Monotone Literals and Learning in QBF reasoning

Enrico Giunchiglia, Massimo Narizzano, and Armando Tacchella

DIST, Università di Genova, Viale Causa, 13 – 16145 Genova, Italy
{enrico,mox,tac}@dist.unige.it

Abstract. Monotone literal fixing (MLF) and learning are well-known lookahead and lookback mechanisms in propositional satisfiability (SAT) . When considering Quantified Boolean Formulas (QBFs), their separate implementation leads to significant speed-ups in state-of-the-art DPLL-based solvers.
This paper is dedicated to the efficient implementation of MLF in a QBF solver with learning. The interaction between MLF and learning is far from being obvious, and it poses some nontrivial questions about both the detection and the propagation of monotone literals. Complications arise from the presence of learned constraints, and are related to the question about whether learned constraints have to be considered or not during the detection and/or propagation of monotone literals. In the paper we answer to this question both from a theoretical and from a practical point of view. We discuss the advantages and the disadvantages of various solutions, and show that our solution of choice, implemented in our solver QUBE, produces significant speed-ups in most cases. Finally, we show that MLF can be fundamental also for solving some SAT instances, taken from the 2002 SAT solvers competition.

1 Introduction and Motivations

Monotone literal fixing (MLF) and learning are well-known lookahead and lookback mechanisms in propositional satisfiability (SAT). When considering Quantified Boolean Formulas (QBFs), their separate implementation leads to significant speed-ups in state-of-the-art DPLL-based solvers (see, e.g., [1, 2]). This is witnessed by the plots in Figure 1, which show the performances of our solver QUBE-BJ running with/without MLF (Figure 1, left), and with/without learning (Figure 1, right). The test set used for the plots is the same selection of 450 real-world instances from the 2003 QBF solvers evaluation [3] where QUBE-BJ, a version of QUBE [4] with conflict and solution backjumping [5], was one of the top performers. In Figure 1, each solid-fill dot represents an instance, QUBE-BJ solving time is on the x-axis (seconds, log scale), QUBE-BJ(P) (Figure 1, left), and QUBE-LRN (Figure 1, right) solving times are on the y-axes (seconds, log scale). The diagonal (outlined boxes) represents the solving time of QUBE-BJ against itself and serves the purpose of reference: the dots above the diagonal are instances where QUBE-BJ performs better than the version with MLF (QUBE-BJ(P)) or the version with learning (QUBE-LRN), while the dots below are the instances where QUBE-BJ performances are worse than its siblings. By looking at Figure 1, we can see that both QUBE-BJ(P) and QUBE-LRN outperform QUBE-BJ. Considering the number of instances solved both by QUBE-BJ and each of its siblings, on 228/346

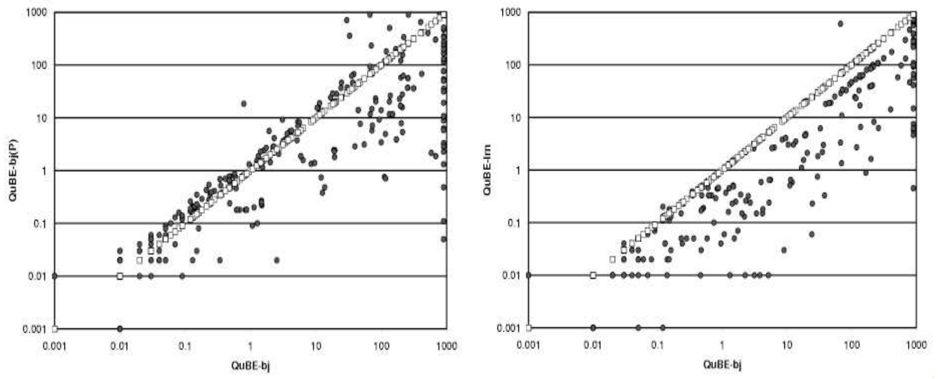

Fig. 1. Effectiveness of MLF and learning in a QBF solver.

(resp. 331/345) instances QuBE-BJ(P) (resp. QuBE-LRN) is as fast as or faster than QuBE-BJ: QuBE-BJ(P) (resp. QuBE-LRN) is at least one order of magnitude faster than QuBE-BJ on 37 (resp. 63) problems, while it is at least one order of magnitude slower on 10 (resp. 6) problems only.

This paper is dedicated to the efficient implementation of MLF in a QBF solver with learning. The interaction between MLF and learning is far from being obvious, and it poses some nontrivial questions about both the detection and the propagation of monotone literals. Indeed, the constraints introduced by learning during the search are redundant, i.e., they could be ignored without affecting the satisfiability of the formula and the correctness of the procedure, but they can substantially speed up the search. The main issue with MLF is that – similarly to what happens in SAT – in QBF solvers learned constraints can greatly outnumber the constraints in the input formula, and this may result in efficiency losses: given a set S of constraints, the detection of a monotone literal l has a cost which is at least linear in the number of occurrences of l in S. If, as in the case of learning, the number of constraints under consideration is increased (substantially), the performances of MLF decrease (significantly). This opens up to the question about whether it is necessary or not to consider learned constraints during the detection and/or propagation of monotone literals. In the paper we answer to this question both from a theoretical and from a practical point of view. We discuss the advantages and the disadvantages of the various solutions, and show that our solution of choice, implemented in our solver QuBE, produces significant speed-ups in most cases. Finally, we show that MLF can be fundamental also for solving a subset of the SAT instances used in the 2002 SAT solvers competition.

The paper is structured as follows. We first review the basics of QBF satisfiability and DPLL based solvers with learning (Section 2). We then discuss the issues related to the implementation of MLF in such solvers (Section 3), and present the possible solutions (Section 4). We end the paper with the experimental analysis (Section 5), the conclusions and the related work (Section 6).

2 Formal Preliminaries

Consider a set P of symbols. A *variable* is an element of P. A *literal* is a variable or the negation of a variable. In the following, for any literal l, $|l|$ is the variable occurring in l; and $\bar{l}$ is the negation of l if l is a variable, and is $|l|$ otherwise. For the sake of simplicity, we consider only formulas in negation normal form (NNF). Thus, for us, a *propositional formula* is a combination of literals using the k-ary ($k \geq 0$) connectives $\wedge$ (for conjunctions) and $\vee$ (for disjunctions). In the following, we use TRUE and FALSE as abbreviations for the empty conjunction and the empty disjunction respectively.

