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) has been accompanied by a steady supply of real-world instances, i.e., QBFs originated by translations from application domains. Instances of this kind showed to be challenging for current state-of-the-art QBF solvers, while the ability to deal effectively with them is necessary to foster adoption of QBF-based reasoning in practice. In this paper we describe three reasoning techniques that we implemented in our solver QUBE++ to increase its performances on real-world instances coming from formal verification and planning domains. We present experimental results that witness the contribution of each technique and the better performances of QUBE++ with respect to other state-of-the-art QBF solvers. The effectiveness of QUBE++ is further confirmed by experiments run on challenging real-world SAT instances, where QUBE++ turns out to be competitive with respect to current state-of-the-art SAT solvers.

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. Following the standard practice of SAT competitions [1, 2], we consider as real-world the instances originated by translations from application domains such as, e.g., Formal Verification [3, 4], Planning [5, 6], and Reasoning about Knowledge [7]. In the last evaluation of QBF solvers [8], instances of this kind emerged as challenging benchmarks for the current state-of-the-art tools: real-world benchmarks represented about 50% of the evaluation test set, and they constituted about 95% of the "hard" instances, i.e., problems that could not be solved by any of the participants within the allotted time. On the other hand, improving the capabilities of QBF tools for reasoning on real-world QBF instances is necessary to foster the adoption of QBF-based approaches in the context of suitable application domains.

In this paper we describe three reasoning techniques that we implemented into our solver QUBE++ to increase its effectiveness on instances coming from formal verification and planning domains, two domains we are particularly interested in. The techniques are: (i) efficient detection of unit and monotone literals using lazy data structures as in [9]; (ii) learning, as introduced in [10], with improvements that generalize the techniques first used in SAT by GRASP [11]; (iii) a branching strategy that exploits information gleaned from the input formula initially, and leverages the information extracted in the learning phase. Using formal verification and planning benchmarks from the QBF evaluation [8], we conducted an experimental campaign to highlight the relative efficiency of the above techniques, and of their combination: the results show that each single technique contributes to the effectiveness of QUBE++ and that the resulting system is a clear improvement over current state-of-the-art QBF solvers. QUBE++ is able to solve 10% of the instances that defied all the participants in the QBF evaluation, and it is up to one, two, and nearly three orders of magnitude faster than, respectively, QUBEREL, QUBEBJ and SEMPROP, i.e., the best QBF solvers on non-random instances according to [8]. We have also tried QUBE++ on a test set of challenging real-world SAT benchmarks together with QUBEREL, the previous best version of QUBE, and ZCHAFF, the winner of SAT 2002 competition [1] and one of the best solvers in SAT 2003 competition [2] on real-world SAT instances. The results show that QUBE++ is able to conquer about 90% of the test set, and it loses, medianly, only a factor of two from ZCHAFF, while QUBEREL, conquers only 50% of the test set and it is, medianly, one order of magnitude slower than ZCHAFF.

The paper is structured as follows. We first present some formal preliminaries and the basic algorithm of QUBE++. We devote three sections to the improvements on lookahead techniques, learning and the branching strategy, respectively. We comment the results of the experiments outlining the effectiveness of each technique and of the resulting system. We end the paper with some remarks.

