QuBE++: an Efficient QBF Solver

Enrico Giunchiglia, Massimo Narizzano, and Armando Tacchella *

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

Abstract. In this paper we describe QuBE++, an efficient solver for Quantified Boolean Formulas (QBFs). To the extent of our knowledge, QuBE++ is the first QBF reasoning engine that uses lazy data structures both for unit clauses propagation and for pure literals detection. QuBE++ also features non-chronological backtracking and a branching heuristic that leverages the information gathered during the backtracking phase. Owing to such techniques and to a careful implementation, QuBE++ turns out to be an efficient and robust solver, whose performances exceed those of other state-of-the-art QBF engines, and are comparable with the best engines currently available on SAT instances.

1 Introduction

The implementation of efficient reasoning tools for deciding the satisfiability of Quantified Boolean Formulas (QBFs) is an important issue in several research fields, including Verification [1, 2], Planning (Synthesis) [3, 4], and Reasoning about Knowledge [5]. Focusing on computer-aided design, both formal property verification (FPV) and formal circuit equivalence verification (FCEV) represent demanding and, at the same time, very promising application areas for QBF-based techniques. For instance, in FPV based on bounded model checking (BMC) [6] counterexamples of lengths limited by a given bound are sought. BMC techniques showed to be excellent for bug finding (see, e.g., [7]), but, unless the bound corresponds to the diameter of the system, BMC tools cannot fully verify the system, i.e., certify it as bug-free. One possible solution, in the words of [6], is: "New techniques are needed to determine the diameter of the system. In particular it would be interesting to study efficient decision procedures for QBF". Still in the FPV arena, it is well known that symbolic model checking of safety properties amounts to solving a symbolic reachability problem (see, e.g., [8]): symbolic reachability can be translated efficiently to the satisfiability of a QBF, while corresponding SAT encodings do not have the same property. In the FCEV arena, a possible application of QBF reasoning is checking equivalence for partial implementations [1]: as in the case of diameter calculation and symbolic reachability, this application requires the expressive power of QBFs in order to be translated efficiently to an equivalent automated reasoning task.

In the recent years, QBF tools have known a fast and steady development. Witnessing the vitality of the field, the 2003 evaluation of QBF solvers [9] hosted eleven

* 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.

solvers that were run on 1720 benchmarks, about half of which coming from randomly generated instances, and the other half coming from encoding of problems into QBF (so called *real-world* instances). The encodings obtained considering Verification and Synthesis problems [1–4] represented about 50% of the total share of real-world instances. The evaluation showed that research in QBF reasoning reached a decent level of maturity, but also that there is still a lot of room for improvement. For instance, although QBF solvers are close relatives of the SAT solvers routinely used in FPV and FCEV applications, they are still lagging behind SAT solvers in terms of efficiency and sophistication.

In this paper we describe QUBE++, a QBF reasoning engine which using lazy data structures both for unit clauses propagation and for pure literals detection. QUBE++ also features a specifically tailored non-chronological backtracking method and a branching heuristic that leverages the information gathered during the backtracking phase. To the extent of our knowledge, QUBE++ is the first solver that combines all the above features in a single tool. Like most current state-of-the-art QBF solvers, QUBE++ is based on the Davis, Putnam, Logemann, Loveland procedure (DPLL) [10, 11]. As such, QUBE++ explores an implicit AND-OR search tree alternating three phases: simplification of the formula (lookahead), choice of a branching literal when no further simplification is possible (heuristic), and backtracking when a contradiction or a satisfying assignment is found (lookback). Each of these phases has been optimized in QUBE++. As for lookahead, we implemented an extension of the lazy data structures described in [12] which enable an efficient detection of unit and monotone literals. Lookback is based on learning as introduced in [13], with improvements generalizing those first used in SAT by GRASP [14]. The heuristic is designed to exploit information gleaned from the input formula initially, and then to leverage the information extracted during lookback. It is worth pointing out that the innovation in QUBE++ comes from the above techniques, as well as from the effort of combining them. Despite their apparent orthogonality, the specific algorithms that we have conceived (e.g., lookback-based heuristic), the nature of the problems that we have faced (e.g., pure literal detection in presence of learned clauses), and the quest for efficiency, posed nontrivial issues that turned the engineering of QUBE++ into a challenging research task.

Since QUBE++ has been designed to perform at its best on real-world QBF instances and to bridge the gap with SAT solvers, we have compared it with other state-of-the-art QBF solvers using verification and synthesis benchmarks from the 2003 QBF evaluation [9], and with the solver ZCHAFF [15] using the test set of challenging real-world SAT benchmarks presented in [16]. Owing to the techniques that we describe and to their careful implementation, QUBE++ turns out to be faster than QUBEREL, QUBEBJ and SEMPROP, i.e., the best solvers on non-random instances according to [9], and also faster and more robust than YQUAFFLE, a new and re-engineered version of QUAFFLE [17]. On the SAT benchmarks, QUBE++ bridges the gap with SAT solvers by conquering about 90% of the instances and losing, in a median-to-median comparison of the run times, only a factor of two from ZCHAFF, while QUBEREL, the best among the QBF solvers that we tried on real-world QBF instances, conquers only 50% of the problems and it is, in a median-to-median comparison of the run times, one order of magnitude slower than ZCHAFF.

The paper is structured as follows. We first review the logic of QBFs. Then we present the basic algorithm of QUBE++ and its three main features: optimized lookahead, UIP-based lookback, and lookback-based heuristic. We comment the results of the experiments outlining the efficiency of QUBE++, and then we conclude with related work and some final remarks.

