Dual Proof Generation for Quantified Boolean Formulas with a BDD-Based Solver

Randal E. Bryant (✉) and Marijn J. H. Heule

Computer Science Department
Carnegie Mellon University, Pittsburgh, PA, United States
{Randy.Bryant, mheule}@cs.cmu.edu

Abstract. Existing proof-generating quantified Boolean formula (QBF) solvers must construct a different type of proof depending on whether the formula is false (refutation) or true (satisfaction). We show that a QBF solver based on ordered binary decision diagrams (BDDs) can emit a single *dual proof* as it operates, supporting either outcome. This form consists of a sequence of equivalence-preserving clause addition and deletion steps in an extended resolution framework. For a false formula, the proof terminates with the empty clause, indicating conflict. For a true one, it terminates with all clauses deleted, indicating tautology. Both the length of the proof and the time required to check it are proportional to the total number of BDD operations performed. We evaluate our solver using a scalable benchmark based on a two-player tiling game.

1 Introduction

Adding quantifiers to Boolean formulas, yielding the logic of *quantified Boolean formulas* (QBFs), greatly extends their expressive power [11], but it presents several challenges, including verifying the output of a QBF solver. Unlike a satisfiable Boolean formula, there is no satisfying assignment for a QBF—the formula is simply false or true. Instead, a proof-generating QBF solver must provide a full proof in either case: a *refutation* proof if the formula is false, or a *satisfaction* proof if the formula is true.

Currently, there is little standardization of the proof capabilities or the proof systems supported by different QBF solvers [21]. Some solvers can generate *syntactic* certificates—ones that can be directly checked by a proof checker. For a false formula, these can be expressed in clausal proof frameworks that augment resolution with rules for universal quantification [18]. For a true formula, several QBF solvers can generate term resolution proofs [12], effectively reasoning about a negated version of the input formula represented in disjunctive form. These require the proof checker to support an entirely different set of proof rules.

An even larger number of solvers can generate *semantic* certificates in the form of Herbrand functions for false formulas and Skolem functions for true ones, describing how to instantiate either the universal or the existential variables [21]. These can be used to expand the original formula into a (often much larger) Boolean formula that is checked with a SAT solver [22] or with a high-degree polynomial algorithm [25]. Performing the check often requires far more effort than does running the solver. These approaches, along with others involving syntactic certificates, require at least two passes—one to determine whether the formula is true or false and one to generate the proof.

This paper describes a new approach to proof generation for QBF, where the solver generates a *dual proof*, serving as either a refutation or a satisfaction proof depending on whether the solver determines the formula to be false or true. A dual proof consists of a sequence of clause addition and deletion steps, each preserving equivalence to the original formula. If the proof terminates with the addition of the empty clause, then it demonstrates that the original formula was contradictory and therefore false. If the proof terminates with all clauses removed, then it demonstrates that the original formula was equivalent to a tautology and is therefore true. The proofs are expressed in a clausal proof framework that incorporates extended resolution, as well as rules for universal and existential quantification [13, 14].

We have implemented a QBF solver PGBDDQ based on ordered binary decision diagrams (BDDs) that can generate dual proofs as it operates. As optimizations, PGBDDQ can be directed to generate refutation or satisfaction proofs, and these can be somewhat shorter and take less time to check than dual proofs. Refutation proofs follow the traditional format of a series of truth-preserving steps leading to an empty clause. Satisfaction proofs follow the novel format of a series of falsehood-preserving steps leading to an empty set of clauses. This approach for satisfaction proofs has been previously used as part of a QBF preprocessor [13, 14], but, to the best of our knowledge, ours is the first use in a complete QBF solver. Whether dual, refutation, or satisfaction, the proofs generated by PGBDDQ have length proportional to the number of BDD operations and can readily be validated by a simple proof checker.

For the case of refutation proofs, PGBDDQ builds on the work of Jussila, et al. [17], whose BDD-based QBF solver EBDDRES could generate refutation proofs in an extended resolution framework. Whereas their solver, as well as all other published BDD-based QBF solvers [23, 24], require the BDD variable ordering to be the inverse of the quantification ordering, PGBDDQ allows independent choices for the two orderings. As will be shown, this can lead to an exponential advantage on some benchmarks.

We evaluate the performance of PGBDDQ using a scalable benchmark based on a two-player tiling game. We show that, with the right combination of Tseitin variable placement, BDD variable ordering and elimination variable ordering, a BDD-based QBF solver can achieve performance that scales polynomially with the problem size. In these cases, PGBDDQ can readily outperform state-of-the-art search-based solvers, while having the added benefit that it generates a checkable proof.

2 Background Preliminaries

A *literal* l is either a variable y or its complement $\overline{y}$. We denote the underlying variable for literal l as $Var(l)$, while $\bar{l}$ denotes the complement of literal l.

A *clause* is a set of literals, representing the disjunction of a set of complemented and uncomplemented variables. The empty clause, indicating logical falsehood, is written $\bot$. We consider only *proper* clauses, where a literal can only occur once in a clause, and a clause cannot contain both a variable and its complement. Logical truth, or tautology, is denoted $\top$ and represented by an empty set of clauses. For clarity, we write clauses as Boolean formulas, such as $x \wedge y \rightarrow z$ for the clause $\{\overline{x}, \overline{y}, z\}$. As a special case, the unit clause consisting of literal l is simply written as l.

ITE: For Boolean values a, b, and c, the *ITE* operation (short for "If-Then-Else") is defined as: $\mathit{ITE}(a,b,c) = (a \wedge b) \vee (\neg a \wedge c)$. This can be also be written as a conjunction of clauses: $\mathit{ITE}(a,b,c) = (a \rightarrow b) \wedge (\neg a \rightarrow c)$.
QBF: We consider quantified formulas in *prenex normal form* over a set of *input variables* X, with input formula Φ_I having the form $\Phi_I = Q_1 X_1\, Q_2 X_2 \cdots Q_m X_m\ \psi_I$. The *quantifier prefix* $\mathcal{Q}_I = Q_1 X_1\, Q_2 X_2 \cdots Q_m X_m$ consists of a series of *quantifier blocks*. Each block j has an associated quantifier $Q_j \in \{\forall, \exists\}$ and a set of variables $X_j \subseteq X$, such that the sets $X_1, X_2, \ldots, X_m$ form a partitioning of X. The formula *matrix* ψ_I is given as a set of clauses referred to as the *input* clauses. An input variable x occurring in some partition X_j is said to be *universal* (respectively, *existential*) when $Q_j = \forall$ (resp., $Q_j = \exists$) and is said to be at *quantification level* j. The type and level of each literal l matches that of its underlying variable $\mathit{Var}(l)$.
Resolution: Let C and D be clauses, where C contains variable y and D contains its complement $\overline{y}$. We also require that there can be no literal $l \in C$, with $l \neq y$, such that $\overline{l} \in D$. The *resolvent* clause is then defined as $\mathit{Res}(C,D) = C \cup D - \{y, \overline{y}\}$. When C and D do not satisfy the above requirements, then $\mathit{Res}(C,D)$ is undefined. This definition does not allow the resolvent to be a tautology.

