QBF reasoning on real-world instances

Enrico Giunchiglia, Massimo Narizzano, and Armando Tacchella [⋆]

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

Abstract. During the recent years, the development of tools for deciding Quantified Boolean Formulas (QBFs) satisfiability has been accompanied by a steady supply of real-world instances, i.e., QBFs originated by translations from application domains such as formal verification and planning. QBFs from these domains showed to be challenging for current state-of-the-art QBF solvers, and, in order to tackle them, several techniques and even specialized solvers have been proposed. Among these techniques, there are (i) efficient detection and propagation of unit and monotone literals, (ii) branching heuristics that leverages the information extracted during the learning phase, and (iii) look-back techniques based on learning.
In this paper we discuss their implementation in our state-of-the-art solver QUBE, pointing out the non trivial issues that arised in the process. We show that all the techniques positively contribute to QUBE performances *on average*. In particular, we show that monotone literal fixing is the most important technique in order to improve capacity, followed by learning and the heuristics. The situation is reversed if we consider productivity. These and other observations are detailed in the body of the paper. For our analysis, we consider the formal verification and planning benchmarks from the 2003 QBF evaluation.

1 Introduction

During the recent years, the development of tools for deciding Quantified Boolean Formulas (QBFs) has been accompanied by a steady supply of real-word instances, i.e., QBFs originated by translations from application domains such as formal verification [1, 2] and planning [3, 4]. QBFs from these domains showed to be challenging for current state-of-the-art QBF solvers, and, in order to tackle them, several techniques and even specialized QBF solvers [5] have been proposed. Among the techniques that have been proposed for improving performances, there are (i) efficient detection and propagation of unit and monotone literals [6, 7], (ii) branching heuristics that leverage the information extracted during the learning phase [8], and (iii) look-back techniques based on conflict and solution learning [9–11].

In this paper we discuss the implementation of the above mentioned techniques in our state-of-the-art QBF solver QUBE [8], pointing out the non trivial issues that arised in the process. We show that all the techniques positively contribute to QUBE performances. In particular, we show that monotone literal fixing is the most important

[⋆] The authors wish to thank MIUR, ASI and the Intel Corporation for their financial support, and the reviewers who helped to improve the original manuscript.

technique in order to improve capacity (i.e., the ability to solve problems [12]), followed by learning and the heuristics. The situation is reversed if we consider productivity (i.e., the ability to quickly solve problems [12]). All these considerations obviously hold *on average*. Indeed, for each technique, there are instances in which it does not produce any benefit or, even worse, in which it causes a degradation in the performances. These and other observations are detailed in the body of the paper. For our analysis, we consider the formal verification and planning benchmarks from the 2003 QBF evaluation [13].[1] Our analysis is, to the best of our knowledge, the first which analyzes the contributions of (i) monotone literal fixing, (ii) the branching heuristic, and (iii) conflict and solution learning, for solving real-world problems. Though, strictly speaking, our results holds only for QUBE, we expect that the same will carry over to the majority of the other systems, which, like QUBE are based on [14]. We expect this to be true especially for solvers implementing learning as look-back mechanism, i.e., for solvers targeted to solve real-world benchmarks.

The paper is structured as follows. We first give some formal preliminaries, and the background knowledge representing the starting point of our analysis. We devote three sections to the improvements on lookahead techniques, the branching strategy and learning, respectively. In each section, we first briefly describe its implementation in QUBE, and then we present and discuss experimental results showing the positive contribution of the technique. We end the paper with some remarks.

2 Preliminaries

In this section, we first give some formal preliminaries. Then, we present the basic algorithm of QUBE, and finally the setting that we have used for our experimental analysis.

2.1 Formal preliminaries

Consider a set P of propositional letters. An *atom* is an element of P. A *literal* is an atom or the negation thereof. Given a literal l, $|l|$ denotes the atom of l, and $\overline{l}$ denotes the *complement* of l, i.e., if $l = a$ then $\overline{l} = \neg a$, and if $l = \neg a$ then $\overline{l} = a$, while $|l| = a$ in both cases. A *propositional formula* is a combination of atoms using the k-ary ($k \geq 0$) connectives $\wedge$, $\vee$ and the unary connective $\neg$. In the following, we use $\top$ and $\bot$ as abbreviations for the empty conjunction and the empty disjunction respectively. A *QBF* is an expression of the form

$$\varphi = Q_1 x_1 Q_2 x_2 \ldots Q_n x_n \Phi \qquad (n \geq 0) \tag{1}$$

where every Q_i $(1 \leq i \leq n)$ is a quantifier, either existential $\exists$, or universal $\forall$; $x_1 \ldots x_n$ are distinct atoms in P, and Φ is a propositional formula. $Q_1 x_1 Q_2 x_2 \ldots Q_n x_n$ is the *prefix* and Φ is the *matrix* of (1). A literal l is *existential*, if $\exists |l|$ is in the prefix, and