A *QBF* is an expression of the form

$$\varphi = Q_1 z_1 Q_2 z_2 \ldots Q_n z_n \Phi \qquad (n \geq 0) \tag{1}$$

where every Q_i $(1 \leq i \leq n)$ is a quantifier, either existential $\exists$ or universal $\forall$, $z_1, \ldots, z_n$ are distinct variables, and Φ is a propositional formula in $z_1, \ldots, z_n$. For example,

$$\exists x_1 \forall y \exists x_2 \{\{\overline{x}_1 \vee \overline{y} \vee x_2\} \wedge \{\overline{y} \vee \overline{x}_2\} \wedge \{x_2 \vee \{\{x_1 \vee \overline{y}\} \wedge \{y \vee x_2\}\}\}\} \tag{2}$$

is a QBF. In (1), $Q_1 z_1 \ldots Q_n z_n$ is the *prefix* and Φ is the *matrix*. We say that a literal l is *existential* if $\exists |l|$ belongs to the prefix of (1), and that it is *universal* otherwise. We say that (1) is in *Conjunctive Normal Form* (CNF) when Φ is a conjunction of *clauses*, where each clause is a disjunction of literals; we say that (1) is in *Disjunctive Normal Form* (DNF) when Φ is a disjunction of *terms*, where each term is a conjunction of literals. We use *constraints* when we refer to clauses and terms indistinctly. Finally, in (1), we define

– the *level of a variable* z_i, to be 1 + the number of expressions $Q_j z_j Q_{j+1} z_{j+1}$ in the prefix with $j \geq i$ and $Q_j \neq Q_{j+1}$;
– the *level of a literal* l, to be the level of $|l|$.

For example, in (2) x_2 is existential and has level 1, y is universal and has level 2, x_1 is existential and has level 3.

The semantics of a QBF φ can be defined recursively as follows. If the prefix is empty, then φ's satisfiability is defined according to the truth tables of propositional logic; if φ is $\exists x \psi$, φ is satisfiable if and only if φ_x or $\varphi_{\overline{x}}$ are satisfiable; if φ is $\forall y \psi$, φ is satisfiable if and only if φ_y and $\varphi_{\overline{y}}$ are satisfiable. If φ is (1) and l is a literal with $|l| = z_i$, φ_l is the QBF whose matrix is obtained from Φ by substituting l with TRUE and $\bar{l}$ with FALSE, and whose prefix is $Q_1 z_1 Q_2 z_2 \ldots Q_{i-1} z_{i-1} Q_{i+1} z_{i+1} \ldots Q_n z_n$. Two QBFs are *equivalent* if they are either both satisfiable or both unsatisfiable.

2.1 Basic QBF solver

Following the terminology of [2], we call *Extended QBF* (EQBF) an expression of the form:

$$Q_1 z_1 \ldots Q_n z_n \langle \Psi, \Phi, \Theta \rangle \qquad (n \geq 0) \tag{3}$$

```
bool SOLVE($Q, \Phi, \Psi, \Theta, S$)
1  $\langle Q, \Phi, \Psi, \Theta, S\rangle \leftarrow$ SIMPLIFY($Q, \Phi, \Psi, \Theta, S$)
2  if $\Phi$ is empty or $\emptyset_\forall \in \Theta$ then
3      return TRUE
4  if $\emptyset_\exists \in \Phi \cup \Psi$ then
5      return FALSE
6  $l \leftarrow$ CHOOSE-LITERAL($Q, \Phi, \Psi, \Theta$)
7  if $l$ is existential then
8      return SOLVE($Q, \Phi, \Psi \cup \{l\}, \Theta, S$) or
              SOLVE($Q, \Phi, \Psi \cup \{\bar{l}\}, \Theta, S$)
9  else
10     return SOLVE($Q, \Phi, \Psi, \Theta \cup \{\bar{l}\}, S$) and
              SOLVE($Q, \Phi, \Psi, \Theta \cup \{l\}, S$)
```

```
set SIMPLIFY($Q, \Phi, \Psi, \Theta, S$)
11 while $\{l\}_\exists \in \Phi \cup \Psi$ or $\{\bar{l}\}_\forall \in \Theta$ do
12     $S \leftarrow S \cup \{l\}$
13     $Q \leftarrow$ REMOVE($Q, |l|$)
14     for each $c \in \Phi \cup \Psi$ s.t. $l \in c$ do
15         $\Phi \leftarrow \Phi \setminus \{c\}$
16         $\Psi \leftarrow \Psi \setminus \{c\}$
17     for each $t \in \Theta$ s.t. $\bar{l} \in t$ do
18         $\Theta \leftarrow \Theta \setminus \{t\}$
19     for each $c \in \Phi$ s.t. $\bar{l} \in c$ do
20         $\Phi \leftarrow (\Phi \setminus \{c\}) \cup \{c \setminus \{\bar{l}\}\}$
21     for each $c \in \Psi$ s.t. $\bar{l} \in c$ do
22         $\Psi \leftarrow (\Psi \setminus \{c\}) \cup \{c \setminus \{\bar{l}\}\}$
23     for each $t \in \Theta$ s.t. $l \in t$ do
24         $\Theta \leftarrow (\Theta \setminus \{t\}) \cup \{t \setminus \{l\}\}$
25 return $\langle Q, \Phi, \Psi, \Theta, S\rangle$
```

Fig. 2. Basic algorithm of a QBF solver.

where $Q_1 z_1 \ldots Q_n z_n \Phi$ is a QBF, Ψ is a conjunction of clauses, Θ is a disjunction of terms, and such that, given any sequence of literals $l_1; \ldots ; l_m$ with $m \leq n$ and $|l_i| = z_i$, the following three formulas are equi-satisfiable:

$$\{\ldots\{\{Q_1 z_1 Q_2 z_2 \ldots Q_n z_n (\wedge_{c\in\Phi} \vee_{l\in c} l)\}_{l_1}\}_{l_2} \ldots\}_{l_m},$$
$$\{\ldots\{\{Q_1 z_1 Q_2 z_2 \ldots Q_n z_n ((\wedge_{c\in\Phi} \vee_{l\in c} l) \wedge (\wedge_{c\in\Psi} \vee_{l\in c} l))\}_{l_1}\}_{l_2} \ldots\}_{l_m},$$
$$\{\ldots\{\{Q_1 z_1 Q_2 z_2 \ldots Q_n z_n ((\wedge_{c\in\Phi} \vee_{l\in c} l) \vee (\vee_{t\in\Theta} \wedge_{l\in t} l))\}_{l_1}\}_{l_2} \ldots\}_{l_m}.$$

