g:=\psi]$ has the same truth value as $P.\,\phi$.
Proof. Let us write "$\exists g$" as an abbreviation of "$\exists\,\mathrm{var}(g)$". Since $\mathrm{var}(g)$ is quantified innermost and existentially, it suffices to prove the following: For every assignment π to all variables in ϕ except g, $\exists g.\,\phi[g:=\psi]|_\pi = \exists g.\,\phi|_\pi$. Consider such an assignment π. There are two cases:

1. If $(x_1 \vee ... \vee x_n)|_\pi = \texttt{false}$, then $\psi|_\pi = (g \vee \texttt{false})|_\pi = g|_\pi$, and thus $\phi[g:=\psi]|_\pi = \phi|_\pi$.
2. If $(x_1 \vee ... \vee x_n)|_\pi = \texttt{true}$, then:
   - (a) $\exists g.\,\phi|_\pi \quad = \phi|_{\pi \cup \langle g\,:\,\texttt{false}\rangle} \vee \phi|_{\pi \cup \langle g\,:\,\texttt{true}\rangle}$ (by def of "$\exists$")
   - (b) $\phi|_{\pi \cup \langle g:\texttt{false}\rangle} = \texttt{false}$ (because at least one binary clause with g is false under $\pi \cup \langle g : \texttt{false}\rangle$.)
   - (c) $\exists g.\,\phi|_\pi \quad = \phi[g:=\texttt{true}]|_\pi$ (follows from the above two steps)
   - (d) $\exists g.\,\phi[g:=\psi]|_\pi \quad = \exists g.\,\phi[g:=(\psi|_\pi)]|_\pi$
     $= \phi[g:=\texttt{true}]|_\pi$
   - (e) $\exists g.\,\phi[g:=\psi]|_\pi \quad = \exists g.\,\phi|_\pi$ (follows from steps (c) and (d))

Theorem 2 If a formula ψ of the form $g \wedge (x_1 \vee ... \vee x_n)$ is a non-binary half-def of a literal g in $P.\,\phi$, then $P.\,\phi[g:=\psi]$ has the same truth value as $P.\,\phi$.
Proof. This is similar to the proof of Theorem 1 above. It suffices to prove the following: For every assignment π to all variables in ϕ except g, $\exists g.\,\phi[g:=\psi]|_\pi = \exists g.\,\phi|_\pi$. There are two cases:

1. If $(x_1 \vee ... \vee x_n)|_\pi = \texttt{true}$, then $\psi|_\pi = g|_\pi$, and therefore $\phi[g:=\psi]|_\pi = \phi|_\pi$.
2. If $(x_1 \vee ... \vee x_n)|_\pi = \texttt{false}$, then:
   - (a) $\exists g.\,\phi|_\pi \quad = \phi|_{\pi \cup \langle g\,:\,\texttt{false}\rangle} \vee \phi|_{\pi \cup \langle g\,:\,\texttt{true}\rangle}$

(b) $\phi|_{\pi \cup \langle g\,:\,\texttt{true}\rangle} = \texttt{false}$ because the non-binary clause with $\neg g$ is false under $\pi \cup \langle g : \texttt{true}\rangle$.

(c) $\exists g.\, \phi|_\pi \quad = \phi[g{:=}\texttt{false}]|_\pi$

(d) $\exists g.\, \phi[g{:=}\psi]|_\pi \quad = \phi[g{:=}\texttt{false}]|_\pi$,

Theorem 3 Let ψ be a half-def of a literal g in $P.\,\phi$, and let D be the set of clauses associated with ψ. (I.e., if ψ is binary half-def then D is the set of binary clauses in which g occurs as an immediate subformula; and if ψ is a non-binary half-def then D is the set of the single non-binary clause in which $\neg g$ occurs as an immediate subformula.) Let γ be the result of removing the clauses D from ϕ. Then $P.\,\gamma[g{:=}\psi]$ has the same truth value as $P.\,\phi$.