[1] At the time of this writing, the 2004 QBF comparative evaluation has just finished. Unfortunately, a now corrected bug in QUBE look-ahead caused QUBE to incorrectly decide a few randomly generated benchmarks, and thus it did not enter in the second stage of the evaluation.

```
bool SOLVE($Q$, $\Sigma$, $\Delta$, $\Pi$, $S$)
 1  $\langle Q, \Sigma, \Delta, \Pi, S\rangle \leftarrow$ LOOKAHEAD($Q$, $\Sigma$, $\Delta$, $\Pi$, $S$)
 2  if ($\Delta = \emptyset$ or $\emptyset_\forall \in \Pi$) then return TRUE
 3  if ($\emptyset_\exists \in \Sigma \cup \Delta$) then return FALSE
 4  $l \leftarrow$ CHOOSE-LITERAL($Q$, $\Sigma$, $\Delta$, $\Pi$)
 5  $\langle Q, \Sigma, \Delta, \Pi\rangle \leftarrow$ ASSIGN($l$, $Q$, $\Sigma$, $\Delta$, $\Pi$)
 6  $V \leftarrow$ SOLVE($Q$, $\Sigma$, $\Delta$, $\Pi$, $S \cup \{l\}$)
 7  if (($l$ is existential and $V$ is TRUE) then
 8      return TRUE
 9  else if (($l$ is universal and $V$ is FALSE) then
10      return FALSE
11  else
12      $\langle Q, \Sigma, \Delta, \Pi\rangle \leftarrow$ ASSIGN($\bar{l}$, $Q$, $\Sigma$, $\Delta$, $\Pi$)
13      return SOLVE($Q$, $\Sigma$, $\Delta$, $\Pi$, $S \cup \{\bar{l}\}$)
```

```
set LOOKAHEAD($Q$, $\Sigma$, $\Delta$, $\Pi$, $S$)
14  do
15      $\langle Q', \Sigma', \Delta', \Pi', S'\rangle \leftarrow \langle Q, \Sigma, \Delta, \Pi, S\rangle$
16      for each $l$ s.t. $\{l\}_\exists \in \Sigma \cup \Delta$ or $\{\bar{l}\}_\forall \in \Pi$ do
17          $S \leftarrow S \cup \{l\}$
18          $\langle Q, \Sigma, \Delta, \Pi\rangle \leftarrow$ ASSIGN($l$, $Q$, $\Sigma$, $\Delta$, $\Pi$)
19      for each $l$ s.t. $\{k \in \Sigma \cup \Delta \cup \Pi \mid \bar{l} \in k\} = \emptyset$ do
20          if ($l$ is existential) then
21              $\langle Q, \Sigma, \Delta, \Pi\rangle \leftarrow$ ASSIGN($l$, $Q$, $\Sigma$, $\Delta$, $\Pi$)
22          else
23              $\langle Q, \Sigma, \Delta, \Pi\rangle \leftarrow$ ASSIGN($\bar{l}$, $Q$, $\Sigma$, $\Delta$, $\Pi$)
24  while $\langle Q, \Sigma, \Delta, \Pi, S\rangle \neq \langle Q', \Sigma', \Delta', \Pi', S'\rangle$
25  return $\langle Q, \Sigma, \Delta, \Pi, S\rangle$
```

```
set ASSIGN($l$, $Q$, $\Sigma$, $\Delta$, $\Pi$)
26  $Q \leftarrow$ REMOVE($Q$, $|l|$)
27  for each $c \in \Sigma$ s.t. $l \in c$ do
28      $\Sigma \leftarrow \Sigma \setminus \{c\}$
29  for each $c \in \Delta$ s.t. $l \in c$ do
30      $\Delta \leftarrow \Delta \setminus \{c\}$
31  for each $t \in \Pi$ s.t. $\bar{l} \in t$ do
32      $\Pi \leftarrow \Pi \setminus \{t\}$
33  for each $c \in \Sigma$ s.t. $\bar{l} \in c$ do
34      $\Sigma \leftarrow (\Sigma \setminus \{c\}) \cup \{c \setminus \{\bar{l}\}\}$
35  for each $c \in \Delta$ s.t. $\bar{l} \in c$ do
36      $\Delta \leftarrow (\Delta \setminus \{c\}) \cup \{c \setminus \{\bar{l}\}\}$
37  for each $t \in \Pi$ s.t. $l \in t$ do
38      $\Pi \leftarrow (\Pi \setminus \{t\}) \cup \{t \setminus \{l\}\}$
39  return $\langle Q, \Sigma, \Delta, \Pi\rangle$
```

Fig. 1. Basic search algorithm of QUBE.

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 in $x_1 \ldots x_n$; we say that (1) is in *Disjunctive Normal Form* (DNF) when Φ is a disjunction of *terms* (or *cubes*), where each term is a conjunction of literals in $x_1 \ldots x_n$. We use the term *constraints* when we refer to clauses and terms indistinctly. 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 $\exists x \psi$ (resp. $\forall x \psi$), φ is satisfiable if and only if $\{\varphi\}_x$ or (resp. and) $\{\varphi\}_{\neg x}$ are satisfiable. If $\varphi = Qx\psi$ is a QBF and l is a literal with $|l| = x$, $\{\varphi\}_l$ is the QBF obtained from ψ by substituting l with $\top$ and $\bar{l}$ with $\bot$.

2.2 QUBE basic algorithm

In Figure 1 we present the pseudo-code of SOLVE, the basic search algorithm of QUBE. SOLVE generalizes the standard backtracking algorithm for QBFs as introduced in [14], by taking into account that clauses and terms can be dynamically learned during the search. Given a QBF (1) in CNF, SOLVE takes five parameters:

1. Q is the prefix, i.e., the list $Q_1x_1, \ldots, Q_nx_n$.
2. Σ is a set of clauses. Initially Σ is empty, but clauses are added to Σ as the search proceeds as result of the learning process, see [9].
3. Δ is the set of clauses corresponding to the matrix of the input formula.
4. Π is a set of terms. As for Σ, initially Π is empty, but terms are added to Π as the search proceeds as result of the learning process, see [9].
5. S is a consistent set of literals called *assignment*. Initially $S = \emptyset$.

In the following, as customary in search algorithms, we deal with constraints as if they were sets of literals, and assume that forall constraint c, it is not the case that x and $\neg x$ belong to c, for some atom x. Further, given $\langle Q, \Sigma, \Delta, \Pi\rangle$ corresponding to a QBF (1), we assume that the clauses and terms added to Σ and Π respectively, do not alter the correctness of the procedure. This amounts to say that, for any sequence of literals $l_1; \ldots; l_m$ with $m \leq n$ and $|l_i| = x_i$, the following three formulas are equi-satisfiable:

$$\{\ldots\{\{Q_1x_1Q_2x_2\ldots Q_nx_n(\wedge_{c\in\Delta}\vee_{l\in c} l)\}_{l_1}\}_{l_2}\ldots\}_{l_m},$$
$$\{\ldots\{\{Q_1x_1Q_2x_2\ldots Q_nx_n((\wedge_{c\in\Delta}\vee_{l\in c} l)\wedge(\wedge_{c\in\Sigma}\vee_{l\in c} l))\}_{l_1}\}_{l_2}\ldots\}_{l_m},$$
$$\{\ldots\{\{Q_1x_1Q_2x_2\ldots Q_nx_n((\wedge_{c\in\Delta}\vee_{l\in c} l)\vee(\vee_{t\in\Pi}\wedge_{l\in t} l))\}_{l_1}\}_{l_2}\ldots\}_{l_m}.$$

Consider a QBF φ corresponding to $\langle Q, \Sigma, \Delta, \Pi\rangle$.