* 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.

```
bool SOLVE(Q, Σ, Π, S)
 1 do
 2   ⟨Q', Σ', Π', S'⟩ ← ⟨Q, Σ, Π, S⟩
 3   ⟨Q, Σ, Π, S⟩ ← LOOKAHEAD(Q', Σ', Π', S')
 4 while ⟨Q, Σ, Π, S⟩ ≠ ⟨Q', Σ', Π', S'⟩
 5 if Σ = ∅ or ∅∀ ∈ Π then return TRUE
 6 if ∅∃ ∈ Σ and Π = ∅ then return FALSE
 7 l ← CHOOSE-LITERAL(Q, Σ, Π)
 8 if l is existential then
 9   return SOLVE(Q, Σ ∪ {l}, Π, S) or
             SOLVE(Q, Σ ∪ {l̄}, Π, S)
10 else
11   return SOLVE(Q, Σ, Π ∪ {l̄}, S) and
             SOLVE(Q, Σ, Π ∪ {l}, S)
```

```
set LOOKAHEAD(Q, Σ, Π, S)
12 while {l}∃ ∈ Σ or {l̄}∀ ∈ Π do
13   S ← S ∪ {l}
14   Q ← REMOVE(Q, |l|)
15   for each c ∈ Σ s.t. l ∈ c do
16     Σ ← Σ \ {c}
17   for each t ∈ Π s.t. l̄ ∈ t do
18     Π ← Π \ {t}
19   for each c ∈ Σ s.t. l̄ ∈ c do
20     Σ ← (Σ \ {c}) ∪ {c \ {l̄}}
21   for each t ∈ Π s.t. l ∈ t do
22     Π ← (Π \ {t}) ∪ {t \ {l}}
23 for each l s.t. l̄ ∉ k for all k ∈ (Σ ∪ Π) do
24   if l is existential then Σ ← Σ ∪ {l}
25   else Π ← Π ∪ {l}
26 return ⟨Q, Σ, Π, S⟩
```

Fig. 1. Basic search algorithm of QUBE++.

2 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 $\bar{l}$ denotes the *complement* of l, i.e., if $l = a$ then $\bar{l} = \neg a$, and if $l = \neg a$ then $\bar{l} = a$, 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

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

where every Q_i ($1 \leq i \leq n$) is a quantifier, either existential $\exists$, or universal $\forall$; $x_1 \dots x_n$ are distinct atoms in P, and Φ is a propositional formula. $Q_1 x_1 Q_2 x_2 \dots Q_n x_n$ is the *prefix* and Φ is the *matrix* of (1). A literal l is *existential*, if $\exists|l|$ is in the prefix, and *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 $x_1 \dots x_n$; 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 in $x_1 \dots x_n$. 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 φ_x or (resp. and) $\varphi_{\neg x}$ are satisfiable. If $\varphi = Qx\psi$ is a QBF and l is a literal, φ_l is the QBF obtained from ψ by substituting l with $\top$ and $\bar{l}$ with $\bot$.

3 QUBE++

In Figure 1 we present the pseudo-code of SOLVE, the basic search algorithm of QUBE++. SOLVE generalizes standard backtrack algorithms for QBFs by allowing mixed CNF/DNF instances, i.e., instances of the kind $Q_1 x_1 \dots Q_n x_n \Phi$, where $\Phi = \Psi \vee \Theta$, Ψ is a conjunction of clauses, and Θ is a disjunction of terms. Even assuming that Θ is initially $\bot$, formulas of this kind arise during the learning process as shown in [10]. SOLVE returns TRUE if the input QBF is satisfiable and FALSE otherwise. In Figure 1, one can see that SOLVE takes four parameters: Q is the prefix, i.e., the list $Q_1 x_1, \dots, Q_n x_n$, Σ is the set of clauses corresponding to Ψ, Π is the set of terms corresponding to Θ, and S is a 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. SOLVE works in three steps (line numbers refer to Figure 1):

1. Simplify the input instance with LOOKAHEAD (lines 1-4): LOOKAHEAD is iterated until no further simplification is possible.
2. Check if the termination condition is met (lines 5-6): if the test in line 5 is true, then S is a *solution*, while if the test in line 6 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 comprised of universal (resp. existential) literals only.

3. Choose heuristically a literal l (line 7) such that (i) $|l|$ is in Q, and (ii) there is no other literal l' not having the same quantifier of $|l|$, and occurring before l in the prefix; 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 9), otherwise an *AND node* is explored (line 11).

Consider an instance $\langle Q, \Sigma, \Pi \rangle$. 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 $c \in \Sigma$ (resp. a term $t \in \Pi$) 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 $|l'|$ is before $|l|$ in the prefix;
- *monotone* if for all constraints $k \in (\Sigma \cup \Pi)$, $\bar{l} \notin k$.

