A Non-Prenex, Non-Clausal QBF Solver with Game-State Learning

William Klieber, Samir Sapra, Sicun Gao, and Edmund Clarke*

Computer Science Department
Carnegie Mellon University
Pittsburgh, Pennsylvania

Abstract. We describe a DPLL-based solver for the problem of quantified boolean formulas (QBF) in non-prenex, non-CNF form. We make two contributions. First, we reformulate clause/cube learning, extending it to non-prenex instances. We call the resulting technique *game-state learning*. Second, we introduce a propagation technique using *ghost literals* that exploits the structure of a non-CNF instance in a manner that is symmetric between the universal and existential variables. Experimental results on the QBFLIB benchmarks indicate our approach outperforms other state-of-the-art solvers on certain benchmark families, including the `tipfixpoint` and `tipdiam` families of model checking problems.

Keywords: QBF, DPLL, non-clausal, non-prenex, clause learning

1 Introduction

Many problems in formal verification (among other areas) are naturally expressed in the language of QBF. Traditionally, QBF solvers have used conjunctive normal form (CNF). Although CNF works well for SAT solvers, it hinders the work of QBF solvers by impeding the ability to detect and learn from satisfying assignments. In fact, a family of problems that are trivially satisfiable in negation-normal form (NNF) were experimentally found to require exponential time (in the problem size) for existing CNF solvers [18].

Various techniques have been proposed for avoiding the drawbacks of a CNF encoding. Zhang et al. have investigated dual CNF-DNF representations in which a boolean formula is transformed into a combination of an equi-satisfiable CNF formula and an equi-tautological DNF [18]. Sabharwal et al. have developed a QBF modeling approach based a game-theoretic view of QBF [14]. Ansotegui et al. have investigated the use of *indicator variables* [1]. These approaches all help to alleviate the problems of a pure CNF encoding, but we argue that a

* This research was sponsored by the GSRC under contract no. 1041377 (Princeton University), National Science Foundation under contracts no. CCF0429120, no. CNS0926181, no. CCF0541245, and no. CNS0931985, Semiconductor Research Corporation under contract no. 2005TJ1366, General Motors under contract no. GMCMUCRLNV301, Air Force (Vanderbilt University) under contract no. 18727S3, International Collaboration for Advanced Security Technology of the National Science Council, Taiwan, under contract no. 1010717, and the Office of Naval Research under award no. N000141010188.

fully non-clausal approach can lead to even greater improvements, especially for instances produced from deeply-nested circuits.

In addition to combined CNF-DNF techniques, fully non-clausal techniques have recently been investigated. A prenex circuit-based DPLL solver with "don't care" reasoning and clause/cube learning has been developed by Goultiaeva et al. [8]. A non-prenex NNF-based DPLL solver with dependency-directed (non-chronological) backtracking, but without learning, was developed by Egly, Seidl, and Woltran [4]. Non-clausal techniques using symbolic quantifier expansion (rather than DPLL) have been developed by Lonsing and Biere [10] and by Pigorsch and Scholl [13]. Giunchiglia et al. have developed a technique for miniscoping quantifiers (pushing quantifiers inward so as to minimize their scope) [7]. Non-clausal representations have also been investigated in the context of SAT solvers [9, 16, 5].

Most existing DPLL-based QBF solvers perform clause/cube learning. However, traditional clause/cube learning was designed for prenex QBF instances, and it is not optimal for (or even directly applicable to) non-prenex QBF instances. We reformulate clause/cube learning and extend it to the non-prenex case. Additionally, we develop a new propagation technique using *ghost literals*. Experimental results indicate that our approach can beat other state-of-the-art solvers on fixed-point computation instances of the type found in the `tipfixpoint` benchmark family.

2 Preliminaries

We consider non-prenex QBF formulas in negation-normal form[1], as described by the following abstract grammar:

$$\phi \;::=\; e_i \mid \neg e_i \mid u_i \mid \neg u_i \mid \phi \vee ... \vee \phi \mid \phi \wedge ... \wedge \phi \mid \exists e_i\ \phi \mid \forall u_i\ \phi$$

We label each conjunction and disjunction with a *gate variable* of the form g_i, as illustrated in Figure 1. The conjunction/disjunction labelled g_i, together with its quantifier prefix (if any), is labelled with the primed gate variable g_i', as illustrated in Figure 1. As indicated in the abstract grammar, each labelled conjunction/disjunction may have any number of conjuncts/disjuncts.

$$\exists e_{10}\ \Big[\underbrace{[\exists e_{11}\ \forall u_{21}\ \overbrace{(e_{10} \wedge e_{11} \wedge u_{21})}^{g_1}]}_{g_1'} \wedge \underbrace{[\forall u_{22}\ \exists e_{30}\ \overbrace{(e_{10} \wedge u_{22} \wedge e_{30})}^{g_2}]}_{g_2'}\Big]$$

Fig. 1. Example QBF instance with gate labels.

[1] Our solver does not require the use of strict NNF. Subformulas containing no quantifiers can be represented in circuit form.

The term "gate variable" arises from the circuit representation of a propositional formula, in which a gate variable labels a logic gate.

Let "*InFmla*" denote the formula that the QBF solver is given as input. We impose the following restriction on *InFmla*: Every variable in *InFmla* must be quantified exactly once, and no variable may occur free (i.e., outside the scope of its quantifier). The variables that occur in *InFmla* are said to be *input variables*.

We represent an assignment π by the set of literals assigned `true` by π. For example, the assignment $\{e_1, \neg u_2\}$ assigns e_1 `true` and assigns u_2 `false`, while leaving all other variables unassigned. We write "$\pi(\ell)$" to denote the value (`true`, `false`, or `undef`) that π assigns to ℓ, as defined as follows: $\pi(\ell) = \texttt{true}$ if $\ell \in \pi$, $\pi(\ell) = \texttt{false}$ if $\neg\ell \in \pi$, and $\pi(\ell) = \texttt{undef}$ otherwise. For any variable x, we treat $\neg\neg x$ as equivalent to x. An assignment may not include both a variable and its negation. An *input assignment* is an assignment in which every assigned variable is an input variable (as opposed to a gate variable).