At a high level of abstraction, SOLVE can be seen as a procedure generating the semantic tree corresponding to φ. The basic operation is thus that of assigning a literal l and simplifying φ accordingly, i.e., compute $\{\varphi\}_l$ and simplify it. In Figure 1, this task is performed by the function ASSIGN(l, Q, Σ, Δ, Π), which

– Removes $|l|$ and its bounding quantifier from the prefix (line 26),
– Removes all the clauses (resp. terms) to which l (resp. $\bar{l}$) pertains (lines 27-32). These clauses are said to be *eliminated* by l.
– Removes $\bar{l}$ (resp. l) from all the clauses (resp. terms) to which $\bar{l}$ (resp. l) pertains (lines 33-38). These clauses are said to be *simplified* by l.

In the following we say that a literal l is:

– *open* if $|l|$ is in Q, and *assigned* otherwise;
– *unit* if there exist a clause $c \in \Sigma \cup \Delta$ (resp. a term $t \in \Pi$) such that l is the only existential in c (resp. universal in t) and there are no literals in c (resp. in t) with an higher level. The *level of an atom* is 1 + the number of expressions $Q_jx_jQ_{j+1}x_{j+1}$ in Q with $j \geq i$ and $Q_j \neq Q_{j+1}$. The *level of a literal* l is the level of $|l|$.
– *monotone* if for all constraints $k \in (\Sigma \cup \Delta \cup \Pi)$, $\bar{l} \notin k$.

Now consider the routine LOOKAHEAD in Figure 1: $\{l\}_\exists$ (resp. $\{l\}_\forall$) denotes a constraint which is unit in l. The function LOOKAHEAD has the task of simplifying its input QBF by finding and assigning all unit (lines 16-18) and monotone literals (lines 19-23). Since assigning unit or monotone literals may cause the generation of new unit or monotone literals (this is different from the SAT case in which assigning monotone literals cannot generate new unit literals) LOOKAHEAD loops till no further simplification is possible (lines 14-15, 24).

The function SOLVE works in four steps:

1. Simplify the input instance with LOOKAHEAD (line 1).
2. Check if the termination condition is met (lines 2-3): if the test in line 2 is true, then S is a *solution*, while if the test in line 3 is true, then a S is a *conflict*; $\emptyset_\exists$ (resp. $\emptyset_\forall$) stands for the *empty clause* (resp. *empty term*), i.e., a constraint without existential (resp. universal) literals.
3. Choose heuristically a literal l (line 4) such that $|l|$ is at the highest level in the prefix. The literal returned by CHOOSE-LITERAL is called *branching literal*.
4. Assign the chosen literal (line 5), and recursively evaluate the resulting QBF (line 6):
   (a) if l is existential and the recursive evaluation yields TRUE then TRUE is returned (lines 7-8), otherwise
   (b) if l is universal and the recursive evaluation yields FALSE then FALSE is returned (lines 9-10), otherwise
   (c) $\bar{l}$ is assigned (line 12), and the result of the recursive evaluation of the resulting QBF is returned (line 13).

   It is easy to see that the execution of the code in lines 5-13 causes the generation of an AND-OR tree, whose *OR nodes* correspond to existential literals, while *AND nodes* correspond to universal literals.

SOLVE returns TRUE if the input QBF is satisfiable and FALSE otherwise.

For the sake of clarity we have presented SOLVE with recursive chronological backtracking. To avoid the expensive copying of data structures that would be needed to save Σ, Δ and Π at each node, QUBE (and most of the available systems as well) features a non-recursive implementation of the lookback procedure. The implementation is based on an explicit search stack and on data structures that can assign a literal during lookahead and then retract the assignment during lookback, i.e., restore Σ, Δ and Π to the configuration before the assignment was made. Further, as we already said in the introduction, QUBE differs from the above high-level description in that it features the techniques that are the subject of the next three sections.

2.3 Experimental setting

In order to evaluate the contribution of the different components to QUBE performances, we considered the 450 formal verification and planning instances that constituted part of the 2003 QBF evaluation[2]: 25% of these instances are from formal verification problems [1, 2], and the remaining are from planning domains [3, 4]. As we said in the introduction, these instances showed to be challenging for the 2003 QBF solvers comparative evaluation. Further, a subset of this testset was also included in the 2004 evaluation, and, according to some preliminary results, some of instances showed to be still hard to solve.

All the experiments were run on a farm of identical PCs, each one equipped with a PIV 3.2GHz processor, 1GB of RAM, running `Linux Debian 3.0`. Finally, each system had a timeout value of 900s per instance.

[2] With respect to the non-random instances used in the 2003 QBF comparative evaluation, our test set does not include the QBF encodings of the modal K formulas submitted by Guoqiang Pan [15].

3 Lookahead

Goal of the LOOKAHEAD function is to simplify the formula by propagating unit and monotone literals till no further simplification is possible. Any efficient implementation of a QBF solver has to rely on an efficient implementation of LOOKAHEAD. Indeed, most of the literals are assigned inside the function LOOKAHEAD: on our test set, if we consider the ratio R between

– the number of calls to ASSIGN made within LOOKAHEAD (lines 18, 21, 23).
– the number of calls to ASSIGN made within SOLVE (lines 5, 12), and

we have that R, on the problems that QUBE takes more than 1s to solve, is 514 on average, i.e., assigning one branching literal, causes hundreds of literals to be assigned inside LOOKAHEAD. Further, by running a profiler on QUBE, we have seen that on all the instances that we have tried, lookahead always amounted to more than 70% of the total runtime: this result echoes analogous remarks made in the SAT literature (see, e.g., [16]). Finally, the need for a fast lookahead procedure is accentuated by the use of learning [9], where the solver adds (possibly very long) constraints to the initial set given in input.

3.1 Algorithm

The implementation of LOOKAHEAD in QUBE is based on an extension of the lazy data structures as presented in [6], to detect and assign unit and monotone literals.

In particular, for detecting existential unit literals, we use the three literal watching (3LW) schema as described in [6]. For detecting universal unit literals, 3LW can be easily adapted to the case. Similar to the SAT case, all the literal watching schemes in [6] do not require to perform elimination of constraints, but only some of the simplifications (those being "watched") are performed when assigning a literal. However, compared to the other watching literal schemes in [6], 3LW performs less operations if in each constraint the existential and universal literals are listed separately, as it is the case in QUBE.