In Figure 2 we present the basic algorithm of a QBF solver. The main function SOLVE takes five parameters: Q is a list and Φ is a set of clauses corresponding, respectively, to the prefix and the matrix of the input QBF, Ψ (resp. Θ) is a set of clauses (resp. terms), and S is a set of literals called *assignment*; initially $S = \emptyset$, $\Psi = \emptyset$, and $\Theta = \emptyset$. SOLVE returns TRUE if the input QBF is satisfiable and FALSE otherwise. SOLVE requires the input QBF to be in CNF, and, as customary in search algorithms, it deals with CNF instances as if they were *sets* of clauses and with clauses as if they were *sets* of literals. SOLVE works in four steps (line numbers refer to Figure 2):

1. Simplify the input instance with SIMPLIFY (line 1).
2. Check if the *termination condition* is met, i.e., if we are done with the current search path and backtracking is needed (lines 2-5): if the test in line 2 is true, then S is a *solution*, while if the test in line 4 is true, then S is a *conflict*; $\emptyset_\exists$ (resp. $\emptyset_\forall$) stands for an *empty clause* (resp. *empty term*), i.e., a constraint comprised of universal (resp. existential) literals only.
3. Choose heuristically a literal l (line 6) such that (i) $|l|$ is in Q, and (ii) there is no other literal l' in Q whose level is greater than the level of l; the literal returned by CHOOSE-LITERAL is called *branching literal*.
4. Branch on the chosen literal: if the literal is existential, then an *OR node* is explored (line 8), otherwise an *AND node* is explored (line 10).

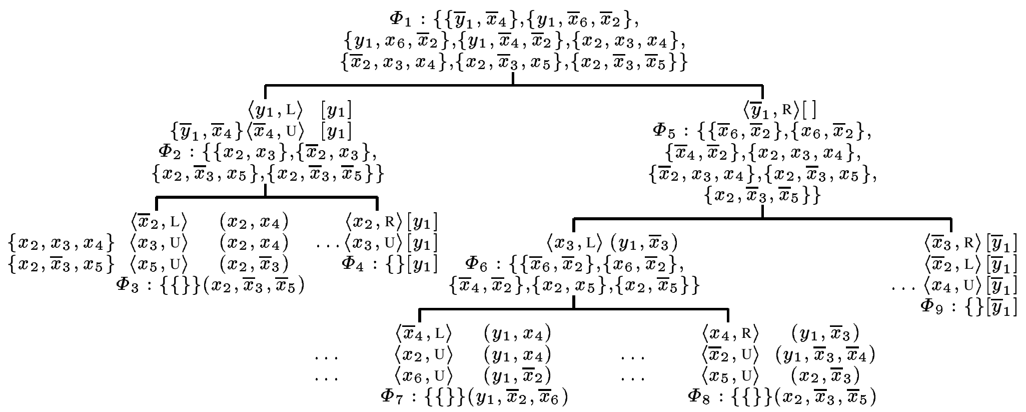
$\Phi_1 : \{\{\overline{y}_1, \overline{x}_4\},\{y_1, \overline{x}_6, \overline{x}_2\}, \{y_1, x_6, \overline{x}_2\},\{y_1, \overline{x}_4, \overline{x}_2\},\{x_2, x_3, x_4\}, \{\overline{x}_2, x_3, x_4\},\{x_2, \overline{x}_3, x_5\},\{x_2, \overline{x}_3, \overline{x}_5\}\}$
$\langle y_1, \text{L}\rangle$ $[y_1]$
$\{\overline{y}_1, \overline{x}_4\}\langle \overline{x}_4, \text{U}\rangle$ $[y_1]$
$\Phi_2 : \{\{x_2, x_3\},\{\overline{x}_2, x_3\}, \{x_2, \overline{x}_3, x_5\},\{x_2, \overline{x}_3, \overline{x}_5\}\}$
$\langle \overline{y}_1, \text{R}\rangle[\,]$
$\Phi_5 : \{\{\overline{x}_6, \overline{x}_2\},\{x_6, \overline{x}_2\}, \{\overline{x}_4, \overline{x}_2\},\{x_2, x_3, x_4\}, \{\overline{x}_2, x_3, x_4\},\{x_2, \overline{x}_3, x_5\}, \{x_2, \overline{x}_3, \overline{x}_5\}\}$
$\langle \overline{x}_2, \text{L}\rangle$ (x_2, x_4)
$\{x_2, x_3, x_4\}$ $\langle x_3, \text{U}\rangle$ (x_2, x_4)
$\{x_2, \overline{x}_3, x_5\}$ $\langle x_5, \text{U}\rangle$ $(x_2, \overline{x}_3)$
$\Phi_3 : \{\{\}\}(x_2, \overline{x}_3, \overline{x}_5)$
$\langle x_2, \text{R}\rangle[y_1]$
$\ldots\langle x_3, \text{U}\rangle[y_1]$
$\Phi_4 : \{\}[y_1]$
$\langle x_3, \text{L}\rangle$ $(y_1, \overline{x}_3)$
$\Phi_6 : \{\{\overline{x}_6, \overline{x}_2\},\{x_6, \overline{x}_2\}, \{\overline{x}_4, \overline{x}_2\},\{x_2, x_5\},\{x_2, \overline{x}_5\}\}$
$\langle \overline{x}_3, \text{R}\rangle[\overline{y}_1]$
$\langle \overline{x}_2, \text{L}\rangle[\overline{y}_1]$
$\ldots\langle x_4, \text{U}\rangle[\overline{y}_1]$
$\Phi_9 : \{\}[\overline{y}_1]$
$\langle \overline{x}_4, \text{L}\rangle$ (y_1, x_4)
$\ldots$ $\langle x_2, \text{U}\rangle$ (y_1, x_4)
$\ldots$ $\langle x_6, \text{U}\rangle$ $(y_1, \overline{x}_2)$
$\Phi_7 : \{\{\}\}(y_1, \overline{x}_2, \overline{x}_6)$
$\langle x_4, \text{R}\rangle$ $(y_1, \overline{x}_3)$
$\ldots$ $\langle \overline{x}_2, \text{U}\rangle$ $(y_1, \overline{x}_3, \overline{x}_4)$
$\ldots$ $\langle x_5, \text{U}\rangle$ $(x_2, \overline{x}_3)$
$\Phi_8 : \{\{\}\}(x_2, \overline{x}_3, \overline{x}_5)$

Fig. 3. Search tree of a QBF solver with learning. The prefix is $\forall y_1 \exists x_2 \exists x_3 \exists x_4 \exists x_5 \exists x_6$.