Definition 1 (Reduction). The *reduction* of a formula f under an input assignment π, denoted by "$f|\pi$", is constructed from f as follows: For each variable x which is assigned a value by π, we delete the quantifier of x and replace each occurrence of x with its assigned value. For example, if $\pi = \{e_1\}$, then $[\exists e_1.\, \forall u_2.\, (e_1 \wedge u_2)]|\pi = [\forall u_2.\, (\texttt{true} \wedge u_2)]$. Formally:

$$\ell|\pi = \begin{cases} \pi(\ell) & \text{if } \pi(\ell) \neq \texttt{undef} \\ \ell & \text{if } \pi(\ell) = \texttt{undef} \end{cases} \qquad (\exists x.f)|\pi = \begin{cases} f|\pi & \text{if } \pi(x) \neq \texttt{undef} \\ \exists x.(f|\pi) & \text{if } \pi(x) = \texttt{undef} \end{cases}$$

$$(f_1 \wedge ... \wedge f_n)|\pi = (f_1|\pi) \wedge ... \wedge (f_n|\pi)$$

$$(f_1 \vee ... \vee f_n)|\pi = (f_1|\pi) \vee ... \vee (f_n|\pi) \qquad (\forall x.f)|\pi = \begin{cases} f|\pi & \text{if } \pi(x) \neq \texttt{undef} \\ \forall x.(f|\pi) & \text{if } \pi(x) = \texttt{undef} \end{cases}$$

Given two input literals x and y, we say that x is *upstream* of y iff the scope of the quantifier of x contains the quantifier of y. We say that a gate literal g is *upstream* of an input literal y iff every variable that occurs in the subformula g is upstream of y.

2.1 QBF as a Two-Player Game

It is helpful to view QBF as a game between two players, Player `E` and Player `U`. We make the following formal definitions:

- The existentially quantified variables are *owned* by Player `E`.
- The universally quantified variables are *owned* by Player `U`.

Informally, the game formulation goes as follows. Throughout the course of the game, the two players assign values to the variables that they own. The order in which the players assign variables is the quantification order of the variables. On each turn of the game, the owner of the outermost-quantified unassigned variable assigns it a value. The goal of Player `E` is to make *InFmla* true, and the goal of

Player U is to make *InFmla* false. For non-prenex instances, we say that each quantifier-prefixed subformula (e.g., g'_1 and g'_2 in Figure 1) is a *subgame*. It may happen that two or more variables are quantified outermost; e.g., in Figure 1 on page 2, after e_{10} is assigned a value, both e_{11} and u_{22} are quantified outermost. In this case, two subgames have become independent of each other; they may be played in parallel or in series.

Definition 2 (Winning under an assignment). Player U *wins* a formula f under π iff $f|_\pi$ is false. Player E *wins* a formula f under π iff $f|_\pi$ is true. (See Definition 1 for the meaning of $f|_\pi$.) (It would be more proper to say "has a winning strategy for" instead of "wins", but for brevity, we'll say simply "wins".)

For example, in Figure 1, Player U wins g'_2 under the empty assignment, and Player E wins g'_2 under $\{e_{10} : \texttt{true}, u_{22} : \texttt{true}\}$.

Proposition 1 Player E wins $[\exists x\ \phi]$ under π if he wins ϕ under either $\pi \cup \{x\}$ or $\pi \cup \{\neg x\}$. Player U wins $[\forall x\ \phi]$ under π if he wins ϕ under either $\pi \cup \{x\}$ or $\pi \cup \{\neg x\}$.

3 Symbolic Game States

In this section, we introduce *game-state learning*, a reformulation of clause/cube learning. For prenex instances, the game-state formulation is isomorphic to clause/cube learning; the differences are merely cosmetic. However, the game-state formulation is more convenient to extend to the non-prenex case.

To motivate the notation of game-state learning, we start by reviewing certain aspects of clause learning. Suppose the input formula *InFmla* is a prenex CNF QBF whose first clause is $(e_1 \vee e_3 \vee u_4 \vee e_5)$. Under an assignment π, if all the literals in the clause are false, then clearly $\mathit{InFmla}|_\pi$ is false. Moreover, if, under π, all the clause's existential literals are assigned false and none of the clause's universal literals are assigned true (i.e., they may either be assigned false or be unassigned), then $\mathit{InFmla}|_\pi$ is false, since the universal player can win by making all the universal literals in the clause false.

As shown in [20], when the QBF clause learning algorithm is applied to

$$\exists e_1 \exists e_3 \forall u_4 \exists e_5 \exists e_7.\, (e_1 \vee e_3 \vee u_4 \vee e_5) \wedge (e_1 \vee \neg e_3 \vee \neg u_4 \vee e_7) \wedge \ldots$$

it can yield the tautological learned clause $(e_1 \vee u_4 \vee \neg u_4 \vee e_5 \vee e_7)$. Although counter-intuitive, this learned clause can be interpreted in the same way as a non-tautological clause: Under an assignment π, if all the clause's existential literals are assigned false and none of the clause's universal literals are assigned true, then $\mathit{InFmla}|_\pi$ is false.

Learned cubes are similar: Under an assignment π, if all the cube's universal literals are assigned true and none of the cube's existential literals are assigned false, then $\mathit{InFmla}|_\pi$ is true. With game-state learning, we explicitly separate the "must be true" literals from the "may be either true or unassigned" literals. (For non-prenex instances, the division is more complicated than just existential-vs-universal.) Instead of writing a cube $(e_1 \vee u_2 \vee \neg e_3)$, we will write a game-state sequent $\langle \{u_2\}, \{e_1, \neg e_3\} \rangle \models (\texttt{E}$ wins $\mathit{InFmla})$.