The resolution operation extends to linear chains and sets of clauses, as well. For a clause sequence $C_1, C_2, \ldots, C_k$, we define its resolvent as:

$$\mathit{Res}(C_1, C_2, \ldots, C_k) = \mathit{Res}(C_1, \mathit{Res}(C_2, \cdots, \mathit{Res}(C_{k-1}, C_k) \cdots))$$

The sequence $C_1, C_2, \ldots, C_k$ is termed the *antecedent*. Again, the operation is undefined if any individual application of the operation is undefined. For a set of clauses ψ, we define $\mathit{Res}(\psi)$ as the set of all resolvents that can be generated from sequences comprised of clauses from ψ with each clause used at most once per sequence.

As a separate notation, for a set of clauses ψ, we let $\mathit{Res}_y(\psi)$ be the set of all defined resolvents $\mathit{Res}(C,D)$ with $C, D \in \psi$, $y \in C$, and $\overline{y} \in D$.
Extension: Extended resolution [28] allows the introduction of *extension variables* to serve as a shorthand notation for other formulas. Generalizing extended resolution to quantified formulas requires additional considerations regarding 1) the distinction between existentially and universally quantified variables, and 2) the position of the extension variables within the quantification ordering. In particular, as extension variables are generated, they must be classified as existential and be inserted into intermediate positions in the ordering [3, 17]. To support this capability, we associate a *quantification level* $\lambda(y)$ with each input and extension variable y. For input variable x, where $x \in X_j$, we define $\lambda(x) = 2j - 1$. Input variables will therefore have odd values for λ. Each extension variable e will be assigned an even value for $\lambda(e)$ according to rules defined below. For literal l, we define $\lambda(l) = \lambda(\mathit{Var}(l))$.

As clauses are added and deleted, and as extension variables are introduced, a formula will be maintained with an overall form

$$\Phi = Q_1 X_1\, \exists E_1\, Q_2 X_2\, \exists E_2 \cdots Q_m X_m\, \exists E_m\ \psi \tag{1}$$

where $E_1, E_2, \ldots, E_m$ is a partitioning of the set of extension variables. The quantifier prefix $\mathcal{Q}$ in (1) is therefore an alternation of input and extension variables, with all extension variables being existentially quantified. We can also view the quantifier prefix

as simply being a set of variables y, being ordered by the values of $\lambda(y)$, and where y is universal when $\lambda(y) = 2j - 1$ with $Q_j = \forall$. Otherwise, y is existential. We use set notation when referring to the quantifier prefix, recognizing that the partitioning of variables into quantifier blocks and the associated quantifier types, are defined implicitly by the function λ.

Two quantifier prefixes $\mathcal{Q}$ and $\mathcal{Q}'$, each with m input variable blocks, are said to be *compatible* when $Q_j = Q'_j$ for $1 \leq j \leq m$, and $\lambda(y) = \lambda'(y)$ for all $y \in \mathcal{Q} \cap \mathcal{Q}'$, where the unprimed and primed symbols correspond to $\mathcal{Q}$ and $\mathcal{Q}'$, respectively.

Extension introduces existential variable e by adding a set of *defining clauses* θ to the matrix and adding e to the quantifier prefix. Consider QBF $\Phi = \mathcal{Q}\,\psi$. Let e be a fresh variable (i.e., $e \notin \mathcal{Q}$) and let θ be a set of clauses that are *blocked* on e [5]. That is, each clause in θ must contain either e or $\overline{e}$, and for any clauses $C, D \in \theta$ for which $e \in C$ and $\overline{e} \in D$, there must be some other literal $l \in C$ such that $\overline{l} \in D$, and therefore $Res_e(\theta) = \emptyset$. Define $\Phi' = \mathcal{Q}'\,\psi'$ as follows. Variable e is assigned quantification level $\lambda(e) = \max\{Even(\lambda(y)) | y \in Var(\theta), y \neq e\}$, where $Var(\theta)$ is defined to be the set of all variables occurring in the clauses in θ. Function $Even$ rounds a number up to the next higher even value, i.e., $Even(a) = 2\lceil a/2 \rceil$. This definition guarantees that $\lambda(e)$ is even and that every variable y occurring in θ will have $\lambda(y) \leq \lambda(e)$. Letting $\mathcal{Q}' = \mathcal{Q} \cup \{e\}$ and $\psi' = \psi \cup \theta$, it can be shown that Φ' is true if and only if Φ is true [17].

Boolean Functions: The *restriction* of Boolean function f with respect to variable x, denoted $f|_x$ is defined as the function that results when variable x is assigned value 1. Similarly, $f|_{\overline{x}}$ is defined as the function that results when x is assigned value 0.

The *Shannon expansion* relates a Boolean function to its restrictions with respect to a variable and its complement. For a function f and variable x:

$$\begin{aligned} f &= \mathit{ITE}\big(x, f|_x, f|_{\overline{x}}\big) \\ &= \big(x \rightarrow f|_x\big) \wedge \big(\overline{x} \rightarrow f|_{\overline{x}}\big) \end{aligned} \tag{2}$$

We will find clausal form (2) to be of use in generating satisfaction proofs.

For Boolean function f and variable x we can define the existential and universal quantifications of f with respect to x as projection operations that eliminate the dependency on x through either disjunction or conjunction:

$$\exists x\, f = f|_x \vee f|_{\overline{x}} \tag{3}$$

$$\forall x\, f = f|_x \wedge f|_{\overline{x}} \tag{4}$$

BDDs: A reduced, ordered binary decision diagram (BDD) provides a canonical form for representing a set of Boolean functions, and an associated set of algorithms for constructing them and testing their properties [1, 7, 8]. A set of functions is represented as a directed acyclic graph, with each function indicated by a pointer to its root node. We will therefore use the symbol u to refer at times to 1) a node in the BDD, 2) the subgraph of the BDD having u as its root, 3) the function represented by this subgraph, and 4) an extension variable associated with the node.

The ordered BDD representation requires defining a total ordering of the variables. Unlike other BDD-based QBF solvers [17, 23, 24], PGBDDQ allows this ordering to be independent of the ordering of variables in the quantifier prefix. The two leaf nodes

are denoted L_0 and L_1, representing the constant functions **0** and **1**, respectively. Each nonterminal node u has an associated variable and two children indicating branches for the two possible values of the variable.