For monotone literal fixing (MLF), we implemented clause watching (CW) as described in [6]. However, the implementation of CW in QUBE posed some problems due to the interaction between monotone literals and learning (see [7]). The first observation is that the detection of monotone literals requires, when assigning a literal l, to perform also all the associated eliminations: These operations at least in part obscure the advantages of 3LW. Then, in order to reduce the burden of eliminating constraints, it is important to reduce the set of constraints to be considered. Problems arise because, given a QBF $\varphi = \langle Q, \Sigma, \Delta, \Pi \rangle$, it may be the case that for a literal l, the condition

$$\{k \in \Delta \mid \bar{l} \in k\} = \emptyset \qquad (2)$$

is satisfied, while

$$\{k \in \Sigma \cup \Delta \cup \Pi \mid \bar{l} \in k\} = \emptyset \qquad (3)$$

is not. Thus, a literal l may be assigned as monotone because it does not occur in the matrix of the input formula, but assigning l to true may cause (i) the simplification of some learned constraint, (ii) the generation of an empty clause or an empty term, and (iii) the presence of $\bar{l}$ in the reason associated to the empty clause/term. This last fact would require the computation of a reason for monotone literals, to be used while backtracking. While from a theoretical point of view it is possible to compute such a reason, it is still an open issue how to do it efficiently, and [7] describes some problems that point out that it cannot be done in all cases.

Summing up, problems arise when

- l is monotone, existential and $\bar{l}$ belongs to a working reason originating from a conflict, or
- l is monotone, universal and $\bar{l}$ belongs to a working reason originating from a solution.

As discussed in [9], if condition (3) is satisfied then assigning l as monotone is not problematic meaning that it is always possible to compute the working reason wr in order to avoid $\bar{l} \in wr$. Still, checking condition (3) is not practical because, when assigning a literal l, would require to eliminate all the corresponding constraints in $\Sigma \cup \Delta \cup \Pi$ (or at least, from $\Sigma \cup \Delta$ if l is existential, and from $\Sigma \cup \Pi$ if l is universal).

The solution that we have adopted for MLF in QUBE is to assign a literal as monotone when condition (2) is satisfied, and then to temporarily delete from $\Sigma \cup \Pi$ all the clauses where $\bar{l}$ occurs. This solves all the above mentioned problems, and it has the following advantages:

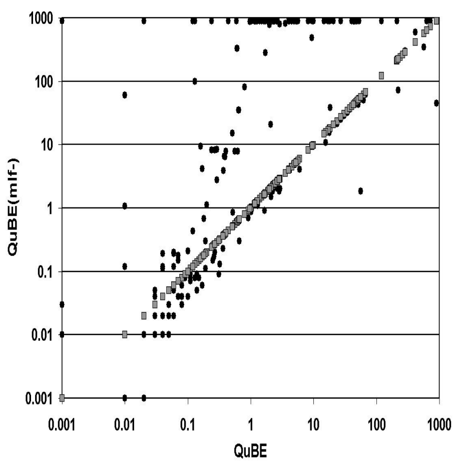
1. It allows to assign as monotone more literals than those that would be assigned according to condition (3), and
2. From a computational point of view, it is far less expensive since, when assigning a literal l it only requires to eliminate constraints in Δ.

3.2 Effectiveness

It is well known that unit literal propagation is fundamental for efficiency, while, at least in the SAT setting, MLF is often considered to be inefficient, and indeed, it is not implemented by most of the state-of-the-art SAT solvers like ZCHAFF.

The performances of QUBE when run with and without MLF (we will call the resulting system QUBE(MLF$^-$)) is shown in Figure 2. In the left plot, the x-axis is the CPU-time of QUBE and the y-axis is the CPU-time of QUBE(MLF$^-$). A plotted point $\langle x, y\rangle$ represents a benchmark on which QUBE and QUBE(MLF$^-$) take x and y seconds respectively.[3] For convenience, we also plot the points $\langle x, x\rangle$, each representing the benchmarks solved by QUBE in x seconds. As it can be seen from the figure, QUBE(MLF$^-$) is faster than QUBE on many benchmarks (96), represented by the points below the diagonal. However, these points are mostly located at the beginning of the plot, and represent instances that are solved in less than a second. Indeed, assigning a literal when MLF is enabled is more expensive and, on easy instances, MLF does

[3] In principle, one point $\langle x, y\rangle$ could correspond to many benchmarks solved by QUBE and QUBE(MLF$^-$) in x and y seconds respectively. However, in this and the other scatter diagrams that we consider, each point (except for the point $\langle 900, 900\rangle$, representing the instances on which both solvers time-out) corresponds to a single instance in most cases.



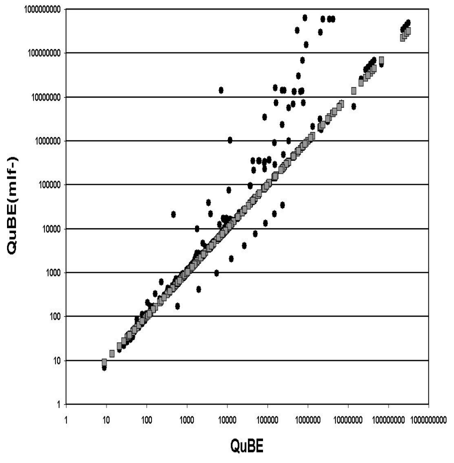


Fig. 2. Effectiveness of monotone literal fixing: CPU time (left) and number of assignment (right)

not pay off. Still, MLF can greatly cut the search tree. This is evident from the right plot in the figure, showing the number of assignments made by QUBE(MLF$^-$) wrt QUBE when considering the instances solved by both solvers: There are only 10 clearly visible points below the diagonal, meaning that the 96 problems on which QUBE(MLF$^-$) is faster than QUBE are due to the burden of MLF. Of these 96 problems,

– there is only one problem (represented by the point on the vertical axis at the extreme right) which is solved by QUBE(MLF$^-$) and on which QUBE timeouts, and
– among the instances that are solved by both solvers, QUBE(MLF$^-$) is faster than QUBE of at least one order of magnitude on only 3 instances, two of which solved by QUBE in less than 0.1s.

On the other hand, (i) QUBE is able to solve 65 instances not solved by QUBE(MLF$^-$), and (ii) QUBE is at least one order of magnitude faster than QUBE(MLF$^-$) on 31 other problems. These and other numbers are reported in Table 1.a.