Consider the quadruple $\langle Q, \Phi, \Psi, \Theta \rangle$. In the following we say that a literal l is:

– *open* if $|l|$ is in Q, and *assigned* otherwise;
– *unit* if there exists a clause $c \in \Phi \cup \Psi$ (resp. a term $t \in \Theta$) such that l is the only existential in c (resp. universal in t) and there is no universal (resp. existential) literal $l' \in c$ (resp. $l' \in t$) such that the level of $|l'|$ is greater than the level of $|l|$.

Now consider the simplification routine SIMPLIFY in Figure 2: $\{l\}_\exists$ (resp. $\{l\}_\forall$) denotes a constraint which is unit in l, and REMOVE(Q,z_i) returns the prefix obtained from Q by removing $Q_i z_i$. The function SIMPLIFY has the task of finding and assigning all unit literals at every node of the search tree. SIMPLIFY loops until either $\Phi \cup \Psi$ or Θ contain a unit literal (line 11). Each unit literal l is added to the current assignment (line 12), removed from Q (line 13), and then it is assigned by:

– removing all the clauses (resp. terms) to which l (resp. $\overline{l}$) pertains (lines 14-18), and
– removing $\overline{l}$ (resp. l) from all the clauses (resp. terms) to which $\overline{l}$ (resp. l) pertains (lines 19-24).

We say that an assigned literal l (i) *eliminates* a clause (resp. a term) when l (resp. $\overline{l}$) is in the constraint, and (ii) *simplifies* a clause (resp. a term) when $\overline{l}$ (resp. l) is in the constraint.

2.2 Learning in QBF

The algorithm presented in Figure 2 uses standard chronological backtracking (CB) to visit an implicit AND-OR search tree induced by the input QBF. The main problem with CB is that it may lead to the fruitless exploration of possibly large subtrees where all the leaves are either conflicts (in the case of subtrees rooted at OR nodes) or solutions (in the case of subtrees rooted at AND nodes). This is indeed the case when the conflicts/solutions are caused by some choices done way up in the search tree. Learning, as introduced for QBFs by [2], improves on CB by recording the information which is unveiled during the search, so that it can be reused later to avoid useless exploration of the search space. Adding a learning mechanism to the basic algorithm of Figure 2 amounts to:

- storing in Ψ the clauses corresponding to the *nogoods* determined while backtracking from a contradiction, and
- storing in Θ the terms corresponding to the *goods* determined while backtracking from a solution

in such a way that the EQBF invariants (1) and (2) introduced in Section 2.1 are fulfilled.

In the following, we call *left branch* the first branch explored in a node, and *right branch* the second branch which is possibly explored. Learning is implemented by associating a *reason* to each assignment resulting from a deduction, be it a unit, or a right branch. Assignments resulting from a choice, i.e., left branches, do not have a corresponding reason. The reason of a unit literal is simply a constraint where the literal is unit, and the reason of a right branch is calculated by the learning process as sketched in the following. Each time a contradiction is found, an empty clause c is in $\Phi \cup \Psi$, and it can be used as (initial) *working reason.* While backtracking from a conflict on an existential literal l:

- if $\overline{l}$ is not in the working reason, then l is not responsible for the current conflict and it can be skipped;
- if $\overline{l}$ is in the working reason and l is a unit or a right branch, then a new working reason is obtained by resolving the reason of l with the old working reason;
- if $\overline{l}$ is in the working reason and l is a left branch, then the current working reason is the reason of the right branch on $\overline{l}$.

Learning works almost symmetrically for solutions and universal literals, except that the initial working reasons are computed differently. Notice that the procedure outlined above is just a sketch (see [2, 6, 7] for more details). For our purposes, it is sufficient to say that the working reasons are exactly the clauses (resp. the terms) that correspond to nogoods (resp. goods) and that can be stored in Ψ (resp. Θ).

Now consider the following QBF:

$$\begin{aligned}\forall y_1 \exists x_2 \exists x_3 \exists x_4 \exists x_5 \exists x_6 \{ & \{\overline{y}_1, \overline{x}_4\}, \{y_1, \overline{x}_6, \overline{x}_2\}, \{y_1, x_6, \overline{x}_2\}, \{y_1, \overline{x}_4, \overline{x}_2\}, \\ & \{x_2, x_3, x_4\}, \{\overline{x}_2, x_3, x_4\}, \{x_2, \overline{x}_3, x_5\}, \{x_2, \overline{x}_3, \overline{x}_5\} \}\end{aligned} \tag{4}$$

In Figure 3 we present a search tree induced by (4) and possibly explored by the QBF solver shown in Figure 2 augmented with learning as described above. In the Figure:

- Φ_1 is the matrix of the input instance (4), and for each $i > 1$, Φ_i is the matrix of (4) after the assignments performed along the path to that node; the numbering of the Φ_i's reflects the depth-first nature of the exploration and we abuse notation by speaking of "node Φ_i" to refer to the node where Φ_i is the result of the applied simplifications.

- Each bifurcation in the search tree corresponds to a node, and the branching literal is indicated immediately below the bifurcation; we tag with "L" the left branch, and with "R" the right branch (the reasons of right branches are not indicated).
- Each unit literal is tagged with "U", and the corresponding reason is indicated at its left, e.g., $\{\overline{y}_1, \overline{x}_4\}$ is the reason of $\overline{x}_4$ at node Φ_2.
- the working reasons computed for conflicts (resp. solutions) during backtracking are indicated at the right of each assignment enclosed by "(" and ")" (resp. "[" and "]"), e.g., $(x_2, \overline{x}_3, \overline{x}_5)$ is the initial working reason of the conflict at node Φ_3, and $[y_1]$ is the initial working reason of the solution at node Φ_4.
- Finally, "..." indicate information that we omitted for the sake of compactness.