Definition 3. A *symbolic game state* is a tuple $\langle L^{\text{now}}, L^{\text{fut}} \rangle$, where L^{now} is a set of literals and L^{fut} is a set of input literals. $\langle L^{\text{now}}, L^{\text{fut}} \rangle$ symbolically represents (or *matches*) exactly those input assignments under which:

1. every literal in L^{now} reduces to `true`, and
2. no literal in L^{fut} is assigned `false` — i.e., for every literal ℓ in L^{fut}, either ℓ is already true or ℓ has not yet been assigned a value (and therefore may become true in the future).

For example, consider again the QBF instance in Figure 1 on page 2. The assignment $\{\neg e_{10}\}$ matches both $\langle \{\neg g'_1\}, \varnothing \rangle$ and $\langle \{\neg g'_1\}, \{u_{21}, \neg u_{21}\} \rangle$ (because $\neg e_{10}$ implies $\neg g'_1$), but not $\langle \{\neg g'_1\}, \{e_{10}\} \rangle$. No assignment matches $\langle \{\neg e_{10}\}, \{e_{10}\} \rangle$.

Definition 4 (Winning under a game state). We say that player P *wins* a formula f under a game state GS, written "$GS \models (P \text{ wins } f)$", iff P wins f under all assignments that match GS. Additionally, we say that P *loses* f under GS, written "$GS \models (P \text{ loses } f)$", iff the opponent of P wins f under GS.

For example, for the QBF instance in Figure 1:

- Neither player wins g'_1 under the game state $\langle \varnothing, \varnothing \rangle$, because Player `U` loses under the matching assignment $\{e_{10}, e_{11}, u_{21}\}$ and Player `E` loses under the matching assignment $\{\neg e_{10}\}$.
- Player `U` wins g'_1 under $\langle \varnothing, \{\neg u_{21}\} \rangle$. For example, under the assignment $\pi = \{e_{11}\}$, $g'_1|_\pi$ is $[\forall u_{21}\, (e_{10} \wedge \texttt{true} \wedge u_{21})]$, which evaluates to `false`.
- Player `E` wins g'_1 under $\langle \{u_{21}\}, \{e_{10}, e_{11}\} \rangle$.

In our solver, instead of learning clauses or cubes, we maintain a game-state database with *sequents* of the form $GS \models (P \text{ wins } g'_i)$. It turns out that whenever we learn a new game-state sequent for a prenex instance, the literals owned by the winner all go in L^{fut}, and the literals owned by the loser and the gate literals go in L^{now}. The relationship between learned game-state sequents and learned clauses/cubes (for prenex instances) is as follows. $\langle L^{\text{now}}, L^{\text{fut}} \rangle \models (\texttt{U} \text{ wins } \mathit{InFmla})$ is equivalent to the learned clause $[\neg \ell_1 \vee ... \vee \neg \ell_n]$ where $\{\ell_1, ..., \ell_n\} = L^{\text{now}} \cup L^{\text{fut}}$ (where L^{now} contains the loser/gate literals and L^{fut} contains the winner literals). This equivalence is easily verified using the interpretation of learned clauses developed on the previous page. Likewise, $\langle L^{\text{now}}, L^{\text{fut}} \rangle \models (\texttt{E} \text{ wins } \mathit{InFmla})$ is equivalent to the learned cube $[\ell_1 \wedge ... \wedge \ell_n]$ where $\{\ell_1, ..., \ell_n\} = L^{\text{now}} \cup L^{\text{fut}}$.

Proposition 2 If $\langle L^{\text{now}} \cup \{\ell\}, L^{\text{fut}} \rangle \models (P \text{ wins } f)$, and ℓ is owned by Player P and the quantifier of ℓ is inside f, then $\langle L^{\text{now}}, L^{\text{fut}} \cup \{\ell\} \rangle \models (P \text{ wins } f)$, provided that $\neg \ell \notin L^{\text{fut}}$.

For example, consider the QBF instance $\forall u_1.\, \exists e_2.\, (u_1 \oplus e_2)$, where "$u_1 \oplus e_2$" means "$(u_1 \wedge \neg e_2) \vee (\neg u_1 \wedge e_2)$". If Player `E` wins under $\langle \{u_1, \neg e_2\}, \varnothing \rangle$, then Proposition 2 tells us that Player `E` wins under $\langle \{u_1\}, \{\neg e_2\} \rangle$.

4 Algorithm

An overview of the top-level solver algorithm is provided in Figure 2. Initially, the current assignment *CurAsgn* is empty. For non-prenex instances, we may temporarily target in on a subgame of the input formula *InFmla* and ignore the rest; the subgame being targetted is recorded in the *TargFmla* global variable. On each iteration of the main loop, we first test to see if we know who wins *TargFmla* under the current assignment. There are two cases:

- If the winner of *TargFmla* is unknown, then we call `DecideLit`, which picks an unassigned input variable (from the first available quantifier block in the prefix of *TargFmla*) and assigns it a value in *CurAsgn*. If there are no more unassigned variables in the quantifier prefix of the current *TargFmla*, then we pick a new *TargFmla* from among the unassigned immediate subformulas of *TargFmla* and try again. After adding a new literal to *CurAsgn*, we call `Propagate` to perform boolean constraint propagation (BCP).
- If the winner is known, then we call `LearnNewGS` to learn a new game-state sequent, adding it to the database. If the new game-state sequent reveals that *InFmla* evaluates to a value v under the empty assignment, then we return v as our final answer. Otherwise, we backtrack. We follow the well-known non-chronological backtracking technique, with the addition that we must also undo changes to *TargFmla* as appropriate. (That is, if we backtrack to the beginning of the k^{th} decision level, then we must restore *TargFmla* to the value that it held at the beginning of the k^{th} decision level. For this purpose, we maintain an array `UndoTarg` that maps each decision level to the value of *TargFmla* to be restored.) After backtracking, the newly-learned game-state sequent will force a literal, so we call `Propagate` to perform BCP. (Is a literal forced even when we leave a subgame b by restoring an old value of *TargFmla* during backtracking? Yes; ghosts of b are forced, as per case 1(b) in Section 4.3.)

```
func Solve() {
    CurAsgn = ∅;
    TargFmla = InFmla;
    while (true) {
        if (the winner of TargFmla under CurAsgn is unknown) {
            DecideLit();  // Picks new TargFmla if necessary.
            Propagate();
        } else {
            GS = LearnNewGS();
            if (TargFmla == InFmla and ∅ matches GS) return winner;
            Backtrack to the earliest point at which GS will force a literal;
            Propagate();
        }
    }
}
```

Fig. 2. Overview of top-level solver algorithm.

4.1 Ghost Literals

Goultiaeva et al. [8] introduce a powerful propagation technique for QBF that significantly improves on existing QBF solvers on a variety of benchmarks. With their technique, if the solver notices that a gate literal g must be true in order for the existential player to win, then g becomes forced. However, this technique is asymmetric between the existential and universal players. A gate literal g is forced if it is needed for the existential player to win, but not if it is needed for the universal player to win. We adapt this technique so that the universal variables benefit from the same propagation technique as do the existential variables and so that the learning procedure for satisfying assignments is just as powerful as for falsifying assignments.

In a prenex solver, for each gate variable g, we would introduce two *ghost* variables, $g\langle\mathtt{U}\rangle$ for Player `U` and $g\langle\mathtt{E}\rangle$ for Player `E`. A ghost literal $g\langle P\rangle$ would be forced whenever we detect that Player P cannot win unless g is made true.

For our non-prenex solver, we need to consider subgames (quantifier-prefixed subformulas, such as g'_1 and g'_2 in Figure 1). We introduce ghost variables of the form $g\langle\mathtt{U}, b\rangle$ and $g\langle\mathtt{E}, b\rangle$ where b is a subgame which contains g as a subformula. A ghost literal $g\langle P, b\rangle$ becomes forced when we detect that Player P cannot win subgame b without g being true. For example, consider the below QBF instance (where g_1 is some propositional formula involving e_1, u_2, and e_3):

$$\exists e_1\, \forall u_2\, \exists e_3\, \forall u_4.\ [\underbrace{[\forall u_5.\ g_1 \vee u_5]}_{g'_2} \wedge u_4] \vee \underbrace{[\forall u_6.\ \neg g_1 \vee u_6]}_{g'_3}$$

Under the empty assignment, $g_1\langle\mathtt{E}, g'_2\rangle$ is forced (because Player `E` cannot win g'_2 under $\varnothing$ unless g_1 is true) and likewise $\neg g_1\langle\mathtt{E}, g'_3\rangle$ is forced.

In order to simplify the propagation and learning procedures, we allow game states to contain ghost literals. A game state with a ghost literal is said to *match* the same input assignments as if the game state contained the corresponding non-ghost gate literal; e.g., $\langle L^{\text{now}} \cup \{g\langle P, b\rangle\},\ L^{\text{fut}}\rangle$ matches the same input assignments as $\langle L^{\text{now}} \cup \{g\},\ L^{\text{fut}}\rangle$.

4.2 Initialization of Game-State Database

In CNF-based QBF solvers, the existential player owns the gate variables[2], and there are clauses (generated from the Tseitin transformation [17]) that ensure that the existential player loses if he assigns a value to a gate variable that turns out to be inconsistent with the inputs to the gate. For example, if $g = e_1 \wedge e_2$, then Player `E` would lose if he assigns $g = \mathtt{true}$ and $e_1 = \mathtt{false}$.

In our solver, instead of generating clauses via the Tseitin transformation, we generate game-state sequents. In a prenex solver, we would generate game-state sequents that ensure that a player P loses if he assigns a ghost gate variable

[2] For CNF solvers, gate variables are introduced when formulas are converted to CNF via the Tseitin transformation [17]; these gate variables are existentially quantified.

a value inconsistent with the gate's inputs. In our non-prenex solver, for each subgame b, we generate game-state sequents that ensure that a player P loses subgame b if he assigns a ghost gate variable $g\langle P, b\rangle$ a value inconsistent with the gate's inputs. For example, if $g = e_1 \wedge e_2$ and subformula g appears in a subgame b, then Player $\mathtt{E}$ would lose b if he assigns $g\langle \mathtt{E}, b\rangle = \mathtt{true}$ and $e_1 = \mathtt{false}$. We construct such game-state sequents as follows. For every gate literal g, if g labels a formula $\ell_1 \wedge ... \wedge \ell_n$ (or $\neg g$ labels a formula $\neg\ell_1 \vee ... \vee \neg\ell_n$), we add the following game-state sequents for each player $P \in \{\mathtt{E}, \mathtt{U}\}$ and each quantifier-prefixed formula b which contains g as a subformula:

- $\langle\{\ell_1, ..., \ell_n, \neg g\}, \varnothing\rangle \models (P \text{ loses } b)$
- $\langle\{\neg\ell_i, g\}, \varnothing\rangle \models (P \text{ loses } b)$ for every $i \in \{1, ..., n\}$

For example, if $g_3 = \neg e_1 \vee \neg u_2$ and g_3 is a subformula of a subgame g'_7, then we add game-state sequents $\langle\{e_1, u_2, g_3\}, \varnothing\rangle \models (\mathtt{E} \text{ loses } g'_7)$, $\langle\{\neg e_1, \neg g_3\}, \varnothing\rangle \models (\mathtt{E} \text{ loses } g'_7)$, and $\langle\{\neg u_2, \neg g_3\}, \varnothing\rangle \models (\mathtt{E} \text{ loses } g'_7)$, among others.