4 Heuristic

In SAT, it is well known that the branching heuristic is important for efficiency. In the QBF setting, the situation is different. Indeed, we are only allowed to choose a literal l if $|l|$ is at the highest level in the prefix. Thus, on a QBF of the form

$$\exists x_1 \forall x_2 \exists x_3 ... \forall x_{n-1} \exists x_n \Phi \tag{4}$$

the heuristic is likely to be (almost) useless: unless atoms are removed from the prefix because unit or monotone, the atom to pick at each node is fixed. On the other hand, on a QBF of the form

$$\exists x_1 \exists x_2 \ldots \exists x_m \Phi \tag{5}$$

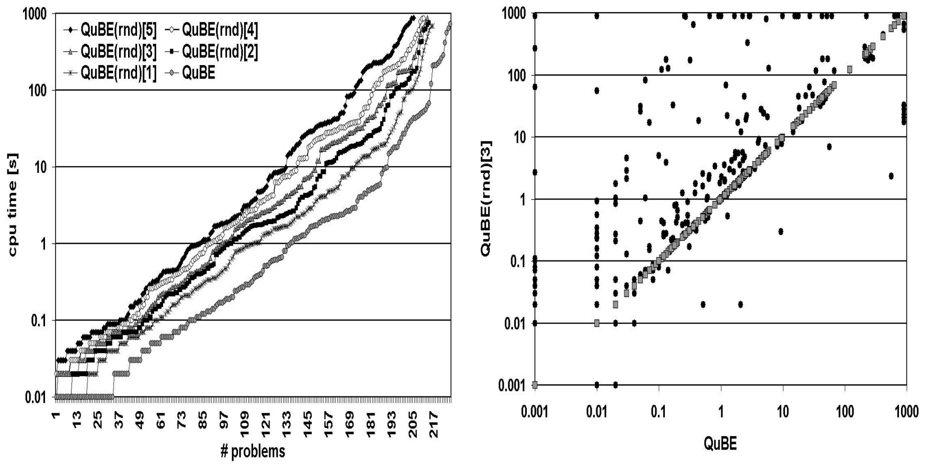


Fig. 3. Effectiveness of the heuristics: overall (left) and best (right)

corresponding to a SAT instance, it is likely that the heuristic will play an important role. Indeed, we expect the role of the heuristic to have more and more importance as the number of alternations (i.e., expressions of the form $Q_j x_j Q_{j+1} x_{j+1}$ with $Q_j \neq Q_{j+1}$) in Q is small compared to the number of variables. In (4) the number of alternations is the number of variables -1, while in (5) is 0. In practice, the number of alternations is in between the two extreme cases represented by the above two equations. In many cases, it is 1 or 2, and thus we expect the heuristic to be important.

In this section, we show that this is the case for QuBE heuristic, meaning that it performs consistently better, on average, than a simple random heuristic.

4.1 Algorithm

Even in SAT, where the number of alternations is 0, the design of an heuristic has to be a trade-off between accuracy and speed. VSIDS (Variable State Independent Decaying Sum) [16] is now at the basis of most recent SAT solvers for real world problems. The basic ideas of VSIDS are to (i) initially rank literals on the basis of the occurrences in the matrix, (ii) increment the weight of the literals in the learned constraints, and (iii) periodically divide by a constant the weight of each literal.

In our QBF setting, the above needs to be generalized taking into account the prefix and also the presence of both learned constraints and terms. In QuBE this is done by periodically sorting literals according to (i) the prefix level of the corresponding atom, (ii) their score, and (iii) their numeric ID. The score of each literal l is computed as follows:

- initially, it is set to the number of clauses in which l belongs,
- at the $i+1$ step, the score is computed by summing the score at the previous step divided by two, and the number of constraints c such that

- $l \in c$, if l is existential, and
- $\bar{l} \in c$, if l is universal.

When the input QBF corresponds to a SAT formula, the above heuristic boils down to VSIDS.

4.2 Effectiveness

To evaluate the role of the heuristic in QuBE, we compared it with QuBE(RND), i.e., QuBE with a heuristic which randomly select a literal at the highest prefix level. Because of the randomness, we run QuBE(RND) 5 times on each instance. Given an instance φ, if we order the time spent to solve φ from the best (1) to the worst (5), then we can define QuBE(RND)[i] to be the system whose performance on φ is the i-th best among the 5 results. The results of QuBE, QuBE(RND)[1-5] are plotted in Figure 3 left. In the figure, the results of each solver are (i) sorted independently and in ascending order, (ii) filtered out by removing the first 148 and the last 76 values, and (iii) plotted against an ordinal in the range [1-226] on the x-axis. The filtering has been done in order to increase the readability of the figure. Indeed, each solver (i) is able to solve at least 148 problems in a time $\geq$ 0.02s, and (ii) timeouts on at least 76 values. Thus, if a point $\langle x, y \rangle$ belongs to the plot of a system S, this means that $x + 148$ instances are solved in less than y seconds by S.

Several observations are in order. The first one is that the heuristic plays a role. This is evident if we consider the five plots of QuBE(RND)[1-5], which show that there can be significant differences among different runs of QuBE with a random heuristics. The second observation is that QuBE is better than QuBE(RND)[1], i.e., the solver among QuBE(RND)[1-5] having the best result on each single instance:

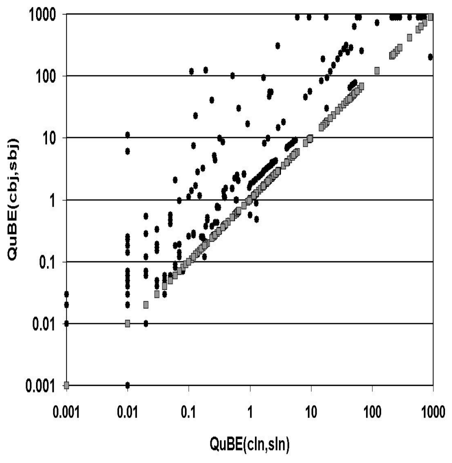
– QuBE (resp. QuBE(RND)[1]) is able to solve 19 (resp. 9) instances that are not solved by QuBE(RND)[1] (resp. QuBE),
– among the instances solved by both solvers, QuBE (resp. QuBE(RND)[1]) is at least one order of magnitude faster than QuBE(RND)[1] (resp. QuBE) on 41 (resp. 10) instances.

The above data can be seen on the right plot, representing the performances of QuBE versus QuBE(RND)[1]. From the right plot, it also emerges that QuBE(RND)[1] is faster on many instances (115). Of course, this number goes down to 69 and 32 if we consider QuBE(RND)[3] or QuBE(RND)[5]. Still, the presence of 32 problems (roughly 8% of the problems that are solved by at least one solver) in which QuBE(RND)[5] is faster than QuBE points out that there are QBFs in which the heuristic does not seem to play any role. On these problems, QuBE pays the overhead of periodically computing the score of each literal, and sort them according to the above outlined criteria.

For more data, see Table 1.