The search represented in Figure 3 proceeds as follows. Branching left on y_1 causes $\overline{x}_4$ to be propagated as unit and Φ_1 is simplified into Φ_2. Then, branching on $\overline{x}_2$ causes x_3 and x_5 to be propagated as unit and a contradiction to be found (Φ_3 contains an empty clause). The initial working reason $r_1 = (x_2, \overline{x}_3, \overline{x}_5)$ is the clause that became empty in Φ_3. Resolving r_1 with the reason of x_5 we obtain $r_2 = (x_2, \overline{x}_3)$. Backtracking stops at the left branch $\overline{x}_2$, since x_2 is in the working reason at that point, and we must branch right on x_2 with reason (x_2, x_4). This time a solution is found (Φ_4 is the empty set), and the corresponding working reason $[y_1]$ is calculated. Backtracking now goes up to the left branch on y_1 and the search continues by branching right on $\overline{y}_1$ with reason $[y_1]$, and so on. Notice that right branches can be propagated as unit literals, as long as their reasons are learned. Notice also that the unit on x_2 at node Φ_7, rests on the assumption that the reason of x_2 computed while backtracking from node Φ_3 has been learned.

3 Monotone Literals and Learning: Issues

As we have seen in section 1, MLF turns out to be a very effective pruning technique for QBF solvers. The introduction of MLF in QBF is due to Cadoli et al. [1]. Following their terminology, given a QBF φ a literal l is *monotone* (or *pure*) if $\overline{l}$ does not appear in the matrix of φ. In the presence of learning, the input instance is augmented by the constraints stored while backtracking, so our first question is:

Question 1. What is the definition of pure literal in the presence of learned constraints?

The straightforward answer is the following

Definition 1. Given an EQBF $\varphi = Q_1 z_1 \ldots Q_n z_n \langle \Psi, \Phi, \Theta \rangle$, a literal l is monotone in φ iff there is no constraint k such that $\overline{l} \in k$ and $k \in \Psi \cup \Phi \cup \Theta$.

In theory, the argument is closed: detecting monotone literals in the presence of learned constraints amounts to considering the matrix Φ, as well as the learned constraints in Ψ and Θ. However, in practice, we are left with the problem of detecting monotone literals as soon as they arise. It turns out that monotone literal detection algorithms must keep track of the eliminated constraints, while this is unnecessary when unit literal propagation is the only lookahead technique. Indeed, the most efficient algorithms for unit propagation like, e.g., two/three literal watching [8], take into account simplifications only. Thus, introducing MLF is bound to increase the time needed to assign a literal l, and the increase is proportional to the number of constraints that contain l. Moreover, even using a lazy data structure to detect monotone literals like, e.g., clause watching [8], for each variable z in Φ we have to maintain an index to *all* the constraints that contain either z or its negation. Such an index must be (i) updated each time a new constraint is learned, (ii) scanned each time we must search for new constraints to watch, and (optionally) (iii) scanned each time we want to forget a learned constraint. Therefore, under the assumptions of Def. 1, even the best known algorithms for monotone literals detection suffer an overhead proportional to the cardinality of the set $\Phi \cup \Psi \cup \Theta$, which, in practice, tends to be substantially bigger than the cardinality of Φ (in theory it may become exponentially bigger).

Given the above considerations, a pragmatical approach would be to keep the original definition of monotone literals unchanged, i.e., use the following

Definition 2. Given an EQBF $\varphi = Q_1 z_1 \ldots Q_n z_n \langle \Psi, \Phi, \Theta \rangle$, a literal l is monotone in φ iff there is no constraint k such that $\overline{l} \in k$ and $k \in \Phi$.

According to this definition, the impact of Ψ and Θ growth is not an issue anymore. The correctness of Def. 2 follows from the fact that Ψ and Θ can be safely discarded without altering the satisfiability of the input QBF or the correctness of the solver. This alternative definition brings us to the second question:

Question 2. Are Def. 1 and Def. 2 equivalent, i.e., do they result in exactly the same sets of monotone literals?

If the answer was positive, then we could rely on Def. 2 for an efficient implementation of monotone literals detection, and we would have an easy integration with learning: if an existential (resp. universal) literal is monotone according to Def. 1, it is always possible to compute the working reason wr of a conflict (resp. solution) in order to avoid $\overline{l} \in wr$ as reported, e.g., in [5].[1] Unfortunately, the answer is negative, as a brief tour of the example in Figure 3 can clarify. Looking at Figure 3, we can see that at node Φ_7, the literal $\overline{x}_4$ is assigned as left branch. According to Def. 2, $\overline{x}_4$ is a monotone literal since there is no constraint in Φ_6 containing x_4. Assuming that all the working reasons are to be learned, the clause $\langle x_2, x_4 \rangle$ has been added to Ψ while backtracking on $\overline{x}_2$ in the leftmost path of the search tree. Therefore, according to Def. 1, $\overline{x}_4$ is not monotone since there is a constraint in $\Phi_6 \cup \Psi_6$ which contains x_4.

Even if Def. 2 captures a different set of monotone literals, still it might be the case that using it does not produce undesired effects, i.e., that the following question admits a positive answer:

Question 3. Is it possible that a working reason obtained from a conflict (resp. a solution) contains a monotone existential (resp. universal) literal detected according to Def. 2?

[1] Notice that this property would be preserved even if we considered monotone an existential (resp. universal) literal l when there is no constraint k in $\Phi \cup \Psi$ (resp. $\Phi \cup \Theta$) such that $\overline{l} \in k$; although less restrictive than Def. 1, this alternative characterization of monotone literals has exactly the same drawbacks.

Notice that the converse situation, i.e., monotone existential (resp. universal) literals appearing in the working reasons obtained from solutions (resp. conflicts) is not an issue, but the normal and expected behavior of the algorithm. Looking at Figure 3 we can see that the answer to our last question is positive. Indeed, if we are to consider x_4 as monotone at node Φ_6, we see that the working reason obtained from the contradiction found at node Φ_7 contains x_4. Notice that the "simple" solutions of discarding x_4 from the working reason (y_1, x_4), is bound to fail: the resulting reason would be (y_1), and it would cause the solver to incorrectly report that (4) is unsatisfiable. As we explain in the next section, computing a suitable reason for $\overline{x}_4$ would involve extra time and more subtle issues that we discuss in some detail. Summing up, we have established that Def. 1 and Def. 2 result in different sets of monotone literals, so they cannot be used indistinctly. In particular, while Def. 2 makes the detection of monotone literals more efficient in the presence of learned constraints, it also introduces complications in the learning algorithm which cannot be easily dismissed.