Now consider the simplification routine LOOKAHEAD in Figure 1: $\{l\}_\exists$ (resp. $\{l\}_\forall$) denotes a constraint which is unit in l, and REMOVE(Q,x_i) returns the prefix obtained from Q by removing $Q_i x_i$. The function LOOKAHEAD has the task of finding and assigning all unit and monotone literals at every node of the search tree. LOOKAHEAD loops until either Σ or Π contains a unit literal (line 12). Each unit literal l is added to the current assignment (line 13), removed from Q (line 14), and then it is assigned by:

- removing all the clauses (resp. terms) to which l (resp. $\bar{l}$) pertains (lines 15-18), and
- removing $\bar{l}$ (resp. l) from all the clauses (resp. terms) to which $\bar{l}$ (resp. l) pertains (lines 19-22).

We say that an assigned literal l (i) *eliminates* a clause (resp. a term) when l (resp. $\bar{l}$) is in the constraint, and (ii) *simplifies* a clause (resp. a term) when $\bar{l}$ (resp. l) is in the constraint. After assigning all unit literals, LOOKAHEAD checks and propagates any monotone literal.

For the sake of clarity we have presented QUBE++ with recursive chronological backtracking. To avoid the expensive copying of data structures that would be needed to save Σ and Π at each node, QUBE++ 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 and Π to the configuration before the assignment was made.

4 Optimized lookahead

As reported by [12] in the case of SAT instances, a major portion of the runtime of the solver is spent in the lookahead process. Running a profiler on a DPLL-based QBF solver like QUBE++ confirms this result: on all the instances that we have tried, lookahead always amounted to more than 70% of the total runtime. The need for a fast lookahead procedure is accentuated by the use of smart lookback techniques such as learning [10], where the solver augments the initial set of constraints with other ones discovered during the search. With learning, possibly large amounts of lengthy constraints have to be processed quickly, otherwise the overhead will dwarf the benefits of learning itself. For these reasons, the implementation of LOOKAHEAD in QUBE++ is based on an extension of the three literal watching (3LW) and the clause watching (CW) lazy data structures as presented in [9] to detect, respectively, unit and monotone literals. The description of CW and 3LW in [9] considers only the case where $\Pi = \emptyset$. In the following we give a particular emphasis to the extensions that we have developed on top of the original algorithms in order to handle the cases where $\Pi \neq \emptyset$.

In QUBE++ 3LW is organized as follows. For each constraint, QUBE++ has to watch three literals w_1, w_2 and w_3: if the constraint is a clause, then w_1, w_2 are existential and w_3 is universal; otherwise, w_1, w_2 are universal and w_3 is existential. Dummy values are used to handle the cases when a clause (resp. a term) does not contain at least two existential (resp. universal) literals and one universal (resp. existential) literal. 3LW for clauses works in the same way as described in [9], while for terms it works as follows. Each time a literal l is assigned, the terms where l is watched are examined. For each such term:

- If l is universal then, assuming $l = w_1$:
  - if w_2 or w_3 eliminate the term then stop;
  - if w_2 is open, then check the universal literals to see if there exists $l_\forall$ such that $l_\forall \neq w_2$ and $l_\forall$ is either open, or it eliminates the term; if so, let $w_1 \leftarrow l_\forall$ and stop, otherwise check the existential literals to see if there exists $l_\exists$ such that either $l_\exists$ eliminates the term or $l_\exists$ is before w_2 in the prefix; if so, let $w_3 \leftarrow l_\exists$ and stop, otherwise a unit literal (w_2) is found;

  - finally, if w_2 is assigned (i.e., w_2 simplified the term) then check the existential literals to see if there exists $l_\exists$ such that $l_\exists$ eliminates the term; if so, let $w_3 \leftarrow l_\exists$ and stop, otherwise an empty term is found.
- If l is existential then, if both w_1 and w_2 are open, or if w_1 or w_2 are eliminating the term, then stop; if either w_1 or w_2 is open (say it is w_2) then check the existential literals to see if there exists $l_\exists$ such that either $l_\exists$ eliminates the term or $l_\exists$ is before w_2 in the prefix; if so, let $w_3 \leftarrow l_\exists$ and stop, otherwise a unit literal (w_2) is found.

By keeping separate account of existential and universal literals in the constraints, 3LW always performs less operations than the other lazy data structures described in [9]. The 3LW algorithm is sound and complete in the sense that it correctly identifies unit and empty constraints, and that it detects all such constraints when they arise at a given node of the search tree.