5 Learning

Learning is a look-back strategy whose effectiveness for solving real-world problems is a consolidated result in the SAT literature (see, e.g., [17, 18, 16]). In the QBF setting, mixed results have been so far obtained, see, e.g., [9, 11, 10, 13, 19]. In particular,



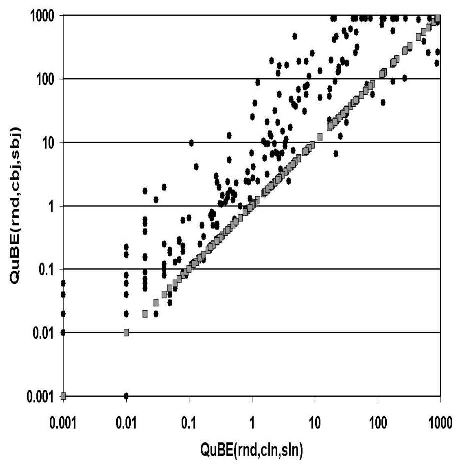


Fig. 4. Effectiveness of learning: with a VSIDS (left) and a random heuristic (right).

in [19] it is argued that while learning "conflicts" (computed while backtracking from an empty clause) leads to the expected positive results, learning "solutions" (computed while backtracking from an empty term or the empty matrix) does not always produce positive results, especially on real-world benchmarks.

Here we show that both conflict and solution learning are essential for QUBE performances.

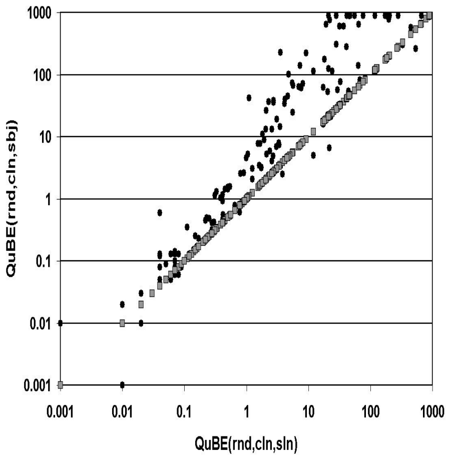
5.1 Algorithm

Learning amounts to store clauses (resp. terms) computed while backtracking by performing clause (resp. term) resolution[4] between a "working reason" that is initially computed when an empty clause (resp. an empty term or the empty matrix) is found, and the "reason" corresponding to unit literals that are stored while descending the search tree, see [21].

As in SAT, two of the key issues are the criteria used for deciding when a constraint has to be learned (i.e., stored in Σ/Π), and then unlearned (i.e., removed from Σ/Π). Learning in QUBE works as follows. Assume that we are backtracking on a literal l having decision level n, i.e., such that there are n AND-OR nodes before l. The constraint corresponding to the current working reason wr is learned if and only if:

- l is existential (resp. universal) if we are backtracking from a conflict (resp. solution),
- all the assigned literals in wr but l, have a decision level strictly smaller than n, and
- there are no open universal (resp. existential) literals in wr that are before l in the prefix.

[4] Clause resolution is called Q-resolution in [20].



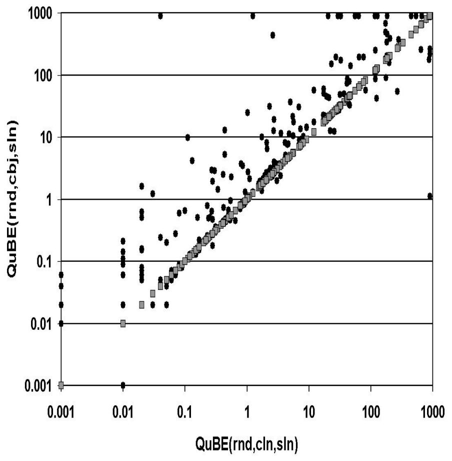


Fig. 5. Effectiveness of conflict (left) and solution (right) learning.

These conditions ensure that l is unit in the constraint corresponding to the reason. Once the constraint is learned, QUBE backjumps to the node corresponding to the literal $l' \neq l$ in wr with maximum decision level. Notice that on a SAT instance, QUBE learning mechanism behaves similarly to the "1-UIP-learning" scheme used in ZCHAFF.

For unlearning, QUBE uses a *relevance bounded learning (of order r)* schema as introduced in SAT by [18]: The set of learned constraints is periodically scanned and the constraints having more than r open literals are deleted.

In QBF, another key issue is the initialization of the working reason wr, especially when a solution is found (when a conflict is found, we can do as in SAT). In the case of a solution, in QUBE, our approach is to first set wr to a set of literals having the following properties:

– it is a subset of the assignment S that led to the current solution,
– it is a prime implicant of the matrix of the input QBF,
– is such that there does not exist another set of literals satisfying the first two properties and
  • with a smaller (under set inclusion) set of universal literals, or
  • causing a deeper backtrack.

Then, some extra computation is done in order to try to further reduce the universal literals in wr. See [21] for more details.

5.2 Effectiveness

Figure 4 left shows the performances of QUBE versus QUBE(CBJ,SBJ), i.e., QUBE without learning but both conflict and solution backjumping enabled [22]. The first consideration is that learning pays off:

– QuBE (resp. QuBE(CBJ,SBJ)) is able to solve 16 (resp. 1) instances that are not solved by QuBE(CBJ,SBJ) (resp. QuBE),
– among the instances solved by both solvers, QuBE (resp. QuBE(CBJ,SBJ)) is at least one order of magnitude faster than QuBE(CBJ,SBJ) (resp. QuBE) on 39 (esp. 0) instances.

Still, the above data are not entirely satisfactory for two reasons.

First, in QuBE learning and the heuristic are tightly coupled: Whenever QuBE learns a constraint, it also increments the score of the literals in it. In QuBE(CBJ,SBJ) no constraint is ever learned. As a consequence, in QuBE(CBJ,SBJ), (i) literals are initially sorted on the basis of their occurrences in the input QBF, and (ii) the score of each literal is periodically halved till it becomes 0. When all the literals have score 0, then literals at the same prefix level are chosen according to their lexicographic order.

Second, independently from the heuristic being used, a plot showing the performances of QuBE with and without learning, does not say which of the two learning schemes (conflict, solution) is effective [19].