After adding the game-state sequents to the database, we normalize them as follows. Consider a game-state sequent of the form $\langle L^{\text{now}}, L^{\text{fut}}\rangle \models (P \text{ loses } b)$. First, we use Proposition 2 (on page 5) to move input literals owned by the winning player from L^{now} to L^{fut}. Second, we replace each gate literal g in L^{now} with the ghost literal $g\langle P, b\rangle$. For example, consider a game-state sequent $\langle\{e_1, u_2, g_3\}, \varnothing\rangle \models (\mathtt{E} \text{ loses } g'_7)$. We move u_2 using Proposition 2 (assuming that the quantifier of u_2 is within the formula g'_7) and replace g_3 with $g_3\langle \mathtt{E}, g'_7\rangle$, yielding $\langle\{e_1, g_3\langle \mathtt{E}, g'_7\rangle\}, \{u_2\}\rangle \models (\mathtt{E} \text{ loses } g'_7)$.

Recall that a ghost literal $g\langle P, b\rangle$ should become forced when g must be true in order for P to win b. Thus, for every quantifier-prefixed subformula b, the ghost literals $\neg b\langle \mathtt{U}, b\rangle$ and $b\langle \mathtt{E}, b\rangle$ should be forced. To ensure that the propagation procedure in Section 4.3 forces these literals, we add the following game-state sequents for every gate variable b that labels a quantifier-prefixed formula:

- $\langle\{b\langle \mathtt{U}, b\rangle\}, \varnothing\rangle \models (\mathtt{U} \text{ loses } b)$ (to force $\neg b\langle \mathtt{U}, b\rangle$)
- $\langle\{\neg b\langle \mathtt{E}, b\rangle\}, \varnothing\rangle \models (\mathtt{E} \text{ loses } b)$ (to force $b\langle \mathtt{E}, b\rangle$)

4.3 Propagation and Forced Literals

CurAsgn may contain forced ghost literals, so in general we can't say *CurAsgn* is a match for a game-state in the sense of Definition 3, because *CurAsgn* is not necessarily an input assignment. Instead, let us say that *CurAsgn* is a *ghost match* for a game-state sequent $\langle L^{\text{now}}, L^{\text{fut}}\rangle \models (P \text{ loses } b)$ iff every literal in L^{now} is assigned true by *CurAsgn* and no literal in L^{fut} is assigned false by *CurAsgn*.

During the `Propagate` procedure, conceptually we examine each learned game-state sequent *GS* of the form $\langle L^{\text{now}}, L^{\text{fut}}\rangle \models (P \text{ loses } b)$ in which none of the literals in $L^{\text{now}} \cup L^{\text{fut}}$ are assigned `false` and b is a subformula of *TargFmla*. There are three cases:

1. If all literals in L^{now} are true, then *CurAsgn* is a ghost match for *GS*, so P loses b under the current assignment.[3] There are two subcases to consider:
   (a) If $b = \mathit{TargFmla}$, then we know who wins *TargFmla* under the current assignment, so we stop propagation and return to the `Solve` procedure.
   (b) If $b \neq \mathit{TargFmla}$, then for all subgames s that contain b, the ghost variables $b\langle \mathtt{E}, s\rangle$ and $b\langle \mathtt{U}, s\rangle$ are forced to be false (if P=`E`) or true (if P=`U`).

2. If there is exactly one unassigned literal ℓ_U in L^{now}, then $\neg\ell_U$ is forced if:
   (1) ℓ_U is owned by P or is a ghost literal of the form $g\langle P, b\rangle$, and
   (2) ℓ_U is upstream of all unassigned literals in L^{fut}, and
   (3) ℓ_U does not appear outside subgame b if ℓ_U is an input literal (so that forcing $\neg\ell_U$ can't cause P to lose a different subgame).

   For example, consider again the QBF instance in Figure 1 on page 2. The game-state sequent $\langle\{u_{22}, \neg g_2\langle \mathtt{U}, g_2'\rangle\}, \{e_{10}, e_{30}\}\rangle \models (\mathtt{U} \text{ loses } g_2')$ will force $\neg u_{22}$ if $\mathit{CurAsgn} = \{\neg g_2\langle \mathtt{U}, g_2'\rangle, e_{10}\}$. However, $\neg u_{22}$ will not be forced if $\mathit{CurAsgn} = \{\neg g_2\langle \mathtt{U}, g_2'\rangle, e_{30}\}$, since e_{10} is upstream of u_{22}, and thus Player `U` can delay assigning a value to u_{22} until `E` has assigned a value for e_{10}.

3. If more than one literal in L^{now} is unassigned, then *GS* doesn't force a literal.

When a game-state sequent *GS* forces a literal ℓ, we set `antecedent[`ℓ`]` = *GS*.

Watched Literals. We use a straightforward adaptation of the watched-literals rule [11, 6]. For each game-state sequent $\langle L^{\text{now}}, L^{\text{fut}}\rangle \models (P \text{ wins } g)$, we watch two literals in L^{now} and one literal in L^{fut}.

Optimized Implementation of Ghost Literals. If a subformula g occurs in a subgame b, and b itself occurs in a larger subgame s, then we say that this occurrence of g is an *indirect* occurrence in s. For example, in Figure 1, e_{10} occurs directly in g_1' and g_2' but occurs only indirectly in *InFmla*.