4 Monotone Literals and Learning: Solutions

As discussed in the previous section, if we assume an efficient implementation of monotone literals detection according to Def. 2, we are left with the problem that an existential (resp. universal) monotone literal may appear in the working reason obtained from a conflict (resp. a solution), and there is no easy way to get rid of it. In the following we call these monotone literals *spurious*. If we want to keep Def. 2, then we have two possible solutions: (i) compute reasons for spurious monotone literals, or (ii) forbid simplifications caused by monotone literals in learned constraints. The first solution works *a posteriori*: we let spurious monotone literals arise in the working reason, and then we take care of them. The second one works *a priori*: we prevent spurious literals from appearing in the working reasons. Each solution has its own advantages and disadvantages, which are discussed in the next two subsections. We anticipate that the second solution is the one of our choice, because of its simplicity of implementation combined with its effectiveness.

4.1 Computing reasons for monotone literals

As we have seen in Section 2.2 every deduction in a search algorithm with learning must be supported by a reason, either readily available (as in the case of unit literals), or computed by the learning algorithm (as in the case of right branches). Monotone literals share some similarities with unit literals: they can be propagated as soon as they are detected regardless of their level, and they are the result of a deduction, as opposed to branching literals which result from a choice and must be assigned following the prefix order. However, while the reason of a unit literal can be localized in the constraints that force the literal to be unit, monotone literals are deduced by considering the input instance as a whole. More precisely, a literal l is monotone when all the constraints where $\overline{l}$ occurs are eliminated from at least another literal assigned before l. For instance, looking at Φ_1 in Figure 3, we see that x_4 occurs in $\{x_2, x_3, x_4\}$ and $\{\overline{x}_2, x_3, x_4\}$: whenever x_3 is assigned, both clauses are eliminated and $\overline{x}_4$ becomes monotone, i.e., given Φ_1, once we assign $x_3, \overline{x}_4$ can be assigned as monotone. In this example, a natural choice for the reason of $\overline{x}_4$ to be assigned as monotone at node Φ_6 would be exactly $r = \{\overline{x}_3, \overline{x}_4\}$. However, r cannot be obtained by applying resolution to some of the constraints in Φ_1. Intuitively, adding r to the set of constraints under consideration reduces the set of assignments S satisfying the input formula, but it does not alter the satisfiability of the input QBF. More precisely, let φ be a QBF in CNF, $S = \{l_1, l_2, \ldots, l_n\}$ $(n \geq 0)$ be a set of literals, and φ_S be the QBF obtained from φ by assigning the literals in S and then performing unit propagation. The following property holds:

Proposition 3. *Given a QBF φ in CNF, and an existential literal l in φ, let S be a set of literals such that the level of each literal in S is greater than or equal to the level of l. If l is monotone in φ_S, then φ is equivalent to the QBF obtained by adding $(\vee_{l' \in S}\overline{l'} \vee l)$ to the matrix Φ of φ.*

In the hypotheses of the proposition, we can add a new constraint (r in our example) which does not change the satisfiability of the input QBF (and thus of the EQBFs computed during the search), and which enables l (x_4 in our example) to be assigned as a unit literal. Proposition 3 is general enough: during the search, if S is the current assignment, and l is a monotone existential literal, then we can always reconstruct a set S' satisfying the hypotheses of Proposition 3 by

1. considering the set $S' = \{l' \mid l' \in c \cap S, \overline{l} \in c, c \in \Phi\}$. S' contains at least one literal (eliminating the clause) for each clause $c \in \Phi$ with $\overline{l} \in c$.
2. If the level of each literal in S' is greater than the level of l, we are done.
3. Otherwise, let l' one literal in S' with level smaller than the level of l. Then, l' has been assigned as a unit with a reason r. Assign $(S' \setminus \{l'\}) \cup (\{l'' \mid \overline{l''} \in r\} \setminus \{\overline{l'}\})$ to S' and go to step 2.

The procedure is guaranteed to terminate: Indeed, all the branching literals in S have a level greater or equal to the level of l.

In Proposition 3, the assumption that φ is in CNF allows us to define φ_S easily, but it is not essential: if φ is not in CNF, then the clause can be added conjunctively to its matrix. If φ is in CNF, then the clause can be added to the matrix in a seemingly straightforward way. However, this solution has some drawbacks. First, exponentially many monotone existential literals can be detected during the search, and each one may cause the addition of a new clause to Φ with a resulting blow up in space: in such a situation, even restricting monotone literal detection to Φ might cause a substantial increase in the search time. Second, since the reason of a monotone literal l is not readily available, for each spurious monotone literal we should also pay the overhead associated to the computation of its reason.

Things become practically unfeasible when considering universal monotone literals. For these literals, we can state the following proposition, in which we assume that φ_S is defined analogously to the above.

Proposition 4. *Given a QBF φ in DNF, and a universal literal l in φ, let S be a set of literals such that the level of each literal in S is greater than or equal to the level of l. If l is monotone in φ_S, then φ is equivalent to the QBF obtained by adding $(\wedge_{l' \in S} l' \wedge l)$ to its set of terms.*

As before, the assumption that φ is in DNF is not essential. The proposition can be stated for an arbitrary QBF, and the term has to be added disjunctively to its matrix. So far, it is obvious that the treatment of universal monotone literals outlined in this section cannot be easily integrated in DLL-based QBF solvers. As we said in section 2, these solvers work with a CNF representation of the input instance, and this property is lost as soon as we start adding reasons for monotone universal literals. Absorbing the impact of this structural change would involve a complete redesign of the algorithms to deal with non-CNF QBF instances. If the input QBF is in DNF, a symmetrical argument applies to the computation of reasons for monotone existential literals.

4.2 Avoiding simplifications on monotone literals

In the following, we describe how to use Def. 2 for the detection of monotone literals and, at the same time, to avoid the intricacies outlined in the previous subsection. The basic idea is to prevent spurious monotone literals from appearing in the working reason by skipping the simplifications which involve a monotone literal and a learned constraint. Indeed, the reason why Def. 2 causes spurious monotone literals is that it conveniently disregards the learned constraints in the *detection* phase. Pushing this philosophy further, we can solve the problem by (partially) disregarding learned constraints also during the *propagation* phase. The major (and only) drawback of this approach is that we impact negatively on the effectiveness of learning by preventing some pruning to occur. However, this seems a small price to pay when compared to the implications of computing reasons for spurious monotone literals.