The implementation of CW in QUBE++ is obtained by associating to each literal a single watched constraint. When the watched constraint is eliminated during the search, a new constraint to be watched is sought by scanning a list of constraints that QUBE++ maintains for each literal. If all the constraints where l occurs are eliminated, then $\bar{l}$ is pure and it can be propagated accordingly. The CW algorithm is sound and complete in the sense that it correctly identifies monotone literals, and that it detects all such literals when they arise at a given node of the search tree. The use of CW (and 3LW) speeds up the lookahead process by examining fewer constraints, and the search process as a whole, by avoiding the bookkeeping work needed by non-lazy data structures when assignments are retracted during backtracking.

Lazy data structures do not provide up-to-date information about the status of the formula, e.g, how many constraints have been eliminated, or how many binary, ternary, etc. constraints are left. Therefore, they have an impact on the implementation of QUBE++ and, in particular, on the termination condition (when are all the clauses in Σ eliminated?) and on the search heuristic (how to score literals?). The first issue is solved having QUBE++ try to assign all the literals: if no empty constraint is detected beforehand, then a solution is found. As for the heuristic, the issue is more complicated and we have dedicated a separate section to it. Despite these apparent limitations, we have run experiments using the real-world instances from the QBF evaluation that confirm the effectiveness of lazy data structures vs. a non-lazy counterpart. We compared QUBE++ vs. an early version of the system using a non-lazy data structure; both versions featured chronological backtracking so that no advantage for fast exploration of large constraints sets is expected for lazy data structures. Moreover, the version using non-lazy data structures keeps track of eliminated clauses, and therefore identifies solutions as soon as they arise. Even in this unfavorable setting, lazy data structures are, on average, 25% faster then their non-lazy counterpart. Considering the ratio of literal assignments vs. CPU time, lazy data structures perform, on average, two times more assignments per second than a non-lazy data structure.

5 Learning

Only a minority of state-of-the-art QBF solvers uses standard chronological backtracking (CB) as lookback algorithm (see [8]). This is not by chance, since CB 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 choice done way up in the search tree. To solve this problem [13] introduced conflict backjumping and solution backjumping (CBJ, SBJ) for QBFs. Using CBJ (resp. SBJ) the lookback procedure jumps over the choices that do not belong to the reason of the conflict (resp. solution) that triggered backtracking. Intuitively, given the QBF instance $\langle Q, \Sigma, \Pi\rangle$, if S is a conflict (resp. a solution), then a reason R is a subset of S such that $\langle Q, \Sigma \cup \{l : \bar{l} \in R\}, \Pi\rangle$ (resp. $\langle Q, \Sigma, \Pi \cup \{R\}\rangle$) is logically equivalent to $\langle Q, \Sigma, \Pi\rangle$.

Reasons are initialized when a conflict or a solution is detected, and they are updated while backtracking. For details regarding this process, see [10]. With CBJ/SBJ reasons are discarded while backtracking over the nodes that caused the conflict or the solution, and this may lead to a subsequent redundant exploration. With learning as introduced for QBFs by [10], the reasons computed may be stored as constraints to avoid repeating the same search. In particular, although learning did not emerge from the QBF evaluation as an all-time winner [8], its effectiveness on real-world test cases is a consolidated result in the SAT literature (see, e.g., [11, 14, 12]), and positive results have been reported also for QBF reasoning (see, e.g., [10, 15]).

The fundamental problem with learning is that unconstrained storage of clauses (resp. terms) obtained by the reasons of conflicts (resp. solutions) may lead to an exponential memory blow up. In practice, it

is necessary to introduce criteria (i) for limiting the constraints that have to be learned, and/or (ii) for unlearning some of them. The implementation of learning in QUBE++ works as follows. Assume that we are backtracking on a literal l assigned at decision level n, where the *decision level* of a literal is the number of AND-OR nodes before l. The constraint corresponding to the reason for the current conflict (resp. solution) is learned only if:

- l is existential (resp. universal),
- all the assigned literals in the reason except l, are at a decision level strictly smaller than n, and
- there are no open universal (resp. existential) literals in the reason that are before l in the prefix.