BDD packages support multiple operations for constructing and testing the properties of Boolean functions represented by a BDD. A number of these are based on the *Apply* algorithm [6]. Given root nodes u and v representing functions f and g, respectively, and a Boolean operation (e.g., AND), the algorithm generates a root node w representing the result of applying the operation to those functions (e.g., $f \wedge g$). It operates by traversing its arguments via a series of recursive calls, using a table to cache previously computed results. Variants of the Apply algorithm can also perform restriction and quantification.

QBF Solving with a BDD: With the ability to perform disjunction, conjunction, and quantification of Boolean functions, there is a straightforward algorithm for solving a QBF with a BDD. It starts by computing a representation of the formula matrix using the Apply algorithm with operation $\vee$ for each clause and conjuncting these using the Apply algorithm with operation $\wedge$. Then, quantifiers are eliminated by working from the innermost quantifier block X_m and working outward, using either universal or existential quantifier operations. At the end, the BDD will be reduced to either L_0 indicating that the formula is false, or L_1 indicating that the formula is true. This basic algorithm can be improved by deferring some of the conjunctions and by carefully selecting the order of quantification within each quantifier block [23, 24].

3 Logical Foundations

A *clausal proof* consists of a sequence of steps starting with the clauses in the input formula Φ_I. Each step either adds a set of clauses, and possibly an extension variable, or it removes a set of clauses. These additions and removals define a sequence of QBFs $\Phi_1, \Phi_2, \ldots, \Phi_t$, with $\Phi_1 = \Phi_I$ and each Φ_i of the form $\mathcal{Q}_i\, \psi_i$.

For a refutation proof, each step i must preserve truth, i.e., $\Phi_i \rightarrow \Phi_{i+1}$, and it must end with $\bot \in \psi_t$. This construction serves as a proof that $\Phi_I = \Phi_1 \rightarrow \Phi_2 \rightarrow \cdots \rightarrow \Phi_t = \bot$, and therefore the input formula is false. A satisfaction proof follows the same general format, except that it requires each step i to preserve falsehood: $\Phi_{i+1} \rightarrow \Phi_i$, and it reaches a final result with $\psi_t = \emptyset$. This construction serves as a proof that $\top = \Phi_t \rightarrow \Phi_{t-1} \rightarrow \cdots \rightarrow \Phi_1 = \Phi_I$, and therefore the input formula is true. A *dual* proof requires that each step preserves equivalence: $\Phi_i \leftrightarrow \Phi_{i+1}$, i.e., it is both truth and falsehood preserving. Only the final step with $\psi_t \in \{\bot, \top\}$ determines whether it is a refutation or a satisfaction proof.

3.1 Inference Rules

Table 1 shows the equivalence-preserving inference rules we use in our proofs. These are based on *redundant clauses*—cases where there are two sets of clauses ψ and θ such that $\mathcal{Q}\, \psi \leftrightarrow \mathcal{Q}'\, (\psi \cup \theta)$, for compatible prefixes $\mathcal{Q}$ and $\mathcal{Q}'$. Thus, adding clauses θ to the matrix ψ defines an equivalence-preserving addition rule, while deleting them from the matrix $\psi \cup \theta$ defines an equivalence-preserving removal rule.

Table 1. Inference rules where clause set θ is redundant with respect to the clauses in ψ.

Addition	Removal	Requirements
Resolution addition	Resolution deletion	$\theta \subseteq Res(\psi)$.
Universal reduction	—	$\theta = \{C\}$. l universal. $\lambda(l') < \lambda(l)$ for all existential $l' \in C$. $C \cup \{l\} \in \psi$.
Extension	Existential elimination	y existential. $y \notin Var(\psi)$. $y \in Var(C)$ for all $C \in \theta$. $Res_y(\theta) \subseteq \psi$. $\lambda(y') \leq \lambda(y)$ for all $y' \in Var(\theta)$.

We have already described resolution in Section 2. Universal reduction (also known as "forall reduction" [4, 17]) is the standard rule for eliminating universal variables in a QBF refutation proof [18].

The extension rule forms the basis for adding extension variable $y = e$ and its defining clauses θ. For this case, the clauses in θ are blocked with respect to y, and therefore $Res_y(\theta) = \emptyset$. As a deletion rule, the existential elimination rule is used to remove extension variables and their defining clauses, as well as to remove the existential input variables. It is a generalization of *blocked clause elimination* [5] in that the clauses in θ need not be blocked, as long as ψ contains all of the resolvents with respect to variable y. The redundancies used by the resolution, extension, and existential elimination rules are special cases of the quantified resolution asymmetric tautology (QRAT) property [13, 14].

3.2 Integrating Proof Generation into BDD Operations

As described in [16, 17, 26] and [9], we use a BDD to represent Boolean functions defined by applying Boolean operations to the input variables X. When creating node u, we introduce an extension variable, also referred to as u, with up to four defining clauses. For node u with variable x, and children nodes u_1 and u_0, these clauses encode the formula $u \leftrightarrow \mathit{ITE}(x, u_1, u_0)$. As described in Section 2, we will have $\lambda(u) = \max\{\lambda(x) + 1, \lambda(u_1), \lambda(u_0)\}$.

As in [9], we associate leaf nodes L_0 and L_1 directly with logical values $\bot$ and $\top$. When constructing node u, if either u_1 or u_0 is a leaf node, the defining clauses may be simplified, and some may degenerate to tautologies. By defining $\lambda(\bot) = \lambda(\top) = 0$, we can still use the above formula to define the value of $\lambda(u)$, such that $\lambda(u_1) \leq \lambda(u)$, $\lambda(u_0) \leq \lambda(u)$, and $\lambda(x) < \lambda(u)$. This guarantees that the value of $\lambda(u)$ is greater or equal to that of any node or variable occurring in the subgraph with root u.

For node u, define its *support set* $S(u)$ as the set of variables occurring at some node in the subgraph with root u. Based on our construction, any node u will have $\lambda(u) = 2j$ if and only if there is some j and some x for which $x \in X_j \cap S(u)$, and this property does not hold for any $j' > j$.

As a final notation, let $\theta(u)$ denote the set consisting of the defining clauses for all nodes in the subgraph with root u.

The BDD package implements the set of operations shown in the Table 2. Each generates a result node w, and it also generates sets of clauses forming extended reso-

Table 2. Required BDD Operations. Each generates a root node plus a set of proofs.