To see that the above solution is indeed effective, assume that an existential (resp. universal) literal l is monotone according to Def. 2, i.e., there is no constraint k in Φ such that $\bar{l} \in k$, but l is not monotone according to Def. 1, i.e., there is at least a clause c in Ψ (resp. a term t in Θ) such that $\bar{l} \in c$ (resp. $\bar{l} \in t$). Let l be an existential literal. If we simplify c when assigning l then, if all the remaining literals of c are simplified, a contradiction is found, and the corresponding working reason will contain l as a spurious monotone literal. If we disregard c when assigning l then c will never be a conflict even if all its remaining literals are simplified, so the working reason wr of a possible conflict can be calculated with $l \notin wr$. Analogously, let l be a universal literal. If we simplify t when assigning $\bar{l}$ then, if all the remaining literals of t are simplified, a solution is found, and the corresponding working reason will contain $\bar{l}$ as a spurious monotone literal. As before, disregarding t when assigning $\bar{l}$ solves the issue. The approach is sound since it is equivalent to temporarily forgetting the learned constraints, and we know from Section 2.2 that this can be done without affecting the correctness of the procedure.

5 Experimental Analysis

As shown in Section 1, both MLF and learning can separately improve the performances of a QBF solver. The question that we address in the following is whether our solution of choice for the integration of MLF with learning is effective, and, in particular, if a

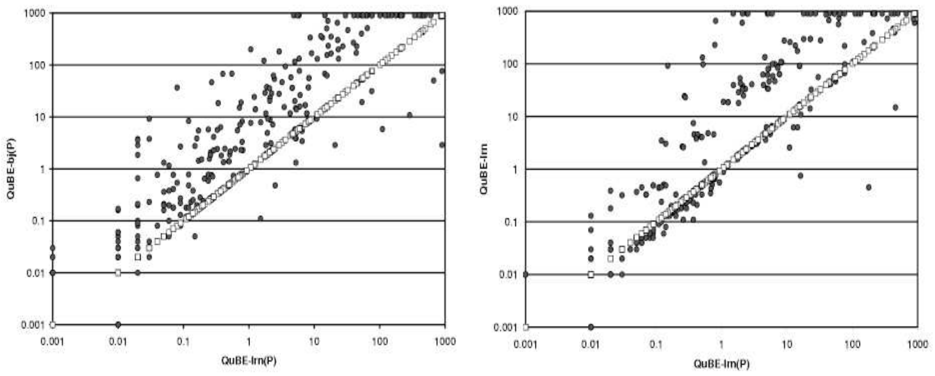

Fig. 4. Effectiveness of MLF in a QBF solver with learning.

QBF solver with MLF and learning can be more effective than the same solver without one of the two optimizations. To answer this question, we consider the systems QUBE-BJ(P) and QUBE-LRN described in Section 1, and the system QUBE-LRN(P), a version of QUBE that integrates MLF and learning using the techniques described in Section 4.2. Since QUBE is designed with the purpose of being effective on real-world instances, our experiments use the same set of 450 verification and planning benchmarks that constituted part of the 2003 QBF evaluation [3]: 25% of these instances are from verification problems [9, 10], and the remaining are from planning domains [11, 12]. All the experiments where run on a farm of PCs, each one equipped with a PIV 2.4GHz processor, 1GB of RAM, and running `Linux RedHat 7.2`. For practical reasons, the time limit for each benchmark is set to 900s.

The results of our experiments are reported in the plots of Figure 4. In the plots, each solid-fill dot represents an instance, QUBE-LRN(P) solving time is on the x-axis (seconds, log scale), QUBE-BJ(P) (Figure 4 left), QUBE-LRN (Figure 4 right) solving times are on the y-axes (seconds, log scale). The diagonal (outlined boxes) represents the solving time of QUBE-LRN(P) against itself: similarly to the plots of Figure 1 described in Section 1, the relative position of the solid-fill and the outlined dots gives us an instance-wise comparison between QUBE-LRN(P) and its siblings. The plots of Figure 4 show that, all other things being equal, QUBE-LRN(P) substantially outperforms both QUBE-BJ(P) and QUBE-LRN. Focusing on QUBE-LRN(P) vs. QUBE-BJ(P), we see that both solvers exceed the time limit on 74 instances. On the remaining 376, QUBE-LRN(P) is as fast as, or faster than QUBE-BJ(P) on 345 instances, and it is at least one order of magnitude faster than QUBE-BJ(P) on 79 problems (the converse is true on 10 problems only). These numbers witness that adding MLF does not impact on the performance gap between backjumping and learning. Focusing on QUBE-LRN(P) vs. QUBE-LRN, we see that both solvers exceed the time limit on 75 instances. On the remaining 375, QUBE-LRN(P) is as fast as, or faster than QUBE-LRN on 260 instances, and it is at least one order of magnitude faster than QUBE-LRN on 66 problems (the converse is true on 17 problems only). Therefore, using the techniques described

test	ZCHAFF	QUBE-LRN	QUBE-LRN(P)
bart10	34881	31295	144
bart11	1399736	2145767	162
bart12	3993454	–	180
bart13	176	176	176
bart14	–	6241121	195
bart15	–	–	215
bart16	–	–	210
bart17	–	–	231
bart18	–	–	252
bart19	–	–	248
bart20	–	–	270
bart21	–	–	293
bart22	–	–	288
bart23	–	–	312
bart24	–	–	336
bart25	–	–	378
bart27	–	–	432
bart26	–	–	405
bart28	–	–	514
bart29	–	–	466

Table 1. QUBE-LRN(P), QUBE-LRN, and ZCHAFF performances on FPGA routing instances.

in Section 4.2 we have managed to exploit the pruning power of MLF, without incurring into the possible inefficiencies due to learning. Overall, QUBE-LRN(P) is the best among the four versions of QUBE that we analyzed in our experiments.