Notice that these three conditions ensure that l is unit in the constraint corresponding to the reason. Once QUBE++ has learned the constraint, it backjumps to the node at the maximum decision level among the literals in the reason, excluding l. We say that l is a *Unique Implication Point* (UIP) and therefore the lookback in QUBE++ is *UIP-based*. Notice that our definition of UIP generalizes to QBF the concepts first described by [11] and used in the SAT solver GRASP. On a SAT instance, QUBE++ lookback scheme behaves similarly to the "1-UIP-learning" scheme used in ZCHAFF and described in [16]. Even if QUBE++ is guaranteed to learn at most one clause (resp. term) per each conflict (resp. solution), still the number of the learned constraints may blow up, as the number of backtracks can be exponential. To stop this course, QUBE++ scans periodically the set of learned constraints in search of those that became *irrelevant*, i.e., clauses (resp. terms) where the number of open literals exceeds a given parameter r. The method, called *relevance bounded learning* and introduced for SAT solvers by [14], ensures that the number of learned clauses and terms is $O(m^r)$, where m is the number of distinct atoms in the input formula. In other words, using relevance bounded learning ensures that the number of learned clauses (resp. terms) is polynomial in the number of distinct existential (resp. universal) atoms in the input formula.

6 Branching strategy

The report by [8] lists the development of an effective heuristic for QBF solvers among the challenges for future research. For our purposes, an heuristic is effective when it performs consistently better, on average, than a simple random heuristic. To understand the nature of such a challenge, let us introduce QBFs in the form

$$\exists X_1 \forall X_2 \exists X_3 ... \forall X_{n-1} \exists X_n \Phi \tag{2}$$

where each $X_i = x_{i1}, \ldots x_{im_i}$, and $Q_i X_i$ stands for $Q_i x_{i1} \ldots Q_i x_{im_i}$. Running an heuristic on (2) amounts to choosing an open literal among the m_i atoms available at the *prefix level* i, with the proviso that atoms at prefix level i must be assigned before we can choose atoms from prefix level $i+1$. Varying n and each of the m_i's, we can range from formulas like

$$\exists x_1 \forall x_2 \exists x_3 ... \forall x_{n-1} \exists x_n \Phi \tag{3}$$

where $m_i = 1$ for every i, to formulas like

$$\exists x_1 \exists x_2 \ldots \exists x_m \Phi \tag{4}$$

where $n = 1$, i.e., (4) is a SAT instance. If we consider QBFs of the sort (3) then it is likely that the heuristic is almost useless: unless an atom $|l|$ is removed from the prefix because l is either unit or monotone, the atom to pick at each node is fixed. On the other hand, considering QBFs of the sort (4), we know from SAT literature that nontrivial heuristics are essential to reduce the search space. In practice, QBF instances lay between one of the extremes marked by (3) and (4), but instances like (3) are fairly uncommon, particularly in real-world problems. For this reason, it does make sense to try to devise an heuristic for QUBE++, but to make it effective, we must also minimize its overhead. This task is complicated further by the fact that QUBE++ uses lazy data structures, and therefore the heuristic cannot efficiently extract complete and up-to-date information about the formula. In order to accomplish this, we designed CHOOSE-LITERAL in QUBE++ to use the information gleaned from the input formula at the beginning of the search, and then to exploit the information gathered during the learning process. This can be done with a minimum overhead, and yet enable QUBE++ to make informed decisions at each node. The heuristic is implemented as follows. To each literal we associate two counters, initially set to 0: the number of clauses c such that $l \in c$, and the number of terms t such that $l \in t$. Each time a constraint is added, either because it is an