Operation	Arguments	Result	Proved Properties	
			Truth Preserving	Falsehood Preserving
FROMCLAUSE	C	$u = \bigvee_{l \in C} l$	$C, \theta(u) \vdash u$	$u, \theta(u) \vdash C$
APPLYAND	u, v	$w = u \wedge v$	$u \wedge v \rightarrow w$	$w \rightarrow u, w \rightarrow v$
APPLYOR	u, v	$w = u \vee v$	$u \rightarrow w, v \rightarrow w$	$w \rightarrow u \vee v$
RESTRICT	u, l	$w = u\|_l$	$l \wedge u \rightarrow w$	$l \wedge w \rightarrow u$

lution proofs of some properties relating the result to the arguments. As shown, some of these properties are truth preserving, while others are falsehood preserving. In each of these, C indicates a clause, u, v, and w are BDD nodes (or their associated extension variables), and l is a literal of an input variable.

These operations serve the following roles:

- FROMCLAUSE generates the BDD representation u of a clause C. It also generates a set of resolution steps proving that the unit clause u is logically entailed by the input clause and defining clauses: $u \in Res(\{C\} \cup \theta(u))$, and the converse: $C \in Res(\{u\} \cup \theta(u))$.
- APPLYAND generates the BDD representation w of the conjunction of its arguments. It also generates a proof that the extension variables for the argument and result nodes satisfy $u \wedge v \rightarrow w$, as well as a proof of the converse: $w \rightarrow u$ and $w \rightarrow v$, and therefore $w \rightarrow u \wedge v$.
- APPLYOR generates the BDD representation w of the disjunction of its arguments. Its generated proofs include $u \rightarrow w$ and $v \rightarrow w$, implying that $u \vee v \rightarrow w$, as well as the converse: $w \rightarrow u \vee v$.
- RESTRICT generates the restriction w of argument u with respect to literal l. It generates proofs that the operation satisfies *downward* implication: $l \wedge u \rightarrow w$, and also *upward* implication: $l \wedge w \rightarrow u$. This operation has the property that for $x = Var(l)$, variable x will not occur in the subgraph with root w, i.e., $x \notin S(w)$.

4 Integrating Proof Generation into a QBF Solver

PGBDDQ solves a QBF by maintaining a set T of root nodes, which we refer to as "terms." Each term is the result of conjuncting and applying elimination operations to some subset of the input clauses. T initially contains the root nodes for the BDD representations of the input clauses. The solver repeatedly removes one or two terms from T, performs a quantification or conjunction operation, and adds the result to T, except that terms with value L_1 are not added. Quantifiers are eliminated in reverse order, starting with block X_m and continuing through X_1. The process continues until either some generated term is the leaf value L_0, indicating that the formula is false, or the set becomes empty, indicating that the formula is true. The solver simultaneously generates proof steps, including ones that add a unit clause u for each node $u \in T$.

Our presentation describes the general requirements for applying conjunction and elimination operations. These operations can be used to implement the basic method

described in Section 2, as well as more sophisticated strategies that defer conjunctions until they are required before performing some of the elimination operations [23, 24].

Universal quantification commutes with conjunction and so can be applied to the terms independently. Applying existential quantification, on the other hand, requires performing conjunction operations until the variables to be quantified occur only in a single term.

4.1 Dual Proof Generation

For both technical and implementation reasons, which we explain below, we require the input formula to have only a single variable in each quantifier block. This restriction can be satisfied by rewriting an arbitrary QBF, such that a quantifier block with k variables is *serialized*, splitting it into a sequence of k distinct quantification levels.

When generating a dual proof, the solver generates steps proving that each update to the set of terms T preserves equivalence with the input formula. More formally, consider a matrix ψ containing the following clauses: 1) unit clause u for each $u \in T$, plus 2) all of the defining clauses $\theta(u)$ for the subgraph rooted by each node $u \in T$. Let $\mathcal{Q}$ be the compatible quantifier prefix formed by augmenting input prefix $\mathcal{Q}_I$ with the extension variables associated with the nodes in these subgraphs. Then each update preserves the invariant that $\mathcal{Q}_I\, \psi_I \leftrightarrow \mathcal{Q}\, \psi$. Furthermore, the solver takes care to systematically delete clauses once they are no longer needed, using the removal rules listed in Table 1. That enables it to finish with an empty set of clauses in the event the formula is true. The initial set of terms T consists of a root node u for each input clause C, and the solver uses the proof that $C, \theta(u) \vdash u$ to justify adding unit clause u to the proof. It then uses this unit clause, plus the proof that $u, \theta(u) \vdash C$ to justify deleting input clause C.

Each step proceeds by generating new terms and by adding and removing clauses in the proof. Suppose the step involves computing results with root nodes $w_1, ..., w_n$ based on argument terms $u_1, \ldots, u_k$. If any of the result nodes is BDD leaf L_0, then the formula is false. The solver can use truth-preserving rules generated by the BDD operations to justify adding an empty clause. Otherwise, the solver removes the argument terms from T and adds the result nodes, except for any equal to BDD leaf L_1. The solver uses the existing unit clauses plus the truth-preserving rules to justify adding unit clauses for each newly added term. It then uses the falsehood-preserving rules and the newly added unit clauses to justify deleting the unit clauses associated with the argument terms. It must also explicitly generate rules to remove some intermediate clauses that are added during these proof constructions. Other clauses, including the defining clauses for the BDD nodes and the clauses added during the BDD operations get removed by a separate process described in Section 4.2. The net effect for each step then is to replace the argument terms in T by the non-constant result terms, maintaining a unit clause for each term in T as part of the proof.

Conjunction operations. For $u, v \in T$, the solver computes $w = \textsc{ApplyAnd}(u, v)$. For the case where $w = L_0$ the generated truth-preserving proof will be the clause $\overline{u} \vee \overline{v}$, which resolves with unit clauses u and v to generate the empty clause—the solver has proved that the formula is false.

Otherwise, the solver sets T to be $T - \{u, v\} \cup w$. The proof for adding unit clause w follows by resolving the unit clauses u and v with the generated clause $\overline{u} \vee \overline{v} \vee w$, (i.e., $u \wedge v \rightarrow w$). The generated clauses $w \rightarrow u$ and $w \rightarrow v$ each resolve with unit clause w to justify deleting unit clauses u and v.

Universal elimination operation. This operation is performed when $Q_j = \forall$, and by our restriction, we must have $X_j = \{x\}$ for some universal variable x. We also require that the input variables for blocks $X_{j'}$ such that $j' > j$ have already been eliminated.

