# Towards a Symmetric Treatment of Satisfaction and Conflicts in Quantified Boolean Formula Evaluation

Lintao Zhang, Sharad Malik

Department of Electrical Engineering, Princeton University, Princeton, NJ 08544
{lintaoz,sharad}@ee.princeton.edu

**Abstract.** In this paper, we describe a new framework for evaluating Quantified Boolean Formulas (QBF). The new framework is based on the Davis-Putnam (DPLL) search algorithm. In existing DPLL based QBF algorithms, the problem database is represented in Conjunctive Normal Form (CNF) as a set of clauses, implications are generated from these clauses, and backtracking in the search tree is chronological. In this work, we augment the basic DPLL algorithm with conflict driven learning as well as satisfiability directed implication and learning. In addition to the traditional clause database, we add a cube database to the data structure. We show that cubes can be used to generate satisfiability directed implications similar to conflict directed implications generated by the clauses. We show that in a QBF setting, conflicting leaves and satisfying leaves of the search tree both provide valuable information to the solver in a symmetric way. We have implemented our algorithm in the new QBF solver Quaffle. Experimental results show that for some test cases, satisfiability directed implication and learning significantly prunes the search.

## 1. Introduction

A Quantified Boolean Formula is a propositional logic formula with existential and universal quantifiers preceding it. Given a Quantified Boolean Formula, the question whether it is satisfiable (i.e. evaluates to 1) is called a Quantified Boolean Satisfiability problem (QBF). In the rest of the paper, we will use QBF to denote both the formula and the decision problem, with the meaning being clear from the context. Many practical problems ranging from AI planning [1] to sequential circuit verification [2] [3] can be transformed into QBF problems. QBF is P-Space Complete, thus placing it higher in the complexity hierarchy than NP-Complete problems. It is highly unlikely that there exists a polynomial time algorithm for QBF. However, because of its practical importance, there is interest in developing efficient algorithms that can solve many practical instances of QBF problems.

Research on QBF solvers has been going on for some time. In [4], the authors present a resolution-based algorithm and prove that it is complete and sound. In [5], the authors proposed another QBF evaluation method that is essentially a resolution-based procedure. Both of these resolution-based algorithms suffer from the problem of space explosion; therefore, they are not widely used as practical tools. Other efforts have resulted in some QBF solvers that will not blow up in space (e.g. [7] [8] [10]

[17]). These solvers are all based on variations of the Davis Logemann Loveland (sometimes called DPLL) algorithm [6]. Most of these methods can be regarded as a generalization of the algorithms commonly used to evaluate Boolean propositional formulas (called the Boolean Satisfiability Problem or SAT). SAT can be regarded as a restricted form of QBF. In SAT, only existential quantifiers are allowed. SAT differs from QBF in that when a satisfying assignment is found, the algorithm will stop, while QBF may need to continue the search because of the universal quantifiers. Because the above-mentioned QBF algorithms are by and large based on SAT algorithms (even though they may incorporate some QBF specific rules and heuristics), they all operate on a clause database, and use clauses to generate implications and conflicts. In SAT, a conflict is the source of more information for the future, while a satisfying assignment is the end of the search. As a result QBF solvers based on a SAT search inherit this characteristic and focus on conflicts for deriving further information for search progress. However, as far as QBF is concerned, a satisfying assignment is not the end of the search and, as we will show, can also be used to derive further information to drive the search. A symmetric treatment of satisfaction and conflict is highly desirable (and useful) and is the focus of this paper.

Due to its importance, significant research effort has been spent on finding fast algorithms for SAT. Recent years have seen major advances in SAT research, resulting in some very efficient complete SAT solvers (e.g. GRASP [12], SATO [14], rel_sat [13], Chaff [15]). These solvers are also based on the DPLL algorithm, and all of them employ conflict driven learning and non-chronological backtracking techniques (e.g. [12] [13]). Experiments shows that conflict driven learning is very effective in pruning the search space for structured (in contrast of random) SAT problems. Recently, Zhang and Malik [17] have developed method to incorporate Conflict Driven Learning in a QBF solver. Their experiments show that conflict driven learning, when adapted in a QBF solver, can speed up the search process greatly. However, just like other DPLL based QBF solvers mentioned above, the solver they have developed is not able to treat satisfiable leaves and conflicting leaves symmetrically. The solver has to use chronological backtracking on satisfying leaves.

In this paper, we describe our work that augments the solver described in [17]. Our framework is still based on the DPLL algorithm; therefore, it will not suffer from the memory explosion problem encountered by the resolution-based algorithms [4] [5]. We introduce the notion of Satisfiability Directed Implication and Learning, and show how to augment the widely used CNF database with *cubes* to make this possible. In our framework, the solver will operate on an *Augmented CNF* database. Because of this, our solver will have an almost symmetric (or dual) view for Satisfying Leaves as will as Conflicting Leaves encountered in the search.

A closely related work to this paper is presented by E. Giunchiglia *et al.* recently in [11]. In that paper, the authors demonstrate how to add backjumping into their QBF solving process. Our work differs from this in that we keep the knowledge from conflicts as learned clauses and the knowledge of satisfying branches as learned cubes. Backjumping (or sometimes called non-chronological backtracking) is a direct result of the learned clauses and cubes. Because of learning, the knowledge obtained from some search space can be utilized in other search spaces. In contrast, in [11] learning is not possible. Our framework also has the notion of satisfiability directed implication, which is not available in their work.