If a subformula g occurs directly in only a single subgame b, then we only need to explicitly record only two ghost variables, $g\langle \mathtt{U}, b\rangle$ and $g\langle \mathtt{E}, b\rangle$. For any other quantified formula s that contains g as a subformula,

$$\text{we infer } \underbrace{g\langle P, s\rangle \in \mathit{CurAsgn}}_{(P \text{ needs } g \text{ to win } s)} \text{ iff } \underbrace{g\langle P, b\rangle \in \mathit{CurAsgn}}_{(P \text{ needs } g \text{ to win } b)} \text{ and } \underbrace{b\langle P, s\rangle \in \mathit{CurAsgn}}_{(P \text{ needs } b \text{ to win } s)}$$

since the only way g can influence the value of s is via b. If a subformula g occurs directly in multiple subgames, then we must record two ghost variables (existential and universal) for each subgame in which it directly occurs.

[3] Let $\mathit{CurAsgn}_I = \{\ell \mid \ell \in \mathit{CurAsgn} \text{ and } \ell \text{ is an input literal}\}$. If all literals in L^{now} are input literals, then $\mathit{CurAsgn}_I$ matches *GS*, because all literals in L^{now} are assigned true by $\mathit{CurAsgn}_I$ and no literals in L^{fut} are assigned false by $\mathit{CurAsgn}_I$. If there are ghost literals in L^{now}, then P is still doomed to lose b, because P needs the corresponding gate literals to be true in order to win, but if these gate literals become true, then $\mathit{CurAsgn}_I$ will match *GS* and P loses under *GS*.

4.4 Learning New Game States

As shown in Figure 2 on page 6, when it becomes known which player wins *TargFmla* under the current assignment, we call `LearnNewGS` to learn a new game-state sequent. The only way for it to become known who wins *TargFmla* under *CurAsgn* is for *CurAsgn* to become a ghost match for a game-state sequent in the database (see case 1(a) in Section 4.3). Thus, when we enter `LearnNewGS`, the current assignment is a ghost match for some game state.

The procedure for learning a new game-state sequent is shown in Figure 3. We first make a copy of the existing game state that is a ghost match for the current assignment. We then remove the most recently forced literal in L^{now} (not owned by the winner) by *discharging* it via its antecedent, as detailed in Figure 3. We continue to discharge until the L^{now} slot either is empty or has a *good* unique implication point (UIP), as determined by the criteria from [19][4], or until we hit a literal quantified outside *TargFmla*.

For prenex instances, the procedure for discharging a forced literal is similar to *resolution* in clause learning: If $[x_1 \vee ... \vee x_n \vee \ell]$ and $[\neg\ell \vee y_1 \vee ... \vee y_m]$ are true, then $[x_1 \vee ... \vee x_n \vee y_1 \vee ... \vee y_m]$ is also true. The basic argument for the soundness of the discharge method goes as follows. Let $\langle L_A^{\text{now}} \cup \{\ell\}, L_A^{\text{fut}} \rangle \models (P \text{ wins } f)$ be *GS*, and let $\langle L_B^{\text{now}} \cup \{\neg\ell\}, L_B^{\text{fut}} \rangle \models (P \text{ wins } h)$ be the antecedent of ℓ. Discharging ℓ via its antecedent yields $\langle L_A^{\text{now}} \cup L_B^{\text{now}},\ L_A^{\text{fut}} \cup L_B^{\text{fut}} \rangle \models (P \text{ wins } f)$. To simplify matters, let us assume that ℓ is upstream of every literal in L_B^{fut}, so that ℓ is forced under any assignment that matches $\langle L_B^{\text{now}}, L_B^{\text{fut}} \rangle$. Since P wins f under any assignment that matches $\langle L_A^{\text{now}} \cup \{\ell\}, L_A^{\text{fut}} \rangle$, we conclude that if an assignment π matches both $\langle L_B^{\text{now}}, L_B^{\text{fut}} \rangle$ and $\langle L_A^{\text{now}}, L_A^{\text{fut}} \rangle$ (i.e., if π matches $\langle L_A^{\text{now}} \cup L_B^{\text{now}},\ L_A^{\text{fut}} \cup L_B^{\text{fut}} \rangle$) then ℓ is forced and P wins f.

```
func LearnNewGS() {
    GS = GetMatchingGS().copy();
    do {  ℓ = (most recently forced literal in GS not owned by winner);
          if (ℓ is quantified outside TargFmla) break;
          Discharge(GS, ℓ);
    } until (GS.now.IsEmpty() || HasGoodUIP(GS));
    return GS;
}

func Discharge(GS, ℓ) {
    GS.now.remove(ℓ);
    GS.now = (GS.now ∪ (antecedent[ℓ].now - {¬ℓ}));
    GS.fut = (GS.fut ∪  antecedent[ℓ].fut);
}
```

Fig. 3. Overview of Learning Algorithm

[4] Specifically, an input literal ℓ (owned by the loser) in $\langle L^{\text{now}}, L^{\text{fut}} \rangle$ is a *good* UIP if (1) the decision variable of ℓ's decision level belongs to the losing player, (2) every literal in $(L^{\text{now}} \setminus \{\ell\})$ belongs to an earlier decision level than ℓ, and (3) every literal in L^{fut} that is upstream of ℓ belongs to a decision level earlier than that of ℓ.