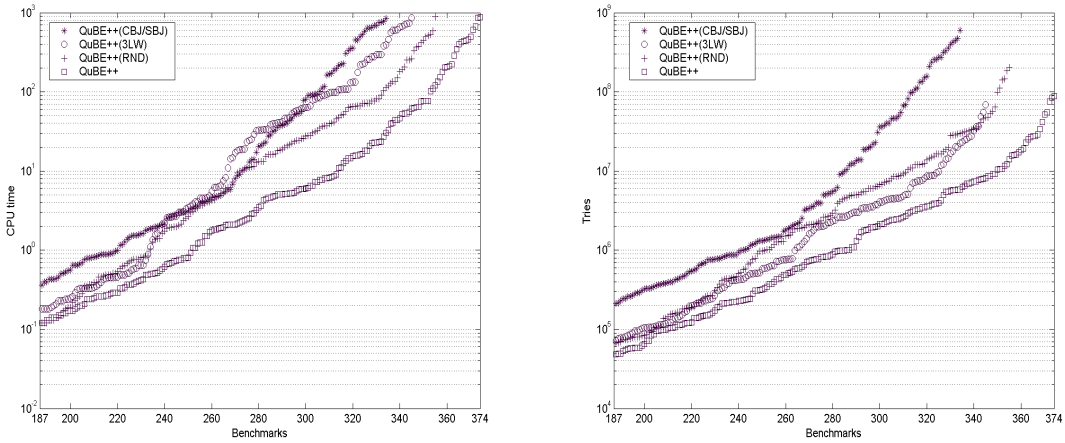

Fig. 2. Relative effectiveness of the techniques implemented in QuBE++.

input clause or a learned constraint, the counters are incremented; when a learned constraint is removed, the counters are decremented. This generates a tiny overhead since constraints are examined anyway during the learning/unlearning process. In order to choose a suitable branching literal, we arrange literals in a priority queue according to (i) the prefix level of the corresponding atom, (ii) the score and (iii) the numeric ID. In this way, atoms at prefix level i are always before atoms at prefix levels $j > i$, no matter the score; among atoms that have the same prefix level, the open literal with the highest score comes first; ties are broken preferring low numeric IDs. Choosing a branching literal is thus inexpensive, since it amounts to picking the first literal in the priority queue. Periodically, we rearrange the priority queue by updating the score of each literal l: this is done by halving the old score and summing to it the variation in the number of constraints k such that $l \in k$, if l is existential, or the variation in the number of constraints k such that $\bar{l} \in k$, if l is universal. The variations are measured with respect to the last update. Rearranging the priority queue is an expensive operation, and therefore QuBE++ does it only at multiples of a fixed threshold in the number of nodes. In this way, the overhead of the update is amortized over as many nodes as the threshold value. Clearly, an higher threshold implies less overhead per node.

7 Experimental results

To test the effectiveness of the techniques implemented in QuBE++ we used the set of 450 formal verification and planning instances that constituted part of the QBF evaluation[1]: 25% of these instances are from formal verification problems [3, 4], and the remaining are from planning domains [5, 6]. 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`.

In Figure 2 we compare the runtime and tries distribution of QuBE++ with QuBE++(3LW), QuBE++(CBJ/SBJ) and QuBE++(RND), which are obtained from QuBE++ by switching off, respectively, MLF, UIP-based learning and the branching strategy. Notice that we use tries as a CPU independent performance measure, instead of branches. The number of tries is the number of times that a literal is assigned a value, for whatever reason, be it a choice of the heuristic, a unit literal, or a monotone literal. The number of branches is always less than (or equal to) the number of tries, and tries are more informative than branches as a CPU independent performance measure, because most of the overall run time of the solver is spent on assigning literals (as reported, e.g., by [17]). Ordering the results of each solver independently and in ascending order yields the two plots of Figure 2: the x-axis is an ordinal in the range (187-374) since we left out (i) the problems that could not be solved within 900 seconds, and (ii) the values smaller than the median of the distributions; the y-axis of Fig. 2 (left) is CPU seconds, while the y-axis of Fig. 2 (right) is the number of tries, i.e., the number of literals assigned by QuBE++. By looking at Figure 2 we can see that all the techniques contribute to the effectiveness of QuBE++. The most effective one, despite its substantial overhead, is UIP-based learning: QuBE++ solves 40 instances more than QuBE++(CBJ/SBJ), and it is up to

[1] Our test set is thus comprised of non-random instances without the QBF encodings of the modal K formulas submitted by Guoqiang Pan [7].