To address the first problem, we consider QuBE with a random heuristic, with and without learning: We call the resulting systems QuBE(RND) and QuBE(RND,CBJ,SBJ) respectively. As in the previous section, we run each system 5 times on each benchmark, and we introduce the systems QuBE(RND)[i] and QuBE(RND,CBJ,SBJ)[i] ($1 \leq i \leq 5$)). The results for QuBE(RND)[3] and QuBE(RND,CBJ,SBJ)[3] are plotted in Figure 4 right. From the plot, it is easy to see that QuBE(RND)[3] is faster than QuBE(RND,CBJ,SBJ)[3] in most cases. To witness this fact

– QuBE(RND) (resp. QuBE(RND,CBJ,SBJ)) is able to solve 21 (resp. 2) instances that are not solved by QuBE(CBJ,SBJ) (resp. QuBE),
– among the instances solved by both solvers, QuBE (resp. QuBE(CBJ,SBJ)) is at least one order of magnitude faster than QuBE(CBJ,SBJ) (resp. QuBE) on 68 (esp. 2) instances.

Comparing with the results on the left plot, it seems that with a random heuristic learning becomes more important. This fact witnesses also in our setting the well-known tension between look-ahead and look-back techniques: Using a “smart” look-ahead makes the look-back less important, and viceversa. Still, because of the randomness in the systems, 5 runs are too few in order to draw precise quantitative conclusions. Still, at from a qualitative point of view, it is clear that learning can produce significant speed-ups.

To address the second problem, we considered the systems QuBE(RND,CBJ,SLN) and QuBE(RND,CLN,SBJ), i.e., the systems obtained from QuBE(RND) by disabling conflict learning and solution learning respectively. As usual we run each system 5 times on each instance, and we define QuBE(RND,CBJ,SLN)[i] and QuBE(RND,CLN,SBJ)[i] ($1 \leq i \leq 5$) as before. Figure 5 shows the performances of QuBE(RND)[3] versus QuBE(RND,CBJ,SLN)[3] (left plot) and QuBE(RND,CLN,SBJ)[3] (right plot). From the plots, we see that both conflict and solution learning pay off. In each plot, there are only a few points well below the diagonal. Further, by comparing the two plots, it seems that solution learning is more effective than conflict learning, but again we have to take this as qualitative indication. Some detailed quantitative information is reported in Table 1.b. From the table, we can see that the overall performances of QuBE(RND)[i]

Table 1.a

QuBE	$=$	$<$	$>$	$\ll$	$\gg$	$\bowtie$	$\times 10<$	$\times 0.1>$	TO
QuBE(MLF$^-$)	141	73	95	65	1	75	31	3	140
QuBE(RND)[1]	137	112	106	19	9	67	41	10	86
QuBE(RND)[2]	134	139	80	21	9	67	44	6	88
QuBE(RND)[3]	126	166	60	22	9	67	53	5	89
QuBE(RND)[4]	124	189	37	24	9	67	62	4	91
QuBE(RND)[5]	108	213	24	29	8	68	79	4	97
QuBE(CBJ,SBJ)	146	181	31	16	1	75	39	0	91

Table 1.b

QuBE(RND)[3]	$=$	$<$	$>$	$\ll$	$\gg$	$\bowtie$	$\times 10<$	$\times 0.1>$	TO
QuBE(RND)[1]	136	0	225	0	3	86	0	43	86
QuBE(RND)[2]	169	0	192	0	1	88	0	19	88
QuBE(RND)[4]	156	203	0	2	0	89	27	0	91
QuBE(RND)[5]	109	244	0	8	0	89	61	0	97
QuBE(RND,CBJ,SBJ)[1]	131	145	72	13	7	82	27	20	95
QuBE(RND,CBJ,SBJ)[2]	137	164	43	17	2	87	43	7	104
QuBE(RND,CBJ,SBJ)[3]	123	192	25	21	2	87	68	2	108
QuBE(RND,CBJ,SBJ)[4]	110	205	17	29	2	87	83	1	116
QuBE(RND,CBJ,SBJ)[5]	84	222	10	45	2	87	99	1	132
QuBE(RND,CBJ,SLN)[1]	130	96	128	7	5	84	20	26	91
QuBE(RND,CBJ,SLN)[2]	133	134	82	12	5	84	27	14	96
QuBE(RND,CBJ,SLN)[3]	129	169	48	15	3	86	40	5	101
QuBE(RND,CBJ,SLN)[4]	115	209	20	17	1	88	54	1	105
QuBE(RND,CBJ,SLN)[5]	86	245	6	24	1	88	87	0	112
QuBE(RND,CLN,SBJ)[1]	135	78	142	6	4	85	7	36	91
QuBE(RND,CLN,SBJ)[2]	151	110	90	10	4	85	15	15	95
QuBE(RND,CLN,SBJ)[3]	169	134	39	19	1	88	29	5	107
QuBE(RND,CLN,SBJ)[4]	141	183	11	26	0	89	51	0	115
QuBE(RND,CLN,SBJ)[5]	103	218	2	38	0	89	69	0	127

Table 1. Comparison among various versions of QuBE. In each table, the comparison considers a system taken as reference and written in the top left box in the table: QuBE in Table 1.a, and QuBE(RND)[3] in Table 1.b. In each table, if A is the system taken as reference in it, and $B \neq A$ is a solver in the first column, then the other columns report the number of problems that: "$=$", A and B solve in the same time; "$<$", A and B solve but A takes less time than B; "$>$", A and B solve but A takes more time than B; "$\ll$", A solves while B does not; "$\gg$", A does not solve while B does; "$\bowtie$", A and B do not solve; "$\times 10<$", both A and B solve but on which A is at least one order of magnitude faster; "$\times 0.1<$", both A and B solve but on which A is at least one order of magnitude slower; "TO", B does not solve. The number of timeouts for QuBE and QuBE(RND)[3] is 76 and 89 respectively.

are better than the performances of QuBE(RND,CBJ,SLN)[i], QuBE(RND,CLN,SBJ)[i] and QuBE(RND,CBJ,SBJ)[i]. The above positive results for solution learning are confirmed if we compare the number of solutions found by QuBE(RND,CBJ,SLN)[i] and QuBE(RND,CBJ,SBJ)[i] as in [19]: For example, considering the instances solved by both solvers in more than 1s and for which at least one solution is found by both, the average (resp. maximum) of the ratio between the number of solutions found by QuBE(RND,CBJ,SBJ)[3] and QuBE(RND,CBJ,SLN)[3] is 5.4 (resp. 42.4). The negative results reported in [19] for solution-based look-back mechanisms are not comparable with ours, given the different mechanisms implemented by the respective solvers (e.g., for computing the initial solution and for monotone literal fixing), and the different experimental setting (e.g., the testset).