## 2. Problem Formulation

A QBF has the form

$$Q_1 x_1 \ldots\ldots Q_n x_n \; \varphi \qquad (1)$$

Where φ is a propositional formula involving propositional *variables* $x_i$ (i=1…n). Each $Q_i$ is either an existential quantifier ∃ or a universal quantifier ∀. Because ∃x∃y φ = ∃y∃x φ and ∀x∀y φ = ∀y∀x φ, we can always group the quantified variables into disjoint sets where each set consists of adjacent variables with the same type of quantifier. Therefore, we can rewrite (1) into the following form:

$$Q_1 X_1 \ldots\ldots Q_n X_n \; \varphi \qquad (2)$$

$X_i$'s are mutually disjoint sets of variables. Each variable in the formula must belong to one of these sets. We will call the variables existential or universal according to the quantifier of their respective quantification sets. Also, each variable has a quantification level associated with it. The variables belonging to the outermost quantification set have quantification level 1, and so on.

A *literal* is the occurrence of a variable in either positive or negative *phase*. A *clause* is a disjunction (logic *or*) of literals. A *cube* is a conjunction (logic *and*) of literals (this term is widely used in logic optimization, see e.g. [18]). In the rest of the paper, we will use concatenation to denote conjunction, and "+" to denote disjunction. A propositional formula φ is said to be in Conjunctive Normal Form (CNF) if the formula is a conjunction of clauses. When φ is expressed in CNF, the QBF becomes

$$Q_1 X_1 \ldots\ldots Q_n X_n \; C_1 \, C_2 \ldots C_m \qquad (3)$$

Here, the $C_i$'s are clauses. In the following, we will call the QBF in form (3) a QBF in Conjunctive Normal Form (CNF).

It is also possible to express a propositional formula φ in the Sum of Product (SOP) form, or sometimes called Disjunctive Normal Form (DNF). In that case, the formula is a disjunction of cubes. A QBF formula in DNF looks like:

$$Q_1 X_1 \ldots\ldots Q_n X_n \; (S_1 + S_2 + \ldots + S_{m'}) \qquad (4)$$

Here, the $S_i$'s are cubes. We will call a QBF in this form a QBF in Disjunctive Normal Form (DNF). CNF and DNF are not the only representations for propositional formulas and QBF. Suppose we have

$$\varphi = C_1 \ldots C_m \; = \; S_1 + S_2 + \ldots\ldots + S_{m'}$$

Then

$$\begin{aligned}
Q_1 X_1 \ldots\ldots Q_n X_n \; \varphi \; &= Q_1 X_1 \ldots\ldots Q_n X_n \; C_1 \, C_2 \ldots C_m \\
&= Q_1 X_1 \ldots\ldots Q_n X_n \; (S_1 + S_2 + \ldots\ldots + S_{m'}) \\
&= Q_1 X_1 \ldots\ldots Q_n X_n \; (C_1 \, C_2 \ldots C_m + S_1 + S_2 + \ldots\ldots + S_{m'}) \\
&= Q_1 X_1 \ldots\ldots Q_n X_n \; C_1 \, C_2 \ldots C_m (S_1 + S_2 + \ldots\ldots + S_{m'}) \\
&= Q_1 X_1 \ldots\ldots Q_n X_n \; (C_1 \, C_2 \ldots C_m + \Sigma \text{AnySubset}\{ S_1, S_2 \ldots, S_{m'}\}) \qquad (5) \\
&= Q_1 X_1 \ldots\ldots Q_n X_n \; (\Pi \text{AnySubset}\{ C_1 , C_2 \ldots, C_m\})(S_1 + S_2 + \ldots + S_{m'}) \qquad (6)
\end{aligned}$$

Here we use $\Sigma\omega$ to denote disjunction of elements in set $\omega$, and $\Pi\omega$ to denote conjunction of the elements in the set $\omega$. We use AnySubset($\omega$) to denote any set $\upsilon$ s.t. $\upsilon \subseteq \omega$. We will call the QBF in form (5) a QBF in *Augmented Conjunctive Normal Form (ACNF)* and QBF in form (6) a QBF in *Augmented Disjunctive Normal Form (ADNF)*. Because we will use ACNF extensively in our future discussion, we will define it here.

**Definition 1:** A Propositional formula $\varphi$ is said to be in *Augmented CNF (ACNF)* if

$$\varphi = C_1 C_2 \ldots C_m + S_1 + S_2 + \ldots\ldots + S_k$$

Where $C_i$'s are clauses, and $S_j$'s are cubes. Moreover, each $S_j$ is contained in the term $C_1\, C_2 \ldots C_m$. i.e.

$$\forall i \in \{1,2 \ldots k\}, \quad S_i \Rightarrow C_1\, C_2 \ldots C_m$$

A Quantified Boolean Formula in form (2) is said to be in *Augmented CNF* if the propositional formula $\varphi$ is in Augmented CNF. We will call all the conjunction of clauses $C_1\, C_2 \ldots C_m$ in the ACNF the *clause term*. By definition, in an ACNF all the cubes are contained in the clause term. Deleting any or all of the cubes will not change the propositional Boolean function $\varphi$ or the Quantified Boolean Function F.

Traditionally, QBF problems are usually presented to the solver in CNF. The QBF solver operates on a clause database that corresponds to the CNF clauses. All the theorems and deduction rules are valid under the assumption that the QBF is in CNF. In this paper, our discussion will concentrate on QBF in ACNF. CNF is a special case of ACNF. The conclusions that are drawn from QBF in ACNF will be applicable to QBF in CNF as well.

## 3. The QBF Solver Framework

### 3.1 Algorithm Overview

Our framework for solving QBF is based on the well-known Davis-Putnam-Logemann-Loveland (DPLL) algorithm. The DPLL procedure is a branch and search procedure on the variables. Therefore, in the rest of the paper, many of the statements will have the implicit "with regard to the current assignment of variable values" as a suffix. For example, when we say "the clause is conflicting", we mean that "the clause is conflicting in the context of the current variable assignments". We will omit this suffix for concise presentation when no confusion can result. The value assignment to the variables may be a partial assignment. We will call variables (and literals) that have not been assigned *free*. Each branch in the search has a *decision level* associated with it. The first branch variable has decision level 1, and so on. All of the variables implied by a decision variable will assume the same decision level as the decision variable. In the rest of the paper, we may use terms like "in the current branch". This has the same meaning as "in the partial variable assignment resulting from the implication of the current branching variables' assignments".