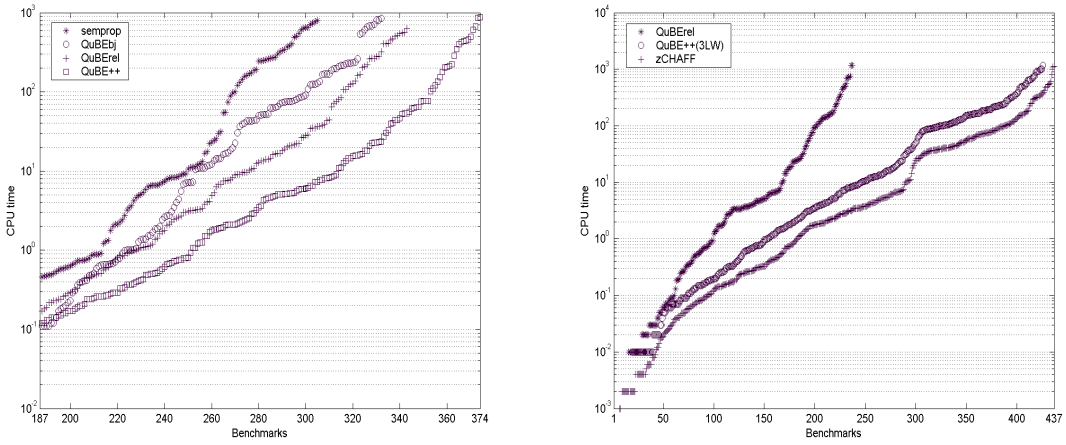

Fig. 3. QUBE++ vs. state-of-the-art QBF and SAT solvers.

more than two orders of magnitude faster. MLF comes second: QUBE++ solves 29 instances more than QUBE++(3LW) and it is up to more than one order of magnitude faster. Notice that, despite the overhead of MLF, the gap between QUBE++ and QUBE++(3LW) in terms of CPU time is wider than the corresponding gap in terms of tries. This phenomenon is explained by the fact that switching off MLF causes monotone literals to be propagated as branching literals, thus artificially inflating the number of nodes and consequently the number of updates that the heuristic performs on its priority queue. Finally, the branching strategy also helps in reducing the search space: QUBE++ solves 19 instances more than QUBE++(RND), and it is up to one order of magnitude faster. If we restrict our attention to the 306 instances solved by all the variants of QUBE++, it turns out that QUBE++ is faster than QUBE++(RND) and QUBE++(CBJ/SBJ) on 243 of them (about 80%), and it is faster than QUBE++(3LW) on 213 of them (about 70%); in all the problems where QUBE++ is slower than one of its variants its runtime is in any case less than 1 minute.

In Figure 3 (left) we show the results of a comparison between QUBE++ and the state-of-the-art QBF solvers that were reported as best on non-random instances by [8]. In particular, we selected the three top performers on these kind of instances: SEMPROP [18], QUBEBJ [13], and QUBEREL [10]. By looking at Figure 3 (left) it is evident that QUBE++ advances the state of the art in the solution of real-world QBFs. Within 900 seconds, QUBE++ solves 31, 43, and 69 more instances than QUBEREL, QUBEBJ, and SEMPROP, respectively. Among these, QUBE++ is the only one able to solve 13 instances, of which 6 defied all the participants in the QBF evaluation [8]. Overall, QUBE++ is up to one, two, and nearly three orders of magnitude faster than QUBEREL, QUBEBJ, and SEMPROP, respectively. A different view of these results is obtained comparing QUBE++ and the SOTA solver, i.e., the best time of SEMPROP, QUBEBJ and QUBEREL run in parallel on three different processors. Focusing on a subset of 304 nontrivial benchmarks, i.e., those where the run time of either QUBE++ or the SOTA solver exceeds 10^{-1} seconds, we see that, on 197 instances, QUBE++ is as fast as, or faster than, the SOTA solver: the median CPU times ratio indicates that QUBE++ is at least 3.7 times faster than the SOTA solver on half of these instances, while on 23 of them the gap is more than one order of magnitude. On the remaining 107 instances, QUBE++ is slower than the SOTA solver: on 8 of these instances, QUBE++ run time exceeds 900 seconds, while SEMPROP, QUBEBJ and QUBEREL manage to solve 4, 6, and 1 instances, respectively; on the remaining 99 instances, the median CPU time ratio indicates that QUBE++ is at least a factor of 5 slower than the SOTA solver on half of them, but the same value calculated separately for QUBEBJ, QUBEREL and SEMPROP, indicates that QUBE++ is medianly as fast as QUBEREL and SEMPROP and about a factor of 2.5 slower than QUBEBJ. In Figure 3 (right) we also checked the standing of QUBE++ with respect to the state of the art in SAT using a set of 483 challenging real-world SAT instances. We compared QUBE++(3LW), QUBEREL, the best state-of-the-art solver on real-world QBFs according to our experiments, and ZCHAFF, the winner of SAT 2002 competition [1] and one of the best solvers in SAT 2003 competition [2] on real-world SAT instances. We have used QUBE++(3LW) instead of QUBE++ since it is well known that MLF is not helpful in SAT and, in our experiments, MLF degraded the performances of QUBE++ on most SAT instances. The results show that QUBE++(3LW) is, medianly, only a factor of two slower than ZCHAFF, while QUBEREL is, medianly, one order of magnitude slower than ZCHAFF. Overall, both QUBE++ and ZCHAFF conquer about 90% of the instances within 1200 seconds, while QUBEREL can solve only 50% of them. Considering