In Table 1 we present some of the results obtained running QUBE on a collection of challenging real-world SAT instances described in [13]. The original motivation for this experiment was to assess the performances of QUBE with respect to state-of-the-art SAT solvers. While selecting the solvers to compare with QUBE, we observed that many of the recent ones (e.g., ZCHAFF [14], and BERKMIN [15]) implement some kind of learning mechanism, but most of them do not implement MLF. The reason of this, apart from the intricacies that we have explored in this paper, can also be traced back to the common belief that MLF is not an effective technique in SAT. In the words of Freeman [16]: "... [MLF] only helps for a few classes of SAT problems, and even when it does help, it does not result in a substantial reduction in search tree size.". Indeed, a comparative analysis between QUBE-LRN(P) and QUBE-LRN on our SAT test set, reveals that the overall performances of QUBE-LRN are better than those of QUBE-LRN(P). However, the results on the subset comprised of the "bart"[2] FPGA routing instances reported in Table 1, witness that MLF, combined with learning and an efficient implementation, can dramatically improve the performances also in SAT. Each line of the table reports the label of the instance (first column), the number of *tries*, i.e., the total number of assignments, performed by ZCHAFF (second column), and our solvers QUBE-LRN (third column) and QUBE-LRN(P) (fourth column). We used tries instead of CPU times in order to give a precise account about the size of the search space explored by the solvers. The dash "–" indicates that the number of tries could not be assessed because the solver exceeded the CPU time limit of 20 minutes. As we can see from Table 1, with the only exception of "bart13", the search space of QUBE-LRN(P) is substantially smaller than that of QUBE-LRN and ZCHAFF, and, on most instances, the performance gap is at least three orders of magnitude. We conjecture that the reason of such an improvement lies in the favorable interaction between MLF

[2] This is the original nickname given to the instances when they appeared for the first time in the context of the 2002 SAT solvers competition.

and the specific CNF conversion used by the authors of the benchmarks series. We are currently investigating the relationships of this result with some recent advancements in non-CNF SAT (see, e.g., [17])

6 Conclusions and Related Work

We have considered the problem of the efficient implementation of MLF in a QBF solver with learning. As we have seen, various solutions are possible, and we discussed the advantages and the disadvantages of each one. Implementing our solution of choice in our state-of-the-art solver QUBE produces positive effects on its performances.

About the related work, MLF has been first defined for QBFs in [18], and since then it has been considered also by many other authors (see, e.g., [5, 19, 2]. Some of these works present also learning schemes (see also [20]), but none of them discusses in any detail whether MLF has been integrated in the corresponding QBF solver, how, and the problems faced in practice.

References

1. M. Cadoli, M. Schaerf, A. Giovanardi, and M. Giovanardi. An algorithm to evaluate quantified boolean formulae and its experimental evaluation. *Journal of Automated Reasoning*, 28(2):101–142, 2002.
2. E. Giunchiglia, M. Narizzano, and A. Tacchella. Learning for Quantified Boolean Logic Satisfiability. In *18th National Conference on Artificial Intelligence (AAAI 2002)*. AAAI Press/MIT Press, 2002.
3. D. Le Berre, L. Simon, and A. Tacchella. Challenges in the QBF arena: the SAT'03 evaluation of QBF solvers. In *Sixth International Conference on Theory and Applications of Satisfiability Testing (SAT 2003)*, volume 2919 of *Lecture Notes in Computer Science*. Springer Verlag, 2003.
4. E. Giunchiglia, M. Narizzano, and A. Tacchella. QuBE: A system for deciding Quantified Boolean Formulas satisfiability. In *First International Joint Conference on Automated Reasoning (IJCAR 2001)*, volume 2083 of *Lecture Notes in Artificial Intelligence*. Springer Verlag, 2001.
5. E. Giunchiglia, M. Narizzano, and A. Tacchella. Backjumping for Quantified Boolean Logic Satisfiability. In *Seventeenth International Joint Conference on Artificial Intelligence (IJCAI 2001)*. Morgan Kaufmann, 2001.
6. E. Giunchiglia, M. Narizzano, and A. Tacchella. Backjumping for Quantified Boolean Logic satisfiability. *Artificial Intelligence*, 145:99–120, 2003.
7. E. Giunchiglia, M. Narizzano, and A. Tacchella. QBF reasoning on real-world instances. In *7th International Conference on Theory and Applications of Satisfiability Testing (SAT 2004)*, 2004. Accepted, final version pending.
8. I.P. Gent, E. Giunchiglia, M. Narizzano, A. Rowley, and A. Tacchella. Watched Data Structures for QBF Solvers. In *Sixth International Conference on Theory and Applications of Satisfiability Testing (SAT 2003)*, volume 2919 of *Lecture Notes in Computer Science*. Springer Verlag, 2003.
9. C. Scholl and B. Becker. Checking equivalence for partial implementations. In *38th Design Automation Conference (DAC'01)*, 2001.

10. Abdelwaheb Ayari and David Basin. Bounded model construction for monadic second-order logics. In *12th International Conference on Computer-Aided Verification (CAV'00)*, number 1855 in LNCS, pages 99–113. Springer-Verlag, 2000.
11. J. Rintanen. Constructing conditional plans by a theorem prover. *Journal of Artificial Intelligence Research*, 10:323–352, 1999.
12. C. Castellini, E. Giunchiglia, and A. Tacchella. Sat-based planning in complex domains: Concurrency, constraints and nondeterminism. *Artificial Intelligence*, 147(1):85–117, 2003.
13. E. Giunchiglia, M. Maratea, and A. Tacchella. (In)Effectiveness of Look-Ahead Techniques in a Modern SAT Solver. In *9th Conference on Principles and Practice of Constraint Programming (CP 2003)*, volume 2833 of *Lecture Notes in Computer Science*. Springer Verlag, 2003.
14. M. W. Moskewicz, C. F. Madigan, Y. Zhao, L. Zhang, and S. Malik. Chaff: Engineering an efficient SAT solver. In *Proceedings of the 38th Design Automation Conference (DAC'01)*, pages 530–535, 2001.
15. E. Goldberg and Y. Novikov. BerkMin: A fast and robust SAT-solver. In *Design, Automation, and Test in Europe (DATE '02)*, pages 142–149, March 2002.
16. J. W. Freeman. *Improvements to propositional satisfiability search algorithms*. PhD thesis, University of Pennsylvania, 1995.
17. A. Kuehlmann, M. K. Ganai, V. Paruthi. Circuit-based Boolean Reasoning. In *38th Design Automation Conference*, 2001.
18. 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.
19. R. Letz. Lemma and model caching in decision procedures for quantified boolean formulas. In *Proceedings of Tableaux 2002*, LNAI 2381, pages 160–175. Springer, 2002.
20. L. Zhang and S. Malik. Conflict driven learning in a quantified boolean satisfiability solver. In *Proceedings of International Conference on Computer Aided Design (ICCAD'02)*, 2002.