Example. Consider the QBF below.

$$\overbrace{\exists e_{10}\ \big[\underbrace{\big[\exists e_{11}\ \forall u_{21}.\ \overbrace{(e_{11} \wedge u_{21})}^{g_1} \vee \overbrace{(e_{11} \wedge \neg u_{21})}^{g_2}\big]}_{g'_3} \vee \underbrace{\big[\forall u_{22} \forall u_{23}.\ e_{10} \wedge u_{22} \wedge u_{23}\big]}_{g'_4}\big]}^{g'_5}$$

Fig. 4. Example non-prenex QBF instance

1. The initial assignment includes $g\langle \mathtt{E}, g'\rangle$ and $\neg g\langle \mathtt{U}, g'\rangle$ for $g \in \{g_3, g_4, g_5\}$.
2. $\langle\{g_1\langle \mathtt{U}, g'_3\rangle, \neg g_3\langle \mathtt{U}, g'_3\rangle\}, \varnothing\rangle \models (\mathtt{E}$ wins $g'_3)$ forces $\neg g_1\langle \mathtt{U}, g'_3\rangle$.
3. $\langle\{g_2\langle \mathtt{U}, g'_3\rangle, \neg g_3\langle \mathtt{U}, g'_3\rangle\}, \varnothing\rangle \models (\mathtt{E}$ wins $g'_3)$ forces $\neg g_2\langle \mathtt{U}, g'_3\rangle$.
4. Player E decides to assign $e_{10} = \mathtt{true}$.
5. All the variables in the outermost quantifier prefix are now assigned, so we must pick a subformula to investigate. We pick g'_3 as the new target subformula.
6. Player E decides to assign $e_{11} = \mathtt{true}$.
7. $\langle\{u_{21}, \neg g_1\langle \mathtt{U}, g'_3\rangle\}, \{e_{11}\}\rangle \models (\mathtt{E}$ wins $g'_3)$ forces $\neg u_{21}$.
8. $\langle\{\neg u_{21}, \neg g_2\langle \mathtt{U}, g'_3\rangle\}, \{e_{11}\}\rangle \models (\mathtt{E}$ wins $g'_3)$ is a (ghost) match for the current assignment. Since g'_3 is the current *TargFmla*, we learn a game state. We discharge $\neg u_{21}$, then $\neg g_2\langle \mathtt{U}, g'_3\rangle$, then $\neg g_1\langle \mathtt{U}, g'_3\rangle$, and finally $\neg g_3\langle \mathtt{U}, g'_3\rangle$, yielding the new game-state sequent $\langle\varnothing, \{e_{11}\}\rangle \models (\mathtt{E}$ wins $g'_3)$.
9. We now backtrack, removing e_{11} and e_{10} from the current assignment and reverting *TargFmla* to *InFmla*.
10. Having backtracked, our newly learned game-state sequent now forces $g_3\langle \mathtt{U}, g'_5\rangle$.
11. $\langle\{g_3\langle \mathtt{U}, g'_5\rangle, \neg g_5\langle \mathtt{U}, g'_5\rangle\}, \varnothing\rangle \models (\mathtt{E}$ wins *InFmla*) matches current assignment.
12. We learn the new game-state sequent $\langle\varnothing, \{e_{11}\}\rangle \models (\mathtt{E}$ wins *InFmla*).
13. The empty assignment matches this new game-state, so our final answer is that $\mathit{InFmla} = \mathtt{true}$.

5 Experimental Results

We implemented the ideas in this paper in a solver which we call *GhostQ*. In our experimental results, GhostQ always did at least as well as CirQit and it outperformed Qube on the `k`, `tipdiam`, and `tipfixpoint` families.

We ran GhostQ on the non-CNF instances from QBFLIB on 2.66 GHz machine with a timeout of 300 seconds. For comparison we show the results for CirQit published in [8] (which were conducted on a 2.8 GHz machine with a timeout of 1200 seconds). (CirQit is not publicly available.) As shown in Table 1, GhostQ performs better CirQit on every benchmark family except `consistency`.

Table 1. Comparison between GhostQ and CirQit.

Family	inst.	GhostQ		CirQit	
Seidl	*150*	**150**	(1606 s)	147	(2281 s)
assertion	*120*	**12**	(141 s)	3	(1 s)
consistency	*10*	0	(0 s)	0	(0 s)
counter	*45*	**40**	(370 s)	39	(1315 s)
dme	*11*	**11**	(13 s)	10	(15 s)
possibility	*120*	**14**	(274 s)	10	(1707 s)
ring	*20*	**18**	(28 s)	15	(60 s)
semaphore	*16*	**16**	(4 s)	16	(7 s)
Total	*492*	261	(2435 s)	240	(5389 s)

Table 2. Comparison between GhostQ and Qube.

Family	inst.	GhostQ		Qube	
bbox-01x	*450*	171	(133 s)	**341**	(1192 s)
bbox_design	*28*	19	(256 s)	**28**	(15 s)
bmc	*132*	43	(266 s)	**49**	(239 s)
k	*61*	**42**	(355 s)	13	(55 s)
s	*10*	10	(1 s)	10	(5 s)
tipdiam	*85*	**72**	(143 s)	60	(235 s)
tipfixpoint	*196*	**165**	(503 s)	100	(543 s)
sort_net	*53*	0	(0 s)	**19**	(176 s)
all other	*121*	9	(38 s)	**23**	(227 s)
Total	*1136*	531	(1695 s)	643	(2687 s)

Table 3. Comparison between GhostQ and Non-DPLL Solvers.

		Timeout 60 s			Timeout 600 s	
Family	inst.	GhostQ	Quantor	sKizzo	GhostQ	AIGsolve
bbox-01x	*450*	171	130	166	178	173
bbox_design	*28*	19	0	0	22	23
bmc	*132*	43	106	83	51	30
k	*61*	42	37	47	51	56
s	*10*	10	8	8	10	10
tipdiam	*85*	72	23	35	72	77
tipfixpoint	*196*	165	8	25	170	133
sort_net	*53*	0	27	1	0	0
all other	*121*	9	49	31	17	35
Total	*1136*	531	388	396	571	537

In Tables 1–2, we give the number of instances solved and the time needed to solve them. (Times shown do not include time spent trying to solve instances where the solver timed out.) In Table 3, we give the number of instances solved.