The top-level algorithm for our framework is described in Fig. 1. It is an iterative (instead of recursive) version of the DPLL algorithm similar to many Boolean SAT

solvers (e.g. [12] [15]) and many other QBF solvers (e.g. [7] [8] [10]). The difference between our framework and them is the actual meaning of each of the functions.

```
while(1) {
  decide_next_branch();
  while (true) {
    status = deduce();
    if (status == CONFLICT) {
      blevel = analyze_conflict();
      if (blevel == 0)
        return UNSAT;
      else backtrack(blevel);
    }
    else if (status == SATISFIABLE) {
      blevel = analyze_SAT()
      if (blevel == 0)
        return SAT;
      else backtrack(blevel);
    }
    else break;
  }
}
```

**Fig. 1.** The top level DPLL algorithm for QBF evaluation.

Unlike a regular SAT solver, the decision procedure `decide_next_branch()` in Fig. 1 needs to obey the quantification order. A variable can be chosen as a branch variable if and only if all variables that have smaller quantification levels are already assigned. This is similar to other QBF solvers.

Unlike other QBF solvers, the solver database is in ACNF; thus, the function `deduce()` will have different rules and can generate different implications. We will describe these rules in the following sections. The status of deduction can have three values: `UNDETERMINED`, `SATISFIABLE` or `CONFLICT`. The status is `SATISFIABLE (CONFLICT)` if we know that φ must evaluate to 1 (0) under the current partial variable assignment, otherwise, the status is `UNDETERMINED`. The purpose of function `deduce()` is to prune the search space. Therefore, any rules can be incorporated in the function without affecting the correctness of the algorithm as long as the rules are valid (e.g. the function will not return `SATISFIABLE` when the problem is `CONFLICT`, and vice-versa). Some algorithms use the unit literal rule [7], some may add pure literal rule [7] [10], and some add failed literal detection [8] and sampling [8] to `deduce()`. As long as the rules are valid, the algorithm is correct.

When deduction finds that the current branch is satisfiable, in Boolean SAT, the solver will return immediately with the satisfying assignment. In a QBF solver, because of the universal quantifiers, we need to make sure that both branches of a universal variable lead to a satisfiable solution. Therefore, the solver needs to backtrack and continue the search.

When the status of deduction is `CONFLICT`, we say that a conflicting leaf is reached. When the status of deduction is `SATISFIABLE`, we say that a satisfying leaf is reached. The functions `analyze_conflict()` and `analyze_SAT()` will analyze the current status and bring the search to a new space by backtracking (and possibly do some learning). The most simplistic DPLL algorithm will backtrack chronologically with no learning (see e.g. [17] for a description of non-chronological backtracking). Many QBF solvers, such as [7] [8], use this backtracking method.

In [11], the authors demonstrated a method for conflict-directed and satisfiability-directed non-chronological backjumping. In their approach, when the current assignment leads to a satisfying leaf or conflicting leaf, the reason for the result is constructed, and the solver will backjump to the decision level that is directly responsible for the conflicting or satisfiable leaf. However, the constructed reason will not be used to generate implications in future reasoning; thus learning is not possible. In [17], the authors demonstrated that conflict driven learning can be adapted and incorporated into QBF solvers. When a conflicting leaf is encountered, a learned clause is constructed and added to the clause database. The learned clause can be used in future search, thus enabling conflict driven learning. However, the algorithm is limited to chronological backtracking when satisfying leaves are encountered. In this work, we will show how to augment [17] by introducing satisfiability directed implication and learning. More specifically, we will keep the function `analyze_conflict()` in [17] intact and improve `analyze_SAT()`. When a satisfying leaf is encountered, `analyze_SAT()` will construct a *learned cube*, and add it to the database, which consists of both clauses and cubes that corresponding to an ACNF formula. The cubes may help search in the future, just as learned clauses do.

### 3.2 Motivation for Augmenting CNF with Cubes

Traditionally, for SAT, the DPLL algorithm requires that the problems be in CNF. The reason for that is because of the two important and useful rules that are direct results for formulas in CNF: the unit literal rule and the conflicting rule. The unit literal rule states that if a clause has only one free literal, then it must be assigned to value 1. The conflicting rule states that if a clause has all literals that evaluate to 0, then the current branch is not satisfiable. The function of these two rules is to direct the solver away from searching space with an obvious outcome. In the SAT case, the obvious outcome is that there is no solution in that space. For example, if there exists a conflicting clause, then any sub-space consistent with the current assignment will not have any solution in it, so we better backtrack immediately. If there exists a unit clause, we know that assigning the unit literal with value 0 will lead to a search space with no solution, so we better assign it 1. In SAT we are only interested in finding one solution, so we only need to prune the search space that has no solution. We call the implication by unit clauses *conflict directed implications* because the purpose of the implication is to avoid conflict (i.e. a no-solution space).

A QBF solver is different from SAT because it usually cannot stop when a single satisfying branch is found. In fact, it needs to search multiple combinations of assignments of the universal variables to declare satisfiability. Therefore, we are not only interested in pruning the space that obviously has no solution, we are also

interested in pruning the space that obviously *has* a solution. Most of the DPLL based QBF solvers are based on [7], which in turn is based on SAT procedures and requires the database in CNF. Even though the implication rule and conflicting rule of QBF is a little different from SAT, these rules are still *conflict directed*, i.e. they bring the search away from an obviously no-solution space. There is no mechanism in these algorithms to bring the search away from an obviously has-solution space.