6 Concluding remarks

Given the results in Table 1.a, we can say that all the techniques contribute to the effectiveness of QUBE. In terms of improving the *capacity* (i.e., the ability to solve problems [12]), the most effective one seems to be MLF, followed by learning and the heuristic: QUBE solves 64, 15 and 13 instances more than QUBE(MLF$^-$), QUBE(CBJ,SBJ) and QUBE(RND) respectively. In terms of improving the *productivity* (i.e., the ability to quickly solve problems [12]) the picture seems to be the opposite. The most effective technique is the heuristic, then learning and finally MLF: the difference between

- the number of problems in which QUBE is at least 1 order of magnitude faster, and
- the number of problems in which QUBE is at least 1 order of magnitude slower

than QUBE(RND), QUBE(CBJ,SBJ) and QUBE(MLF$^-$) is 48, 39 and 28 respectively. In the above statements, we used the phrase "seems to be" to stress once more that the numbers in the table have to be taken as indications of average behaviors because of the randomness of some of the solvers that we considered. Still, the fact that MLF increases capacity more than productivity, and that for the heuristic the situation is the opposite matches our intuition: Indeed, the bottleneck of QUBE and of all the solvers based on [14] is that the search tree is explored taking into account the prefix. Any look-ahead mechanism that, like MLF, overrides the "prefix-rule" may greatly improve the capacity of the solver, because it may allow to assign literals (i) that are fundamental for quickly deciding the formula (un)satisfiability, and (ii) that would have been assigned much later without it. Still, such look-ahead mechanisms have always an associated overhead, which may worsen the productivity. Other examples of look-ahead mechanisms having the above characteristics are the partial instantiation techniques described in [23]: Indeed, the results reported in [23] support our conclusions. The heuristic on the other hand, may still produce exponential speed-ups, but it does not address the main bottleneck of search algorithms based on [14]. Thus, the heuristic may improve performances and thus productivity, but we expect that in some domains many instances (that would be solvable with a proper look-ahead mechanism) will remain unsolvable no matter the heuristic being used.

References

1. C. Scholl and B. Becker. Checking equivalence for partial implementations. In *38th Design Automation Conference (DAC'01)*, 2001.
2. 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.
3. Jussi Rintanen. Constructing conditional plans by a theorem prover. *Journal of Artificial Intelligence Research*, 10:323–352, 1999.
4. Claudio Castellini, Enrico Giunchiglia, and Armando Tacchella. Improvements to SAT-based conformant planning. In *Proc. ECP*, 2001.
5. Maher N. Mneimneh and Karem A. Sakallah. Computing vertex eccentricity in exponentially large graphs: QBF formulation and solution. In *Theory and Applications of Satisfiability Testing, 6th International Conference, (SAT)*, volume 2919 of *LNCS*, pages 411–425. Springer, 2004.

6. I. Gent, E. Giunchiglia, M. Narizzano, A. Rowley, and A. Tacchella. Watched data structures for QBF solvers. In Enrico Giunchiglia and Armando Tacchella, editors, *Theory and Applications of Satisfiability Testing, 6th International Conference, (SAT)*, volume 2919 of *LNCS*, pages 25–36. Springer, 2004.
7. E. Giunchiglia, M. Narizzano, and A. Tacchella. Monotone literals and learning in QBF reasoning. In *Tenth International Conference on Principles and Practice of Constraint Programming, CP 2004*, 2004.
8. E. Giunchiglia, M. Narizzano, and A. Tacchella. QUBE: an efficient QBF solver. In *5th International Conference on Formal Methods in Computer-Aided Design, FMCAD 2004*, 2004.
9. Enrico Giunchiglia, Massimo Narizzano, and Armando Tacchella. Learning for Quantified Boolean Logic Satisfiability. In *Proc. 18th National Conference on Artificial Intelligence (AAAI) (AAAI'2002)*, pages 649–654, 2002.
10. 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.
11. 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.
12. Fady Copty, Limor Fix, Enrico Giunchiglia, Gila Kamhi, Armando Tacchella, and Moshe Vardi. Benefits of bounded model checking at an industrial setting. In *Proc. 13th International Computer Aided Verification Conference (CAV)*, 2001.
13. 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 *LNCS*. Springer Verlag, 2003.
14. M. Cadoli, A. Giovanardi, and M. Schaerf. An algorithm to evaluate quantified boolean formulae. In *Proc. AAAI*, 1998.
15. Guoqiang Pan and Moshe Y. Vardi. Optimizing a BDD-based modal solver. In *Proceedings of the 19th International Conference on Automated Deduction*, 2003.
16. Matthew W. Moskewicz, Conor F. Madigan, Ying Zhao, Lintao Zhang, and Sharad Malik. Chaff: Engineering an Efficient SAT Solver. In *Proceedings of the 38th Design Automation Conference (DAC'01)*, June 2001.
17. J. P. Marques-Silva and K. A. Sakallah. GRASP - A New Search Algorithm for Satisfiability. In *Proceedings of IEEE/ACM International Conference on Computer-Aided Design*, pages 220–227, November 1996.
18. Roberto J. Bayardo, Jr. and Robert C. Schrag. Using CSP look-back techniques to solve real-world SAT instances. In *Proceedings of the 14th National Conference on Artificial Intelligence and 9th Innovative Applications of Artificial Intelligence Conference (AAAI-97/IAAI-97)*, pages 203–208, Menlo Park, July 27–31 1997. AAAI Press.
19. Ian P. Gent and Andrew G.D. Rowley. Solution learning and solution directed backjumping revisited. Technical Report APES-80-2004, APES Research Group, February 2004. Available from http://www.dcs.st-and.ac.uk/~apes/apesreports.html.
20. H. Kleine-Büning, M. Karpinski, and A. Flögel. Resolution for quantified Boolean formulas. *Information and computation*, 117(1):12–18, 1995.
21. E. Giunchiglia, M. Narizzano, and A. Tacchella. Clause-term resolution and learning in quantified Boolean logic satisfiability, 2004. Submitted.
22. Enrico Giunchiglia, Massimo Narizzano, and Armando Tacchella. Backjumping for Quantified Boolean Logic Satisfiability. *Artificial Intelligence*, 145:99–120, 2003.
23. Jussi Rintanen. Partial implicit unfolding in the Davis-Putnam procedure for Quantified Boolean Formulae. In *Proc. LPAR*, volume 2250 of *LNCS*, pages 362–376, 2001.