The `ring` and `semaphore` families consist of prenex instances. The other families are non-prenex, so our solver took advantage of its ability to perform non-prenex game-state learning. During testing of our solver, it was noted that non-prenex learning was especially helpful on the `dme` family.[5]

We compared GhostQ to the state-of-the-art solvers Qube 6.6 [7], Quantor 3.0 [3], and sKizzo 0.8.2 [2]. We ran these solvers on the QBFLIB QBFEVAL 2007 benchmarks [12] on a 2.66 GHz machine, with a time limit of 60 seconds and a memory limit of 1 GB. The results are shown in Tables 2 and 3. We also show the results for AIGsolve published in [13], but these numbers are not directly comparable because they were obtained on a different machine and with a timeout of 600 seconds.

For the CNF benchmarks, we wrote a script to reverse-engineer the QDIMACS file to circuit form and convert it to our solver's input format. (This is similar to the technique in [13], but we also looked for "if-then-else" gates of the form $g = (x\,?\,y : z)$.) Of the four other solvers shown in Tables 2 and 3, Qube is the only other DPLL-based solver, so it is most similar to our solver. Our experimental results show that GhostQ does better than Qube on the `tipdiam` and `tipfixpoint` families (which concern diameter and fixpoint calculations for model checking problems on the TIP benchmarks) and on the `k` family.

The use of ghost literals can help GhostQ in two ways: (1) By treating the gate literals specially instead of treating them as belonging to the existential player, we can more readily detect satisfactions and we can learn more powerful cubes; (2) By using universal ghost literals, we have a more powerful propagation procedure for the universal input literals. (We did not perform unprenexing on any of the originally-CNF benchmarks, so our use of game-state learning doesn't improve performance here.) To further investigate, we turned off downward propagation of universal ghost literals; on most families the effect was negligible, but on `tipfixpoint` we solved only 149 instances instead of 165.

6 Conclusion

In this paper, we have made two contributions. First, we have introduced the concept of *symbolic game states* and used this concept to reformulate clause/cube learning and extend it to the non-prenex case. Using game states, we have also been able to reformulate the techniques for conflict/satisfaction analysis, BCP, and non-chronological backtracking. In all cases, we give a unified presentation which is applicable to both the existential and universal players, instead of using separate terminology and notation for the two players. Further, game states are 'well-behaved' theoretically, in that we no longer need learn and store tautological clauses (or contradictory cubes). Our second contribution is introducing the concept of *ghost literals*, allowing us to improve upon the propagation technique introduced in [8] by eliminating the asymmetry between the players so that

[5] The `dme` family instances were originally given in prenex form, but we pushed the quantifiers inward as a preprocessing step. The unprenexing time was about 0.8 seconds per instance and is included in our solver's total time shown in the table.

the technique can reduce the search space for both the universal and existential players (instead of only the existential player). Experiments show that our techniques work particularly well on certain benchmarks related to formal verification. For future work, it may be worthwhile to investigate whether the ideas of dynamic partitioning [15] can be extended to allow dynamic unprenexing.

References

1. C. Ansótegui, C. P. Gomes, and B. Selman. The Achilles' Heel of QBF. In *AAAI 2005*.
2. M. Benedetti. Evaluating QBFs via Symbolic Skolemization. In *LPAR 2004*.
3. A. Biere. Resolve and Expand. In *SAT 2004*.
4. U. Egly, M. Seidl, and S. Woltran. A Solver for QBFs in Nonprenex Form. In *ECAI 2006*.
5. M. K. Ganai, P. Ashar, A. Gupta, L. Zhang, and S. Malik. Combining strengths of circuit-based and CNF-based algorithms for a high-performance SAT solver. In *DAC 2002*.
6. I. P. Gent, E. Giunchiglia, M. Narizzano, A. G. D. Rowley, and A. Tacchella. Watched Data Structures for QBF Solvers. In *SAT 2003*.
7. E. Giunchiglia, M. Narizzano, and A. Tacchella. Quantifier structure in search based procedures for QBFs. In *DATE 2006*.
8. A. Goultiaeva, V. Iverson, and F. Bacchus. Beyond CNF: A Circuit-Based QBF Solver. In *SAT 2009*.
9. H. Jain, C. Bartzis, and E. M. Clarke. Satisfiability Checking of Non-clausal Formulas Using General Matings. In *SAT 2006*.
10. F. Lonsing and A. Biere. Nenofex: Expanding NNF for QBF Solving. In *SAT 2008*.
11. M. W. Moskewicz, C. F. Madigan, Y. Zhao, L. Zhang, and S. Malik. Chaff: Engineering an Efficient SAT Solver. In *DAC 2001*.
12. M. Narizzano, L. Pulina, and A. Tacchella. QBFEVAL. http://www.qbfeval.org/.
13. F. Pigorsch and C. Scholl. Exploiting structure in an AIG based QBF solver. In *DATE 2009*.
14. A. Sabharwal, C. Ansótegui, C. P. Gomes, J. W. Hart, and B. Selman. QBF Modeling: Exploiting Player Symmetry for Simplicity and Efficiency. In *SAT 2006*.
15. H. Samulowitz and F. Bacchus. Dynamically Partitioning for Solving QBF. In *SAT 2007*.
16. C. Thiffault, F. Bacchus, and T. Walsh. Solving Non-clausal Formulas with DPLL Search. In *Constraint Programming – CP 2004*.
17. 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.
18. L. Zhang. Solving QBF by Combining Conjunctive and Disjunctive Normal Forms. In *AAAI 2006*.
19. L. Zhang and S. Malik. Towards a Symmetric Treatment of Satisfaction and Conflicts in Quantified Boolean Formula Evaluation. In *Constraint Programming – CP 2002*.
20. L. Zhang and S. Malik. Conflict Driven Learning in a Quantified Boolean Satisfiability Solver. In *ICCAD 2002*, 2002.