To cope with this obvious asymmetry, we introduce the Augmented CNF in our framework. In ACNF, cubes are *or*-ed with the clause term. Whenever a cube is satisfied, the whole propositional formula evaluates to 1. Similar to unit clauses, we have the notion of *unit cubes.* A unit cube will generate an implication, but the implication's purpose is to bring the search away from space that obviously *has* solutions. Similar to conflicting clauses, we have the notion of *satisfying cubes*. Whenever a satisfying cube is encountered, we can declare the branch is satisfiable and backtrack immediately. We will describe the rules for the cubes in next sections.

Most frequently the QBF problem is presented to the solver in CNF form. Therefore, initially there is no cube in the database. We need to generate cubes during the solving process. ACNF requires that the generated cubes be contained in the clause term. Therefore, the satisfiability of the QBF will not be altered by these cubes.

### 3.3 Implication Rules

In this section, we will show the rules used in the `deduce()` function in Fig. 1. These rules are valid for QBF in ACNF forms. Therefore, they can be directly applied to the database the solver is working on.

A note on the notation used. We will use $C$, $C_1$, $C_2$ ... to represent clauses, $S$, $S_1$, $S_2$... to represent Cubes. We use $E(C)$, $E(S)$ for the set of existential literals in the clause and cube respectively, and $U(C)$, $U(S)$ for the set of universal literals in the clause and cube respectively. We use $a, b, c$... (letters appearing in the early part of the alphabet) to denote existential literals, and $x, y, z$ ... (letters appearing in the end of the alphabet) to denote universal literals. We use $V(a), V(b)$... to denote the value of the literals. If literal $a$ is free, $V(a) = X$. We use $L(a), L(b)$... to denote the quantification level of the variables corresponding to the literals.

**Definition 2.** A *tautology clause* is a clause that contains both a literal its complement. An *empty cube* is a cube that contains both a literal and its complement.

**Proposition 1.** Conflicting Rule for Non-Tautology Clause: For QBF F in ACNF, if in a certain branch, there exists a non-tautology clause $C$, s.t. $\forall a \in E(C), V(a) = 0$, and $\forall x \in U(C), V(x) \neq 1$, then F cannot be satisfied in the branch. We call such a clause a *conflicting clause*.

**Proposition 2.** Implication Rule for Non-Tautology Clause: For QBF F in ACNF, if in a certain branch, a non-tautology clause $C$ has literal $a$ s.t.

1. $a \in E(C), V(a) = X$. For any $b \in E(C), b \neq a; V(b)=0$.
2. $\forall x \in U(C), V(x) \neq 1$. If $V(x) = X$, then $L(x) > L(a)$

Then the formula F can be satisfied in the branch if and only if $V(a)=1$. We will call such a clause a *unit clause*, and the literal $a$ the *unit literal*. Notice that the unit literal

of a unit clause is always an existential literal. By this proposition, to avoid exploring an obviously no-solution space, we need to assign *a* with 1 to continue search.

**Proposition 3.** Satisfying Rule for Non-Empty Cube: For QBF F in ACNF, if in a certain branch, there exists a non-empty cube *S*, s.t. $\forall x \in U(S), V(x) = 1$, and $\forall a \in E(S), V(a) \neq 0$, then F is satisfied in the branch. We call such a cube a *satisfying cube*.

**Proposition 4.** Implication Rule for Non-Empty Cube: For QBF F in ACNF, if in a certain branch, a non-empty cube *S* has literal *x* s.t.

1. $x \in U(S), V(x) = X$. For any $y \in U(S), y \neq x$ then $V(y)=1$.
2. $\forall a \in E(S), V(a) \neq 0$. If $V(a) = X$, then $L(a) > L(x)$

Then the formula F is satisfied unless $V(a)=0$. We call such a cube a *unit cube*, and the literal *x* the *unit literal*. Notice that the unit literal of a unit cube is always a universal literal. Similar to Proposition 2, to avoid exploring an obviously has-solution space, we need to assign *x* with 0 to continue search.

Proposition 1 and 2 are the regular conflicting rule and implication rules for QBF if the database is in CNF (see, e.g. [7]). Because the cubes are redundant in ACNF, these rules will also apply for QBF in ACNF. Proposition 3 and 4 are exactly the dual of Proposition 1 and 2. When a unit literal in a clause or cube is forced to be assigned a value because of Proposition 2 or 4, we say that this literal (or the variable corresponding to it) is *implied*. The unit cube or clause where the unit literal is coming from is called the *antecedent* of the literal (or variable). The antecedent of an implied universal variable is a cube, and the antecedent of an implied existential variable is a clause. The implication rules corresponding to Proposition 2 and 4 can be used in the function `deduce()` in Fig. 1 for deduction. The pseudo code for it is listed in Fig. 2.

```
deduce() {
  while(problem_sat()==false &&
        num_conflicting_clause()==0) {
    if (exist_unit_clause())
      assign_unit_literal_in_clause_to_be_1();
    else if (exist_unit_cube())
      assign_unit_literal_in_cube_to_be_0();
    else
      return UNDETERMINED;
    }
  if (problem_sat()) return SAT;
  return CONFLICT;
}
```

**Fig. 2.** The deduce() function for both conflict and satisfiability directed implication

### 3.4 Generating Satisfiability-Induced Cubes

In this section, we will discuss how to generate cubes that are contained by the clause terms in an ACNF database. When the QBF problem is given to the solver, it usually

is in CNF and does not have any cubes. To generate cubes that are contained by the clause term, one obvious way is to expand the clause term by the distribution law. Unfortunately, this is not practical since the number of cubes generated is intractable.

Here, we will discuss another way to generate cubes. The main idea is that whenever the search procedure finds that all the clauses are satisfied (i.e. for each clause, at least one literal evaluates to 1), we can always find a set of value 1 literals such that for any clause, at least one of the literals in the set appears in it. We will call such a set a *cover set* of the satisfying assignment. The conjunction of the literals in the cover set is a cube, and this cube is guaranteed to be contained by the clause term.