Since universal quantification commutes with conjunction, the solver can quantify each term individually and let subsequent conjunction operations perform the conjunction indicated in (4). That is, for each $u \in T$ such that $x \in S(u)$, operation RESTRICT is used to compute the two restrictions $w_x = u|_x$ and $w_{\overline{x}} = u|_{\overline{x}}$. These will generate proofs of two downward implications: $l \wedge u \rightarrow w_l$ for $l \in \{x, \overline{x}\}$, as well as proofs of two upward implications: $l \wedge w_l \rightarrow u$.

If w_l equals leaf node L_0 for either $l = x$ or $l = \overline{x}$, then the corresponding downward implication will be a clause of the form $l \wedge u \rightarrow \bot = \overline{l} \vee \overline{u}$. Resolving this with the unit clause u and applying universal reduction generates the empty clause—the solver has proved that the formula is false.

Consider the general case, where neither w_x nor $w_{\overline{x}}$ is a leaf node. The solver sets $T = T \cup \{w_x, w_{\overline{x}}\} - \{u\}$. The downward implications $l \wedge u \rightarrow w_l$ can be resolved with unit clause u to yield the clause $l \rightarrow w_l$ for $l \in \{x, \overline{x}\}$. We can be certain that $\lambda(w_l) < \lambda(x)$ for both values of l, since $x \notin S(w_l)$. Applying universal reduction to the two generated clauses then yields the unit clauses w_x and $w_{\overline{x}}$. Resolving each unit clause w_l with the upward implication $l \wedge w_l \rightarrow u$ gives the clause $l \rightarrow u$, for $l \in \{x, \overline{x}\}$. Resolving these with each other justifies deleting unit clause u. Intermediate clauses $\overline{x} \rightarrow w$, $x \rightarrow w$, $x \rightarrow w_x$, and $\overline{x} \rightarrow w_{\overline{x}}$ are removed by resolution deletion.

The case where one of the restrictions is the leaf node L_1 is handled similarly to the general case, except that this node is not added to T.

Our implementation applies the conjunction operation to terms w_x and $w_{\overline{x}}$ immediately after they are generated to avoid causing the number of terms to expand by a factor of 2^k when the formula contains a sequence of k universal quantifiers.

Existential elimination operations. This operation is performed when $Q_j = \exists$. We can assume that $X_j = \{x\}$ for some existential variable x. We require that the input variables for blocks $X_{j'}$ such that $j' > j$ have already been eliminated. We also require the conjunction operations to have reduced T to contain at most one node u such that $x \in S(u)$. The solver proceeds as follows to existentially quantify x from u yielding a new term w and creating the justification for adding unit clause w. It also removes unit clause u, as well as some intermediate clauses. Note that w can equal L_1, but not L_0.

1. Compute $u_x = \text{RESTRICT}(u, x)$ and $u_{\overline{x}} = \text{RESTRICT}(u, \overline{x})$, generating proofs of the downward implications $x \wedge u \rightarrow u_x$ and $\overline{x} \wedge u \rightarrow u_{\overline{x}}$, as well as the upward implications $x \wedge u_x \rightarrow u$ and $\overline{x} \wedge u_{\overline{x}} \rightarrow u$. Resolving the two downward implications with the unit clause u justifies adding clauses $C_x = x \rightarrow u_x$ and $C_{\overline{x}} = \overline{x} \rightarrow u_{\overline{x}}$. These clauses form the Shannon expansions (2) of u with respect to variable x.
2. For $l \in \{x, \overline{x}\}$, resolving clause C_l with the upward implication $l \wedge u_l \rightarrow u$ justifies adding clauses $x \rightarrow u$ and $\overline{x} \rightarrow u$. Resolving these with each other justifies deleting unit clause u. This step completes the replacement of u by its Shannon expansion.

3. Apply *clause removal* to remove every clause containing a literal l such that $\lambda(l) > \lambda(x) = 2j - 1$. This is described in Section 4.2.
4. C_x and $C_{\overline{x}}$ are the only clauses remaining that contain either x or $\overline{x}$. Resolving these with each other justifies adding clause $u_x \vee u_{\overline{x}}$. The existential elimination rule can now be applied to justify deleting C_x and $C_{\overline{x}}$, with the result that there will be no further clauses containing any literal l with $\lambda(l) \geq \lambda(x)$.
5. Compute $w = \text{APPLYOR}(u_x, u_{\overline{x}})$, generating three proofs: $u_x \rightarrow w$, $u_{\overline{x}} \rightarrow w$, and $w \rightarrow u_x \vee u_{\overline{x}}$.
6. If w is leaf node L_1, then the falsehood-preserving proof generated by APPLYOR derives the clause $u_x \vee u_{\overline{x}}$. This proof justifies deleting the instance of this clause added in step 5. If w is a nonleaf node, then the first two proofs from Step 5 can be resolved with the clause $u_x \vee u_{\overline{x}}$ to justify adding unit clause w, and the third can be resolved with this unit clause to justify deleting clause $u_x \vee u_{\overline{x}}$. This completes the replacement of u by the disjunction of its two restrictions, as in (3).
7. If w is leaf node L_1, then set T to $T - \{u\}$. Otherwise, set it to $T - \{u\} \cup \{w\}$.

Overall Operation: For a false formula, the solver will terminate with the generation of leaf value L_0 during a conjunction or universal quantification operation. These cases will cause the proof to terminate with the addition of an empty clause. For a true formula, the solver will finish with T equal to the empty set, since it never adds a leaf node to T. A final clause removal operation with quantification level 0 then yields $\psi_t = \emptyset$.

We can see now why we impose the restriction that any quantifier block X_j with $Q_j = \forall$ contain only one variable. Without it, the universal variable elimination operation may not be possible. Suppose $X_j = \{x, x'\}$. Attempting to perform the universal quantification operation on variable x could yield a BDD node w_l, with either $l = x$ or $l = \overline{x}$, that depends on x'. That would require that $\lambda(w_l) > \lambda(x') = \lambda(x)$, and so the universal reduction rule could not be applied. Serializing the universal blocks avoids this difficulty, without limiting the generality of the solver.

4.2 Clause Removal

As a dual proof proceeds, the BDD operations cause clauses to be added as extension variables are introduced and as inferences are made via resolution. Other clauses are added and removed explicitly by the proof steps, including the unit clauses for each term and the intermediate clauses generated by the steps. In order to support having the outcome of the solver be true, the defining and resolution clauses must be removed in order to ultimately end up with an empty set of clauses. The solver must justify their removal, since clause deletion is not, in general, equivalence preserving.

Clause removal is triggered when performing existential quantification, just before applying the variable elimination rule with variable x to remove clauses C_x and $C_{\overline{x}}$ (step 3). We must first ensure that there are no other clauses containing x or $\overline{x}$.