Proof. It is easy to verify that, for every every clause c in D, $c[g{:=}\psi]$ reduces to $\texttt{true}$.

$$\begin{aligned}
P.\,\phi &= P.\,\phi[g{:=}\psi] && \text{(Theorems 1 and 2)}\\
&= P.\,(\gamma \wedge D)[g{:=}\psi] && \text{(Definition of } \gamma)\\
&= P.\,\gamma[g{:=}\psi] \wedge D[g{:=}\psi] && \text{(Distributive property)}\\
&= P.\,\gamma[g{:=}\psi] \wedge \texttt{true} &&\\
&= P.\,\gamma[g{:=}\psi] &&
\end{aligned}$$

Recall that the Plaisted-Greenbaum encoding can be used to encode a subformula only if the subformula either occurs only positively or occurs only negatively. Thus, after all occurrences of g have been replaced with a half-def and the half-def's associated clauses are removed, g should occur either only positively or only negatively. (Note that the soundness Theorems 1–3 does not depend on how g occurs in $\phi_{\text{non-CNF}}$, but if g does occur both positively and negatively in $\phi_{\text{non-CNF}}$, then the transformation described here in this paper would be unlikely to be helpful.) If g occurs only positively, it can be replaced with $\texttt{true}$; it occurs only negatively, it can be replaced by $\texttt{false}$.

3.1 Algorithm

Theorems 1–3 suggest a simple, straightforward algorithm, shown in Algorithm 1.

Algorithm 1: Naive Plaisted-Greenbaum Reverse Engineering

```
  Input: A QBF formula $P.\,\phi$
1 while (at least one variable in $P.\,\phi$ has a half-def) do
2     Let $\psi$ be a half-def of a variable $g$ in $P.\,\phi$;
3     Let $D$ be the set of (binary clauses or single non-binary clause) associated with
        $\psi$;
4     Let $\gamma$ be the result of removing the clauses $D$ from $\phi$;
5     $\phi := \gamma[g{:=}\psi]$;
6 end
7 Substitute any existential variables that occur only positively with true;
8 Substitute any existential variables that occur only negatively with false;
```

Example. Let $\Phi = \forall u \, \forall w. \, \exists a \, \exists g. \, (g \vee a) \wedge (g \vee u) \wedge (-g \vee w)$. Then $(g \vee \neg a \vee \neg u)$ is a binary half-def of g. Deleting the associated clauses of the half-def and substituting g with its half-def in Φ yields:

$$\forall u \, \forall w. \, \exists a \, \exists g. \, \neg(g \vee \neg a \vee \neg u) \vee w$$

Since g is existential and occurs only negatively, it can be replaced by `false`:

$$\forall u \, \forall w. \, \exists a \, \exists g. \, \neg(\neg a \vee \neg u) \vee w$$

Note Algorithm 1 has an undesirable property: making a transform for one half-def might destroy the clauses needed to transform a different half-def. To help mitigate this issue, GhostQ does two things: (1) It uses heuristics for which order to try available half-defs. (2) Rather than immediately substituting gate variables with their half-defs, it maintains a table (`GateDef`) that maps gate variables to their definitions. (This requires doing an additional check to avoid cycles in the definition table.)

3.2 Implementation Details

The input to the GhostQ preprocessor is a QDIMACS file that includes a set of clauses C. The preprocessor creates and maintains a hashtable `GateDef` that maps gate variables to their definitions. First the preprocessor tries to reverse Tseitin-encoded gates, removing the defining clauses from C and adding the corresponding gate definitions to `GateDef`. Second, it looks for XOR/ITE gates. Third, it looks for Plaisted-Greenbaum gates.

Definition. A half-def ψ of g is a *definite* half-def iff g is not already defined in `GateDef` and either (1) ψ is the only half-def of g (in C) and there are no half-defs of $\neg g$ (in C) or (2) the only innermost-quantified variable in ψ without a gate def is g.

The GhostQ preprocessor handles Plaisted-Greenbaum as shown in Algorithm 2 (but with some additional heuristics for the order in which to try half-defs). Simplification of variables that occur only positively or only negatively is a separate pass in GhostQ.