For example, consider the ACNF:

`(a + b + x)(c + y')(a + b' + y')(a + x' + y') + xy'`

The variable assignments of {`a=1, b=0, c=X, x=0, y=0`} is a satisfying assignment. The set of literals {`a, y'`} is a cover set. Therefore, cube `ay'` is a cube that is contained in the clause term, and can be added to the ACNF. The resulting formula will be:

`(a + b + x)(c + y')(a + b' + y')(a + x' + y')+ ay'+ xy'`

We will call the cube generated from a cover set of a satisfying assignment a satisfiability-induced cube.

For a satisfying assignment, the cover set is not unique. Therefore, we can generate many satisfiability-induced cubes. Which and how many of these cubes should be added to the database is to be determined by heuristics. Different heuristics, while not affecting the correctness of the algorithm, may affect the efficiency. Evaluating different heuristics for generating satisfiability-induced cubes is beyond the scope of this paper. Here we will simply assume that we have some heuristics (for example, a greedy heuristic) to choose a single covering set for each satisfying assignment.

### 3.5 Conflict-Driven and Satisfiability-Directed Learning

Conflict driven learning in QBF was introduced in [17]. Here we will briefly review it for completeness of discussion. Conflict driven learning occurs when a conflicting leaf is encountered. The pseudo-code for analyzing the conflict as well as generating the learned clauses is shown in Fig. 3. The learning is performed by the function `add_clause_to_database()`. In this function, the solver can throw away all the universal literals that have a higher quantification level than any existential literal in the clause, as pointed out in [4]. The learned clause is generated by the function `resolution_gen_clause()`. Routine `choose_literal()` will choose an implied *existential* variable from the input clause in the reverse chronological order (i.e. variable implied last will be chosen first). Routine `resolve(cl1,cl2,var)` will return a clause that has all the literals appearing in *cl1* and *cl2* except for the literals corresponding to variable *var*. If the generated clause meets some predefined stopping criterion, the resulting clause will be returned, otherwise the resolution process is called recursively. The stopping criterion is that the clause satisfies:

1. Among all its existential variables, one and only one of them has the highest decision level. Suppose this variable is V.
2. V is in a decision level with an existential variable as the decision variable.

3. All universal literals with a quantification level smaller than V's are assigned 0 before V's decision level.

If these criteria are met, after backtracking to a certain decision level (determined by function `clause_asserting_level()`),, this clause will be a unit clause and force the unit literal to assume a different value, and bring the search to a new space. For more details about conflict driven learning in QBF, we refer the readers to [17].

**Proposition 5.** The learning procedure depicted in Fig. 3 will generate valid clauses that can be added to the database even when the QBF is in ACNF. Moreover, the learned clauses will obey the same implication rule and conflicting rule as regular non-tautology clauses even though some of the learned clauses may be tautologies, i.e. contain universal literals in both the positive and negative phases.

```
resolution_gen_clause( cl ) {
  lit = choose_literal (cl);
  var = variable_of_literal( lit );
  ante = antecedent( var );
  new_cl = resolve(cl, ante, var);
  if (stop_criterion_met(new_cl))
    return new_cl;
  else
    return resolution_gen_clause(new_cl);
}
analyze_conflict(){
  conf_cl = find_conflicting_clause();
  new_cl = resolution_gen_clause(conf_cl);
  add_clause_to_database(new_cl);
  back_dl = clause_asserting_level(new_cl);
  return back_dl;
}
```

**Fig. 3.** Generating Learned Clause by Resolution

When a satisfying leaf is encountered, similar to conflict driven learning, we can also perform satisfiability directed learning. Conflict driven learning adds (redundant) clauses into the ACNF database; similarly, satisfiability directed learning adds (redundant) cubes into the ACNF database. The procedure for satisfiability directed learning, which is shown in Fig. 4, is very similar to the procedure for conflict driven learning. The only major difference is that when a satisfiable leaf is encountered, there are two scenarios, while in conflicting case there is only one. The first scenario in the satisfying leaf case is that there exists a satisfying cube in the ACNF; this is similar to the conflicting case, where there exists a conflicting clause. The second scenario, which is unique in the satisfying case, is that all the clauses in the ACNF are satisfied (i.e. every clause has at least one literal evaluate to 1) but no satisfying cube exists. In Fig. 4, if function `find_sat_cube()` returns NULL, then the second case is encountered. In that case, we have to construct a satisfiability-induced cube from the current variable assignment. The learned cube is generated by the function `consensus_gen_cube()`. Routine `choose_literal()` will choose an implied universal variable from the input clause in the reverse chronological order. Routine `consensus(S1,S2,var)` will return a cube that has all the literals

appearing in S1 and S2 except for the literals corresponding to variable var. If the generated cube meets the following conditions, the recursion will stop:

1. Among all its universal variables, one and only one of them has the highest decision level. Suppose this variable is V.
2. V is at a decision level with a universal variable as the decision variable.
3. All existential literals with quantification level smaller than V's are assigned 1 before V's decision level.

If these criteria are met, the resulting cube will have only one universal literal at the highest decision level. After backtracking to a certain decision level (determined by function cube_asserting_level()), this cube will be a unit cube and will force this literal to assume a different value, and bring search to a new space.

```
consensus gen cube( s ) {
  lit = choose_literal (s);
  var = variable_of_literal( lit );
  ante = antecedent( var );
  new_cube = resolve(s, ante, var);
  if (stop_criterion_met(s))
    return new_cube;
  else
    return consensus_gen_cube(new_cube);
}

analyze_SAT(){
  cube = find_sat_cube();
  if (cube == NULL)
    cube = construct_sat_induced_cube();
  if (!stop_criterion_met(cube))
    cube = consensus_gen_cube(cube);
  add_cube_to_database(cube);
  back_dl = cube_asserting_level(cube);
  return back_dl;
}
```

**Fig. 4.** Generating Learned Cube

**Proposition 6.** The learning procedure depicted in Fig. 4 will generate valid cubes that are contained by the clause term of the ACNF and can be added to the database. Moreover, the learned cubes will obey the same implication rule and conflicting rule as non-empty cubes even though some of the learned cubes may contain an existential literal in both positive and negative phases.