Our method is to remove any clause C containing a literal l for which $\lambda(l) > \lambda(x) = 2j - 1$. Clause removal can proceed by stepping through the clauses in the reverse order from how they were added. If a clause that was added by resolution contains a literal l with $\lambda(l) \geq 2j$, it can be removed via resolution deletion, using the same antecedent as was used when it was added.

Suppose the solver encounters the defining clauses for a node u with $\lambda(u) \geq 2j$. It can be certain that all clauses added by resolution that contain either u or $\overline{u}$ have already been removed, since these must have followed the introduction of u in the clause ordering. Similarly, any parent node v of u must have already had its defining clauses removed, since the defining clauses for v must occur after those for u. The existential elimination rule can therefore be used to remove the defining clauses for u.

Working through the set of clauses in reverse order, the solver may encounter clauses added by resolution and defining clauses containing only literals l with $\lambda(l) < 2j - 1$. These need not be removed, and indeed they can prove useful (clauses added by resolution) or necessary (some defining clauses) for subsequent proof steps. They will be deleted by clause removal during later phases.

We can see now why we impose the restriction that any quantifier block X_j with $Q_j = \exists$ contain only one variable. It enables the use of the λ values to determine which clauses should be removed to eliminate any dependency on existential variable x. Serializing the existential quantifier blocks allows this scheme to work without limiting the generality of the solver.

4.3 Specializing to Refutation or Satisfaction Proofs

Dual proofs have the advantage that they can be generated as a single pass, without knowing in advance whether the formula is true or false. On the other hand, they are, by necessity, somewhat longer and require more time to generate and to check. Another approach is to know (or guess) what the outcome will be and then direct the solver to generate a pure refutation or satisfaction proof. Specializing the proof generation to one of these forms is straightforward, and it can take advantage of more efficient ways to perform some of the quantifications.

A refutation proof need only justify that each step preserves truth. This enables several optimizations. Observe that deleting a clause always preserves truth, because it can only cause the set of satisfying solutions for the matrix to expand. Therefore clause deletion can be performed without any justification and instead be incorporated into the BDD garbage collection process [9]. Second, the BDD package need not generate the falsehood-preserving proofs shown in Table 2, reducing the number of clauses generated. Finally, the existential operation of (3) is inherently truth preserving. BDD packages can implement the quantification of a function by an entire set of variables via a variant of the Apply algorithm. If the quantification of root node u generates result node w, then the solver can run an implication test after the BDD computation has been performed to prove that $u \rightarrow w$, as is done with our SAT solver [9]. This avoids the need to serialize existential quantifier blocks and to have the solver generate low-level proof steps for each existential variable.

Conversely, a satisfaction proof need only justify that each step preserves falsehood. Adding a clause always preserves falsehood, since it can only reduce the set of satisfying solutions for the matrix, and therefore clause addition can be performed without any justification. In addition, the BDD package need not generate the truth-preserving proofs shown in Table 2. Finally, universal quantification can be performed on an entire block of variables producing node w from argument u. The solver can then run an implication test to generate a proof that $w \rightarrow u$.

5 Experimental Results

PGBDDQ[1] is written entirely in Python and consists of around 3350 lines of code, including a BDD package, support for generating extended-resolution proofs, and the overall QBF solver. By comparison, our proof-generating BDD-based SAT solver required around 2130 lines of code [9]. PGBDDQ can generate proofs in either the QRAT format [13, 14] or in a format we call QPROOF that supports just the proof rules given in Table 1. The latter format requires explicit lists of antecedents, and therefore each step can be checked without any search.

The overall control of PGBDDQ is based on a form of bucket elimination [10], where each quantifier block X_j defines a bucket. It starts by generating BDD representations of the input clauses. The resulting terms are inserted into buckets according to the value of $\lambda(u)$ for each root node u. As described in Section 3.2, this value will be $2j$ when u contains a variable from block X_j in its support, and it has no variables at higher quantification levels.

Processing proceeds from the highest numbered bucket downward. For a universal level, quantification is performed for each bucket element individually with the results placed into buckets according to their values for λ. For an existential level, the elements are conjuncted and then existential quantification is performed. The result is placed into a bucket according to its value of λ.

We can see that this approach defers conjunction as long as possible, only operating on terms at some quantification level j that truly depend on one or more variables in X_j. Similar techniques have been used in other BDD-based QBF solvers [23, 24]. However, other implementations place terms into buckets according to the BDD level of their root nodes, requiring the BDD variables to be ordered as the inverse of the quantification ordering. By labeling each node with its value of λ, we can determine the appropriate bucket from the root node without regard to the BDD variable ordering.

We have tested PGBDDQ on a number of scalable benchmark problems, finding it performs well in some cases, scaling polynomially, and poorly in others, scaling exponentially. Here we present results for a problem based on a two-player game. It provides insights into how polynomial scaling can be achieved, as well as the performance of the solver and two checkers.

Two-player games provide a rich set of benchmarks for QBF solvers, with each turn being translated into a quantification level. To encode the game from the perspective of the first player (Player A), A's turns are encoded with existential quantifiers, while the second player's (Player B) turns are encoded with universal quantifiers. The formula will be true if the game has a guaranteed winning strategy for A. The encoding of a game into QBF constrains the two players to only make legal moves. It also expresses the conditions under which A is the winner, namely that the game consist of t consecutive moves, for an odd value of t. Conversely, we can encode the formula where B has a winning strategy by reversing the quantifiers and expressing that the game must consist of an even number of consecutive moves. For a game where no draws are possible, these two formulas will be complementary.

[1] A demonstration version, complete with solver, checker, and benchmarks, is available at https://github.com/rebryant/pgbddq-artifact.

Consider a game played on a $1 \times N$ grid of squares with a set of dominos, each of which can cover two squares. Players alternate turns, each placing a domino to cover two adjacent squares. The game completes when no more moves are possible, taking at most $\lfloor N/2 \rfloor$ turns. The first player who cannot place a domino loses. This *linear domino placement* game is isomorphic to the object-removal game "Dawson's Kales" [2]. It can be shown that player B has a winning strategy for $N \in \{0, 1, 15, 35\}$ as well as for all values of the form $34\,i + c$ where $i \geq 0$ and $c \in \{5, 9, 21, 25, 29\}$ [27].

The game is encoded as a QBF by introducing a set of $N-1$ input variables for each possible move, each corresponding to the boundary between a pair of adjacent squares. A set of $N - 1$ Tseitin variables encodes the board state after each move, and sets of clauses enforce the conditions that 1) each move should cover exactly one boundary, and 2) neither that boundary nor the two adjacent ones should have been covered previously. In all, there are around $N^2/4$ universal input variables, $N^2/4$ existential input variables, and $3N^2/2$ Tseitin variables. The number of clauses grows as $\Theta(N^3)$ due to the quadratic number of clauses to enforce the exactly-one constraints on the input variables for each move.