that QUBE++ has to pay the overhead associated to being able to deal with QBFs instead of SAT instances only, it is fair to say that QUBE++ is effectively bridging the gap between SAT and QBF solvers, and that this is mainly due to the techniques that we proposed and their combination.

8 Conclusions

In this paper we have described three reasoning techniques that combined into our solver QUBE++ contributed to make it particularly effective on real world QBF instances coming from formal verification and planning domains. According to our experiments, efficient monotone literal fixing, UIP-based learning and adaptive branching strategies boost performances on their own and, combined together, enable QUBE++ to obtain order-of-magnitude improvements with respect to SEMPROP, QUBEBJ and QUBEREL which ranked as the best solvers on real-world instances in the last QBF evaluation. QUBE++ effectiveness is further confirmed by the fact that its performances are competitive with respect to ZCHAFF on real-world SAT instances.

References

1. L. Simon, D. Le Berre, and E. A. Hirsch. The SAT2002 Competition, 2002.
2. L. Simon and D. Le Berre. The essentials of SAT 2003 Competition. In *Sixth International Conference on Theory and Applications of Satisfiability Testing (SAT 2003)*, volume 2919 of *Lecture Notes in Computer Science*. Springer Verlag, 2003.
3. C. Scholl and B. Becker. Checking equivalence for partial implementations. In *38th Design Automation Conference (DAC'01)*, 2001.
4. 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.
5. J. Rintanen. Constructing conditional plans by a theorem prover. *Journal of Artificial Intelligence Research*, 10:323–352, 1999.
6. C. Castellini, E. Giunchiglia, and A. Tacchella. Sat-based planning in complex domains: Concurrency, constraints and nondeterminism. *Artificial Intelligence*, 147(1):85–117, 2003.
7. Guoqiang Pan and Moshe Y. Vardi. Optimizing a BDD-based modal solver. In *Proceedings of the 19th International Conference on Automated Deduction*, 2003.
8. 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.
9. I. Gent, E. Giunchiglia, M. Narizzano, A. Rowley, and A. Tachella. Watched data structures for QBF solvers. In *Sixth International Conference on Theory and Applications of Satisfiability Testing (SAT'03)*, pages 348–355, 2003. Extended Abstract.
10. E. Giunchiglia, M. Narizzano, and A. Tacchella. Learning for quantified boolean logic satisfiability. In *Eighteenth National Conference on Artificial Intelligence (AAAI'02)*. AAAI Press/MIT Press, 2002.
11. 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.
12. 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.
13. 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.
14. R. J. Bayardo, Jr. and R. C. Schrag. Using CSP Look-Back Techniques to Solve Real-World SAT instances. In *Proc. of AAAI*, pages 203–208. AAAI Press, 1997.
15. 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.
16. L. Zhang, C. F. Madigan, M. W. Moskewicz, and S. Malik. Efficient conflict driven learning in a Boolean satisfiability solver. In *International Conference on Computer-Aided Design (ICCAD'01)*, pages 279–285, 2001.
17. 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.
18. 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.