From the pseudo-code of analyze_conflict() and analyze_SAT() we can see that the DPLL procedure will have an almost symmetric view on satisfying and conflicting leaves in the search tree. Whenever a conflicting leaf is encountered, a clause will be learned to prune the space that obviously has *no* solution. When a satisfying leaf is encountered, a cube will be learned to prune the space that obviously *has* solutions. The only asymmetry that exists is that in our database, the cubes are contained by the clause terms, but not vice-versa. Therefore, the cubes only contain partial information about the propositional formula. Because of this, we may need to

generate a satisfiability-induced cube when a satisfying assignment is found but no existing cube is satisfied. On the other hand, the formula need not be in ACNF for QBF. If the original QBF problem is given in DNF form (problem in DNF is trivial for SAT), we may augment it into ADNF. In that case, we need to construct conflict-induced clauses.

## 4. Experimental Results

We implemented the algorithm described in this paper in the new QBF solver Quaffle, which was first described in [17] to demonstrate the power of conflict driven learning in a QBF environment. We have improved the original Quaffle with a cube database such that the data structure corresponds to an ACNF. We incorporated the new rules on cubes (i.e. Proposition 3 and 4) into the `deduce()` function shown in Fig. 1. We also implemented code corresponding to Fig. 4 in place of `analyze_SAT()`. Because of these improvements, the solver now has the ability to do satisfiability directed implication and learning.

The heuristic we use for generating a cover set from a satisfying assignment is a simple greedy method to minimize the number of universal variables in the set. The decision heuristic we used to decide on the next branching variable is VSIDS [15], with quantification order of the variables being observed. We do not delete learned cubes or learned clauses in all the runs because the benchmarks we tested are mostly time-limited. Both learned clauses and cubes can be deleted in a similar manner as deletion of learned clauses in SAT if memory is limited.

The problem set was obtained from J. Rintanen [19]. We have already reported the performance comparison of Quaffle with other state-of-the-art QBF solvers in [17]. Therefore, in this paper we will only show two versions of Quaffle. One version of Quaffle has satisfiability directed implication and learning turned off. This is the version we reported in [17], we call it Quaffle-CDL to be consistent with our previous work. The other version has satisfiability directed implication and learning turned on. We call this version Quaffle-FULL. All the tests were conducted on a PIII 933 machine with 1G memory. The timeout limit is 1800 seconds for each instance.

Table 1 showed the run time data for the benchmarks (except the ones that cannot be solved by both versions within time limit). From the result table we can see that for some classes of benchmarks such as `impl` and random 3-QBF `R3...`, Quaffle with satisfiability directed implication and learning is faster when compared with Quaffle with no satisfiability directed implication and learning. For some other benchmarks such as the `CHAIN` and `TOILET` sets, the result is not really good. For others such as `BLOCKS` and `logn`, the results are mixed. To get a better understanding of the performance gain and loss, we show some of the detailed statistics in Table 2.

From Table 2 we get some additional insight for the performance difference between Quaffle-CDL and Quaffle-Full. In testcases that have few satisfiable leaves such as `logn...B1, logn...A2, BLOCKS4ii.6.3`, Quaffle-Full take about the same time or just a little bit more than Quaffle-CDL because the satisfiability induced pruning does not have many chances to work. For problem class `CHAIN` and

**Table 1.** Run time of Quaffle with Satisfiability driven implication turned on and off (time unit is second, we use – to denote timeout)