To achieve polynomial performance, we found that several problem-specific techniques are required. First, the Tseitin variables for a given move are placed in an existential quantifier block immediately following the block for the input variables for the move. This is logically equivalent to the usual convention of placing all Tseitin variables in an innermost quantifier block, but it enables the bucket elimination algorithm to process the clauses for each move in sequence, rather than expanding the formulas in terms of only the input variables at the outset. Second, all variables are ordered for the BDD in "boundary-major" ordering. That is, all variables, including input and Tseitin variables, for the first boundary on the board are included from the first quantification level to the last. The variables for the second boundary follow similarly, and so on for all $N - 1$ boundaries. This ordering has the effect that, when processing the clauses for some move, the variables encoding the next, and previous state for a boundary, as well as the proposed change to its state, are localized within the ordering. Finally, when splitting a quantifier block into a series of single-variable blocks, we ordered them according to their BDD variable ordering. Since the solver eliminates variables in the reverse of their quantifier ordering, this convention causes the disjunction and conjunction operations of Equations (3) and (4) to be performed mainly on subgraphs of the BDD below the variables being quantified. This enables greater use of previously computed results via the operation cache.

Table 3 shows the performance of PGBDDQ, two checkers, and two other QBF solvers on the domino placement game as functions of N. It shows first cases where the encoded player has a winning strategy, and therefore the formula is true, and then cases where the encoded player's opponent has a winning strategy, and therefore the formula is false. Dual proofs were generated for both cases. For measurements with sufficient data points, we show the scaling trends, obtained by performing a linear regression on the logarithms of data generated for each value of N in increments of 5. All measurements were performed on a 4.2 GHz Intel Core i7 (I7-7700K) processor with 32 GB of memory running the MacOS operating system. Times are measured in elapsed seconds.

Table 3. Experimental Results for Dual Proof Generation with Linear Domino Placement Game. The first data series are for proofs of true formulas, and the second are for false formulas. Entries shown as "—" indicate cases where the program exceeded a 7200-second time limit.

N	Winner/	Input	PGBDDQ				Other solvers	
	Player	Clauses	Total Clauses	Solve	Qproof	QRAT-TRIM	DEPQBF	GHOSTQ
10	A/A	666	132,138	3.1	3.3	3.4	0.1	0.0
15	B/B	1,725	628,392	15.2	15.7	43.8	3.8	1.3
20	A/A	3,880	2,572,139	67.3	65.5	605.0	1896.6	57.9
25	B/B	6,637	7,098,146	202.6	199.5	4265.6	—	—
40	A/A	24,010	83,736,352	3358.6	3479.5	—	—	—
Trend		$N^{2.7}$	$N^{4.5}$	$N^{4.8}$	$N^{4.8}$			
10	A/B	664	132,403	3.1	3.2	7.3	0.1	0.0
15	B/A	1,728	629,530	15.2	15.5	108.7	3.6	1.0
20	A/B	3,885	2,580,284	67.2	66.7	1521.5	—	49.1
25	B/A	6,631	7,083,515	205.1	190.0	—	—	6942.2
40	A/B	24,000	83,662,168	3279.2	3457.4	—	—	—
Trend		$N^{2.7}$	$N^{4.5}$	$N^{4.8}$	$N^{4.8}$			

As indicated in the column labeled "Input Clauses," the number of clauses grows as $N^{2.7}$, not quite reaching the asymptotic value of N^3. The number of proof clauses generated by PGBDDQ are nearly the same for both true and false formulas, with growth rates of $N^{4.5}$. The time taken by the solver (labeled "Solve") , and by our own checker ("Qproof") scale at about the same rate as the number of proof clauses.

We also benchmarked the QBF proof checker QRAT-TRIM [13, 14]. This program was already equipped to handle our forms of refutation and satisfaction proofs, and it can handle dual proofs without modification. The only concession to the idiosyncrasies of PGBDDQ was to serialize the universal quantifier blocks in the prefix of false formulas. This is required to enable application of the universal reduction rule. The existential blocks can stay intact, since our only reason to serialize these is to guide the clause removal process. Although the scaling of QRAT-TRIM is poor, it is encouraging that the solver can be verified by a checker that predates it by a number of years.

For comparison, we evaluated the performance of two other QBF solvers on this benchmark: DEPQBF, version 6.0 [20], and GHOSTQ [15, 19]. We found they are both very fast for smaller values of N but then reach a narrow range of values for which they transition from running in just a few seconds to exceeding the timeout limit of 7200 seconds. For DEPQBF, this transition occurs as N ranges from 17 to 21, and for GHOSTQ, as N ranges from 21 to 26. PGBDDQ is much slower for small values of N, but it keeps scaling without hitting a sudden cutoff.

Although we did not run EBDDRES [17], we can use PGBDDQ to evaluate the impact of having the BDD variable ordering be the inverse of the quantifier ordering. Our experiments show that this ordering causes the runtime and proof sizes to scale exponentially in N. With $N = 14$ and B as the player, PGBDDQ runs for 4100 seconds to generate a refutation proof with 114,157,025 clauses. By contrast, a boundary-major ordering requires just 6 seconds and generates a proof with 309,387 clauses.

Table 4. Experimental Results for Specialized Proof Generation with Linear Domino Placement Game. The first data series are for satisfaction proofs, and the second are for refutation proofs.

N	Winner/Player	Input Clauses	Total Clauses	Solve	Qproof
10	A/A	666	90,924	1.8	1.4
20	A/A	3,880	1,516,756	36.4	24.0
30	A/A	11,166	10,466,168	346.0	192.6
40	A/A	24,010	44,874,662	1990.3	1254.8
45	A/A	32,241	74,891,554	4033.4	2760.8
Trend		$N^{2.7}$	$N^{4.4}$	$N^{4.8}$	$N^{4.7}$
10	A/B	664	126,127	2.4	1.8
20	A/B	3,885	1,232,252	27.3	18.6
30	A/B	11,159	7,084,367	180.0	121.6
40	A/B	24,010	26,150,238	773.9	565.0
50	A/B	43,904	85,077,630	2955.4	2151.4
Trend		$N^{2.7}$	$N^{4.0}$	$N^{4.3}$	$N^{4.3}$

Table 4 shows the advantage of generating specialized proofs when the formula is known in advance to be true or false. Comparing the columns labeled "Total Clauses" in Tables 3 and 4, we can see especially that refutation proofs are asymptotically shorter. These can take advantage of the more efficient approach to existential quantification in handling the large number of Tseitin variables. Again, the solution and checking time track the proof sizes. These optimizations allowed us to solve larger instances of the problem—up to $N = 45$ for true instances and $N = 50$ for false ones.