Algorithm 2: Plaisted-Greenbaum Reverse Engineering

```
1 repeat
2     for each literal $g$ that has a definite half-def $\psi$ (in $C$) do
3         if the below actions won't cause a cycle in the gate defs then
4             Delete (from $C$) the binary clauses or single non-binary clause associated
                with the half-def $\psi$.
5             Let $h$ be a fresh variable. Replace all occurrences of $g$ with $h$ in both the
                clauses $C$ and the gate definitions (in GateDef), but not in $\psi$.
6             Add a gate definition: GateDef[$h$] $= \psi$.
7         end
8     end
9 until fixed point is reached
```

3.3 Competition Results

In QBFEVAL 2017, there were two tracks in which both **`ghostq-pg-cegar`** and **`ghostq-cegar`** competed: Prenex CNF and Prenex 2QBF. In Prenex 2QBF, **`ghostq-pg-cegar`** solved 246 instances (the most of any solver), while **`ghostq-cegar`** solved only 76. The families in which the two variants of GhostQ differed in their number of solved instances are shown in Table 1. This data comes from the QBFEVAL website[3.]. In Prenex CNF, **`ghostq-pg-cegar`** solved 190 instances, while **`ghostq-cegar`** solved 156. The families in which the two variants of GhostQ differed by more than 2 solved instances are shown in Table 2. This data comes from the QBFEVAL website[4.].

Table 1. Number of solved instances in Prenex 2QBF track

Family	`ghostq-cegar`	`ghostq-pg-cegar`
HardwareFixpoint	6	62
RankingFunctions	0	37
Reduction-finding	10	68
sketch	2	0
Sorting_networks	0	5
terminator	18	34

Table 2. Number of solved instances in Prenex CNF track

Family	`ghostq-cegar`	`ghostq-pg-cegar`
Generalized-Tic-Tac-Toe	4	1
HardwareFixpoint	1	16
LinearBitvectorRankingFunction	0	6
Reduction-finding	1	5
terminator	3	8

3. http://www.qbflib.org/solver_families.php?solver=269&year=2017&track=3
http://www.qbflib.org/solver_families.php?solver=235&year=2017&track=3
4. http://www.qbflib.org/solver_families.php?solver=269&year=2017&track=1
http://www.qbflib.org/solver_families.php?solver=235&year=2017&track=1

References

[1] Alexandra Goultiaeva and Fahiem Bacchus. Recovering and utilizing partial duality in QBF. In *Proceedings of the 16th International Conference on Theory and Applications of Satisfiability Testing*, **7962** of *Lecture Notes in Computer Science*, pages 83–99. Springer, 2013.

[2] Mikoláš Janota, William Klieber, Joao Marques-Silva, and Edmund Clarke. Solving QBF with counterexample guided refinement. *Artificial Intelligence*, **234**:1–25, 2016.

[3] Mikoláš Janota, William Klieber, Joao Marques-Silva, and Edmund Clarke. Solving QBF with counterexample guided refinement. In *Proceedings of the 15th International Conference on Theory and Applications of Satisfiability Testing*, **7317** of *Lecture Notes in Computer Science*, pages 114–128. Springer, 2012.

[4] William Klieber, Mikolás Janota, João Marques-Silva, and Edmund M. Clarke. Solving QBF with free variables. In *Proceedings of the 19th International Conference on Principles and Practice of Constraint Programming*, **8124** of *Lecture Notes in Computer Science*, pages 415–431. Springer, 2013.

[5] William Klieber, Samir Sapra, Sicun Gao, and Edmund M. Clarke. A non-prenex, non-clausal QBF solver with game-state learning. In *Proceedings of the 13th International Conference on Theory and Applications of Satisfiability Testing*, **6175** of *Lecture Notes in Computer Science*, pages 128–142. Springer, 2010.

[6] G. S. Tseitin. On the complexity of derivations in the propositional calculus. *Studies in Constructive Mathematics and Mathematical Logic*, **Part II, ed. A.O. Slisenko**, 1968.