| Testcase | nVar | nCl | T/F | Quaffle-CDL | Quaffle-FULL | Testcase | nVar | nCl | T/F | Quaffle-CDL | Quaffle-FULL |
|---|---|---|---|---|---|---|---|---|---|---|---|
| BLOCKS3i.4.4 | 288 | 2928 | F | 0.07 | 0.09 | impl20 | 82 | 162 | T | 15.51 | 0.02 |
| BLOCKS3i.5.3 | 286 | 2892 | F | 29.03 | 103.73 | logn...A0 | 828 | 1685 | F | 0 | 0 |
| BLOCKS3i.5.4 | 328 | 3852 | T | 2.88 | 146.54 | logn...A1 | 1099 | 62820 | F | 2.21 | 2.14 |
| BLOCKS3ii.4.3 | 247 | 2533 | F | 0.05 | 0.04 | logn...A2 | 1370 | 65592 | T | 125.85 | 193.88 |
| BLOCKS3ii.5.2 | 282 | 2707 | F | 0.13 | 0.48 | logn...B0 | 1474 | 3141 | F | 0 | 0 |
| BLOCKS3ii.5.3 | 304 | 3402 | T | 0.33 | 0.48 | logn...B1 | 1871 | 178750 | F | 8.26 | 8.18 |
| BLOCKS3iii.4 | 202 | 1433 | F | 0.03 | 0.03 | logn...B2 | 2268 | 183601 | F | 763.37 | 750.92 |
| BLOCKS3iii.5 | 256 | 1835 | T | 0.27 | 0.23 | R3...3...50_0.T | 150 | 375 | T | 1.22 | 0.02 |
| BLOCKS4i.6.4 | 779 | 15872 | F | 249.09 | 110.2 | R3...3...50_1.F | 150 | 375 | F | 0.02 | 0.05 |
| BLOCKS4ii.6.3 | 838 | 15061 | F | 367.54 | 591.95 | R3...3...50_2.T | 150 | 375 | T | 0.81 | 0.01 |
| BLOCKS4iii.6 | 727 | 9661 | F | 39.33 | 294.49 | R3...3...50_3.T | 150 | 375 | T | 1.06 | 0 |
| CHAIN12v.13 | 925 | 4582 | T | 0.31 | 7.11 | R3...3...50_4.T | 150 | 375 | T | 1.43 | 0.09 |
| CHAIN13v.14 | 1080 | 5458 | T | 0.66 | 19.09 | R3...3...50_5.T | 150 | 375 | T | 0.95 | 0.06 |
| CHAIN14v.15 | 1247 | 6424 | T | 1.45 | 51.09 | R3...3...50_6.F | 150 | 375 | F | 1.51 | 0.37 |
| CHAIN15v.16 | 1426 | 7483 | T | 3.15 | 142.21 | R3...3...50_7.F | 150 | 375 | F | 0.6 | 0.07 |
| CHAIN16v.17 | 1617 | 8638 | T | 6.82 | 472.38 | R3...3...50_8.F | 150 | 375 | F | 0.29 | 0.05 |
| CHAIN17v.18 | 1820 | 9892 | T | 14.85 | 1794.35 | R3...3...50_9.T | 150 | 375 | T | 0.87 | 0.02 |
| CHAIN18v.19 | 2035 | 11248 | T | 32.4 | - | R3...7...60_0.F | 150 | 390 | F | 0.14 | 0.11 |
| CHAIN19v.20 | 2262 | 12709 | T | 71.41 | - | R3...7...60_1.T | 150 | 390 | T | 0.23 | 0.02 |
| CHAIN20v.21 | 2501 | 14278 | T | 154.86 | - | R3...7...60_2.T | 150 | 390 | T | 1.27 | 0.02 |
| CHAIN21v.22 | 2752 | 15958 | T | 343.62 | - | R3...7...60_3.T | 150 | 390 | T | 0.34 | 0.02 |
| CHAIN22v.23 | 3015 | 17752 | T | 747.3 | - | R3...7...60_4.T | 150 | 390 | T | 13.3 | 0.17 |
| CHAIN23v.24 | 3290 | 19663 | T | 1710.06 | - | R3...7...60_5.F | 150 | 390 | F | 1.3 | 0.11 |
| impl02 | 10 | 18 | T | 0 | 0 | R3...7...60_6.T | 150 | 390 | T | 0.51 | 0.03 |
| impl04 | 18 | 34 | T | 0 | 0 | R3...7...60_7.T | 150 | 390 | T | 2.21 | 0.33 |
| impl06 | 26 | 50 | T | 0 | 0 | R3...7...60_8.F | 150 | 390 | F | 0 | 0 |
| impl08 | 34 | 66 | T | 0.01 | 0.01 | R3...7...60_9.T | 150 | 390 | T | 0.23 | 0.02 |
| impl10 | 42 | 82 | T | 0.02 | 0.01 | TOILET02.1.iv.3 | 28 | 70 | F | 0 | 0 |
| impl12 | 50 | 98 | T | 0.06 | 0.01 | TOILET02.1.iv.4 | 37 | 99 | T | 0 | 0 |
| impl14 | 58 | 114 | T | 0.24 | 0.02 | TOILET06.1.iv.11 | 294 | 1046 | F | 39.51 | 221.45 |
| impl16 | 66 | 130 | T | 0.97 | 0.02 | TOILET06.1.iv.12 | 321 | 1144 | T | 18.23 | 74.16 |
| impl18 | 74 | 146 | T | 3.88 | 0.02 | | | | | | |

TOILET, though there exist many satisfying leaves, satisfiability induced learning is not able to prune much of the search space. The reason for this is because these testcases all have the property that when a satisfying assignment is found, the satisfiability-induced cover set often includes all of the universal literals. Because of this, the learned cubes will not be able to prune any search space (similar to very long conflict clause in SAT). For testcases R3... and impl, satisfiability directed implication and learning dramatically reduced the number of satisfying leaves need to be visited, therefore, the total run time is reduced significantly.

From the experimental results we find that satisfiability directed implication and learning can help prune the search space for some benchmarks but only induce overhead without much help for other benchmarks. Therefore, the question is when to apply it, and how to reduce the overhead when no pruning of search is possible. Currently, publicly available QBF benchmarks are very scarce, and very few of them are actually derived from real world problems. It is not easy to evaluate the applicability of any heuristic when test cases are limited.

**Table 2.** Some detailed statistics of the representative testcases

| Testcase | T/F | Quaffle-CDL | | | Quaffle-Full | | |
|---|---|---|---|---|---|---|---|
| | | No. Sat. Leaves | No. Conf. Leaves | Runtime | No. Sat. Leaves | No. Conf. Leaves | Runtime |
| TOILET06.1.iv.12 | F | 24119 | 7212 | 18.23 | 17757 | 8414 | 74.16 |
| TOILET06.1.iv.11 | T | 30553 | 11000 | 39.51 | 30419 | 13918 | 221.45 |
| CHAIN15v.16 | T | 32768 | 43 | 3.15 | 32768 | 43 | 142.21 |
| CHAIN16v.17 | T | 65536 | 46 | 6.82 | 65536 | 46 | 472.38 |
| CHAIN17v.18 | T | 131072 | 49 | 14.85 | 131072 | 49 | 1794.35 |
| impl16 | T | 160187 | 17 | 0.97 | 106 | 17 | 0.02 |
| impl18 | T | 640783 | 19 | 3.88 | 124 | 19 | 0.02 |
| impl20 | T | 2563171 | 21 | 15.51 | 142 | 21 | 0.02 |
| R3...3...50_8.F | F | 11845 | 374 | 0.29 | 59 | 460 | 0.05 |
| R3...3...50_9.T | T | 33224 | 87 | 0.87 | 35 | 50 | 0.02 |
| logn...A2 | T | 3119 | 11559 | 125.85 | 1937 | 14428 | 193.88 |
| logn...B1 | F | 2 | 601 | 8.26 | 2 | 609 | 8.18 |
| BLOCKS4ii.6.3 | F | 5723 | 52757 | 367.54 | 98 | 45788 | 591.95 |

The overhead of satisfiability directed implication and learning mainly comes from two places. The first overhead is that the added cubes will slow down the implication process. This overhead can be reduced by an intelligent clause and cube deletion heuristic. The other overhead arises from generating the learned cubes. This overhead is tightly related to the heuristic to generate the satisfiability-induced cover set, which in turn affects the quality of the generated cube. Determining an effective cover set without introducing a large overhead is an interesting research question.