6 Conclusions

We have demonstrated that a QBF solver can emit a single proof as it operates, leading to either an empty clause for a false formula or an empty set of clauses for a true one. Both the proof and the time required to check it scale as the number of BDD operations performed. Moreover, a BDD-based QBF solver can allow the choice of BDD variable ordering to be made independently from the quantifier ordering. This feature can be critical to obtaining performance that scales polynomially with the problem size.

Our prototype is only a start in implementing a fully automated QBF solver. Such a solver must be able to choose a BDD variable ordering based on the input formula structure. It must also be able to identify and move Tseitin variables to earlier positions in the quantifier ordering, generating proof steps justifying that this transformation is equivalence preserving.

The underlying operation of PGBDDQ has potential applications beyond QBF solving. The program could stop the process described in Section 4.1 at any point and generate a QBF that is provably equivalent to the input formula. PGBDDQ could therefore be used as a preprocessor for other solvers, and for other applications that require reasoning about Boolean formulas with quantifiers.

Acknowledgements. The second author is supported by NSF grant CCF-2010951.

References

1. Andersen, H.R.: An introduction to binary decision diagrams. Tech. rep., Technical University of Denmark (October 1997)
2. Berlekamp, E.R., Conway, J.H., Guy, R.K.: Winning Ways for your Mathematical Plays: Volume 1, Second edition. CRC Press (2001)
3. Beyersdorff, O., Chew, L., Janota, M.: Extension variables in QBF resolution. In: AAAI Workshop on Beyond NP (2016)
4. Biere, A.: Resolve and expand. In: Theory and Applications of Satisfiability Testing (SAT). LNCS, vol. 3542, pp. 59–70 (2005)
5. Biere, A., Lonsing, F., Seidl, M.: Blocked clause elimination for QBF. In: Conference on Automated Deduction (CADE). LNCS, vol. 6803, pp. 101–115. Springer (2011)
6. Bryant, R.E.: Graph-based algorithms for Boolean function manipulation. IEEE Trans. Computers **35**(8), 677–691 (1986)
7. Bryant, R.E.: Symbolic Boolean manipulation with ordered binary decision diagrams. ACM Computing Surveys **24**(3), 293–318 (September 1992)
8. Bryant, R.E.: Binary decision diagrams. In: Clarke, E.M., Henzinger, T.A., Veith, H., Bloem, R. (eds.) Handbook of Model Checking, pp. 191–217. Springer (2018)
9. Bryant, R.E., Heule, M.J.H.: Generating extended resolution proofs with a BDD-based SAT solver. In: Tools and Algorithms for the Construction and Analysis of Systems (TACAS) (2021)
10. Dechter, R.: Bucket elimination: A unifying framework for reasoning. Artificial Intelligence **113**(1–2), 41–85 (1999)
11. Garey, M.R., Johnson, D.S.: Computers and Intractability: A Guide to the Theory of NP-Completeness. W. H. Freeman (1979)
12. Giunchiglia, E., Narizzano, M., Tacchella, A.: Clause/term resolution and learning in the evaluation of quantified Boolean formulas. Journal of AI Research **26**, 371–416 (2006)
13. Heule, M.J.H., Seidl, M., Biere, A.: A unified proof system for QBF preprocessing. In: International Joint Conference on Automated Reasoning (IJCAR). LNCS, vol. 8562, pp. 91–106 (2014)
14. Heule, M.J.H., Seidl, M., Biere, A.: Solution validation and extraction for QBF. Journal of Automated Reasoning **58**, 97–125 (2017)
15. Janota, M., Klieber, W., Marques-Silva, J.a.P., Clarke, E.M.: Solving QBF with counterexample guided refinement. Artificial Intelligence **234**, 1–25 (2016)
16. Jussila, T., Sinz, C., Biere, A.: Extended resolution proofs for symbolic SAT solving with quantification. In: Theory and Applications of Satisfiability Testing (SAT). LNCS, vol. 4121, pp. 54–60 (2006)
17. Jussila, T., Sinz, C., Biere, A., Kröning, D., Wintersteiger, C.M.: A first step towards a unified proof checker for QBF. In: Theory and Applications of Satisfiability Testing (SAT). LNCS, vol. 4501, pp. 201–714 (2007)
18. Kleine Büning, H., Karpinski, M., Flögel, A.: Resolution for quantified Boolean formulas. Information and Computation **117**(1), 12–18 (1995)
19. Klieber, W.: GhostQ system description. Journal on Satisfiability, Boolean Modeling, and Computation **11**, 65–72 (2019)
20. Lonsing, F., Egly, U.: DepQBF 6.0: A search-based QBF solver beyond traditional QCDCL. In: Conference on Automated Deduction (CADE). LNCS, vol. 10395, pp. 371–384 (2017)
21. Narizzano, M., Peschiera, C., Pulina, L., Tacchella, A.: Evaluating and certifying QBFs: A comparison of state-of-the-art tools. AI Communications **22**(4), 191–210 (2009)
22. Niemetz, A., Preiner, M., Lonsing, F., Seidl, M., Biere, A.: Resolution-based certificate extraction for QBF. In: Theory and Applications of Satisfiability Testing (SAT). LNCS, vol. 7317, pp. 430–435 (2012)

23. Olivo, O., Emerson, E.A.: A more efficient BDD-based QBF solver. In: Principles and Practice of Constraint Programming (CP). LNCS, vol. 6876, pp. 675–690 (2011)
24. Pan, G., Vardi, M.Y.: Symbolic decision procedures for QBF. In: Principles and Practice of Constraint Programming (CP). LNCS, vol. 3258, pp. 453–467 (2004)
25. Peitl, T., Slivovsky, F., Szeider, S.: Polynomial-time validation of QCDCL certificates. In: Theory and Applications of Satisfiability Testing (SAT). LNCS, vol. 10929, pp. 253–269 (2018)
26. Sinz, C., Biere, A.: Extended resolution proofs for conjoining BDDs. In: Computer Science Symposium in Russia (CSR). LNCS, vol. 3967, pp. 600–611 (2006)
27. Sloane, N.J.A., The OEIS Foundation: The on-line encyclopedia of integer sequences (2012), http://oeis.org/A215721, sequence A215721
28. Tseitin, G.S.: On the complexity of derivation in propositional calculus. In: Automation of Reasoning: 2: Classical Papers on Computational Logic 1967–1970. pp. 466–483. Springer (1983)