## 5. Additional Notes

An anonymous reviewer pointed out that two independent in-submission papers [20] and [21] reached results similar to this work. We briefly review them here. In [20], the author proposed model and lemma caching similar to learning in our work, and dependency-directed backtracking (i.e. non-chronological backtracking in this work). However, it does not have a clean way to deal with tautology clauses and empty cubes, which is an important feature of our framework [17]. Moreover, unlike our work (as well as related results in the SAT domain), learning and non-chronological backtracking are not coupled in [20]. In [21], the authors pointed out that "good" solutions should be represented in DNF form and use a separate set of specially marked clauses to perform the same functions as the cubes in our work do. [21] also has the concept of conflict driven learning. However, their work is not resolution and consensus based, therefore require some special treatment for the assignments (i.e. pre-fix closed) to be able to construct valid reasons.

## 6. Conclusions

In this paper, we introduce the notion of satisfiability directed implication and learning and show how to incorporate it in a solver framework. In our new

framework, the QBF solver works on an Augmented CNF database instead of the traditional CNF database. This enables the solver to have an almost symmetric view of both satisfied leaves and conflicting leaves in the search tree. Implications in the new framework not only prune search spaces with no solution, but also prune search spaces with solutions. We have implemented our idea in the new QBF solver Quaffle. Experiments show that Quaffle with satisfiability directed implication and learning can help prune search for many instances.

## 7. References


[1] J. Rintanen. Constructing conditional plans by a theorem prover. *Journal of Artificial Intelligence Research, 10:323-352, 1999*

[2] M. Sheeran, S. Singh, G. Stälmark, Checking Safety Properties Using Induction and a SAT-Solver, in *Proceedings of FMCAD, 2000*

[3] A. Biere, A. Cimatti, E. M. Clarke, and Y. Zhu. Symbolic Model Checking without BDDs, In *Tools and Algorithms for the Analysis and Construction of Systems (TACAS), 1999*

[4] H. Kleine-Büning, M. Karpinski and A. Flögel. Resolution for quantified Boolean formulas. In *Information and Computation, 117(1):12-18, 1995*

[5] D. A. Plaisted, A. Biere and Y. Zhu. A Satisfiability Procedure for Quantified Boolean Formulae, To appear in, *Discrete Applied Mathematics*

[6] M. Davis, G. Logemann, and D. Loveland. A machine program for theorem proving. In *Communications of the ACM*, *5:394-397, 1962*

[7] M. Cadoli, M. Schaerf, A. Giovanardi and M. Giovanardi. An algorithm to evaluate quantified Boolean formulae and its experimental evaluation, in *Highlights of Satisfiability Research in the Year 2000, IOS Press, 2000*

[8] J. Rintanen, Improvements to the Evaluation of Quantified Boolean Formulae, in *Proceedings of International Joint Conference on Artificial Intelligence (IJCAI), 1999*

[9] J. Rintanen, Partial implicit unfolding in the Davis-Putnam procedure for quantified Boolean formulae, in *International Conf. on Logic for Programming, Artificial Intelligence and Reasoning (LPAR), 2001*

[10] E. Giunchiglia, M. Narizzano and A. Tacchella,. Qube: a system for Deciding Quantified Boolean Formulas Satisfiability,. In *Proc. of International Joint Conf. on Automated Reasoning (IJCAR), 2001*

[11] E. Giunchiglia, M. Narizzano and A. Tacchella. Backjumping for Quantified Boolean Logic Satisfiability. In *Proc. of International Joint Conf. on Artificial Intelligence (IJCAI), 2001*

[12] João P. Marques-Silva and Karem A. Sakallah, "GRASP: A Search Algorithm for Propositional Satisfiability, In *IEEE Transactions on Computers*, *vol. 48, 506-521, 1999*

[13] R. Bayard and R. Schrag. Using CSP look-back techniques to solve real-world SAT instances, in *Proc. of the 14th Nat. (US) Conf. on Artificial Intelligence (AAAI), 1997*

[14] H. Zhang. SATO: An efficient propositional prover, In *Proc. of the International Conference on Automated Deduction, 1997*

[15] M. Moskewicz, C. Madigan, Y. Zhao, L. Zhang, and S. Malik. Engineering an efficient SAT Solver, In *Proceedings of the Design Automation Conference, 2001*

[16] L. Zhang, C. Madigan, M. Moskewicz, S. Malik, Efficient Conflict Driven Learning in a Boolean Satisfiability Solver, in *Proc. of International Conference on Computer Aided Design (ICCAD), 2001*

[17] L. Zhang and S. Malik, Conflict Driven Learning in a Quantified Boolean Satisfiability Solver, Accepted for publication, *International Conference on Computer Aided Design (ICCAD), 2002.*

[18] G. Hachtel and F. Somenzi, *Logic Sysntheiss and Verification Algorithms*: Kluwer Academic Publishers, 1996.

[19] J. Rintanen's benchmarks are at http://ww.informatik.uni-freiburg.de/~rintanen/qbf.html

[20] R. Letz, Lemma, Model Caching in Decision Procedures for Quantified Boolean Formulas, in *Proc. International Conf. on Automated Reasoning with Analytic Tableaux and Related Methods, 2002*

[21] E. Giunchiglia, M. Narizzano and A. Tacchella, Learning for Quantified Boolean Logic Satisfiability, in *Proc. of the 18th Nat. (US) Conf. on Artificial Intelligence (AAAI), 2002*