2 Basics

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 *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*, 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 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 $\overline{l}$ 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 instances of the kind $Q_1 x_1 \ldots Q_n x_n \Phi$, where $\Phi = \Psi \vee \Theta$, Ψ is a conjunction of clauses, and Θ is a disjunction of terms: initially Ψ contains the matrix of the input QBF and Θ is $\bot$. Under these assumptions, clauses (resp. terms) are added to Ψ (resp. Θ) during the learning process as shown in [13]. 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 $Q_1 x_1, \ldots, Q_n x_n$, 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 $\Pi = \emptyset$ and $S = \emptyset$). In the following, as customary in search algorithms, we deal with constraints as if they were *sets* of literals. SOLVE works in four steps (line numbers refer to Figure 1):

bool SOLVE(Q, Σ, Π, S)
1 **do**
2 $\quad\langle Q', \Sigma', \Pi', S'\rangle \leftarrow \langle Q, \Sigma, \Pi, S\rangle$
3 $\quad\langle Q, \Sigma, \Pi, S\rangle \leftarrow$ SIMPLIFY(Q', Σ', Π', S')
4 **while** $\langle Q, \Sigma, \Pi, S\rangle \neq \langle Q', \Sigma', \Pi', S'\rangle$
5 **if** $\Sigma = \emptyset$ **or** $\emptyset_\forall \in \Pi$ **then return** TRUE
6 **if** $\emptyset_\exists \in \Sigma$ **and** $\Pi = \emptyset$ **then return** FALSE
7 $l \leftarrow$ CHOOSE-LITERAL(Q, Σ, Π)
8 **if** l is existential **then**
9 **return** SOLVE(Q, $\Sigma \cup \{l\}$, Π, S) **or** SOLVE(Q, $\Sigma \cup \{\bar{l}\}$, Π, S)
10 **else**
11 **return** SOLVE(Q, Σ, $\Pi \cup \{\bar{l}\}$, S) **and** SOLVE(Q, Σ, $\Pi \cup \{l\}$, S)

set SIMPLIFY(Q, Σ, Π, S)
12 **while** $\{l\}_\exists \in \Sigma$ **or** $\{\bar{l}\}_\forall \in \Pi$ **do**
13 $\quad S \leftarrow S \cup \{l\}$
14 $\quad Q \leftarrow$ REMOVE(Q, $|l|$)
15 $\quad$**for** each $c \in \Sigma$ s.t. $l \in c$ **do**
16 $\quad\quad \Sigma \leftarrow \Sigma \setminus \{c\}$
17 $\quad$**for** each $t \in \Pi$ s.t. $\bar{l} \in t$ **do**
18 $\quad\quad \Pi \leftarrow \Pi \setminus \{t\}$
19 $\quad$**for** each $c \in \Sigma$ s.t. $\bar{l} \in c$ **do**
20 $\quad\quad \Sigma \leftarrow (\Sigma \setminus \{c\}) \cup \{c \setminus \{\bar{l}\}\}$
21 $\quad$**for** each $t \in \Pi$ s.t. $l \in t$ **do**
22 $\quad\quad \Pi \leftarrow (\Pi \setminus \{t\}) \cup \{t \setminus \{l\}\}$
23 **for** each l s.t. $\bar{l} \notin k$ for all $k \in (\Sigma \cup \Pi)$ **do**
24 $\quad$**if** l is existential **then** $\Sigma \leftarrow \Sigma \cup \{l\}$
25 $\quad$**else** $\Pi \leftarrow \Pi \cup \{l\}$
26 **return** $\langle Q, \Sigma, \Pi, S\rangle$

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

1. Simplify the input instance with SIMPLIFY (lines 1-4): SIMPLIFY is iterated until no further simplification is possible.
2. Check if the *termination condition* is met, i.e., if we are done with the current search path and backtracking is needed (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 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' such that (i) $|l'|$ is in Q and (ii) $|l'|$ is quantified differently from $|l|$ and it occurs 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 exists 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 SIMPLIFY 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 SIMPLIFY has the task of finding and assigning all unit and monotone literals at every node of the search tree. SIMPLIFY loops while 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, SIMPLIFY 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 algorithm

As reported by [15] 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 [13], 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.

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 presented in [12] to detect, respectively, unit and monotone literals. The description of CW and 3LW in [12] considers only the case where $\Pi = \emptyset$, but it is sufficient to understand CW implementation in QUBE++. As for 3LW, in QUBE++ it 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 [12], 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 [12]. 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 same can be stated for CW (as described in [12]) and monotone literals. The use of 3LW and CW 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 and all the literals have been assigned, 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 efficiency 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 UIP-based lookback algorithm

Only a minority of state-of-the-art QBF solvers uses standard chronological backtracking (CB) as lookback algorithm (see [9]). 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 [18] 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 a 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 [13]. 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 [13], the reasons computed may be stored as constraints to avoid repeating the same search.

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 nodes before l. The constraint corresponding to the reason for the current conflict (resp. solution) is learned only if:

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

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 [14] 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 [19]. 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 is not guaranteed to be polynomial. 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 [20], 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.

6 Lookback-based heuristic

The report by [9] lists the development of an effective heuristic for QBF solvers among the challenges for future research. To understand the nature of such a challenge, consider QBFs having 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 a heuristic on (2) amounts to choosing an open literal among a set X_i, with the proviso that all the atoms in the sets X_j with $j < i$ must be assigned before we can choose atoms from the set X_i. 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 the SAT literature that nontrivial heuristics are essential to reduce the search space. In practice, QBF instances lay between the extremes marked by (3) and (4), and instances like (3) are fairly uncommon, particularly in real-world problems. For this reason, it does make sense to try to devise a heuristic for QUBE++, but to make it efficient, 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.

Given all these considerations, 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 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 anyways 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, a higher threshold implies less overhead per node, but also a less accurate choice of the branching literal.

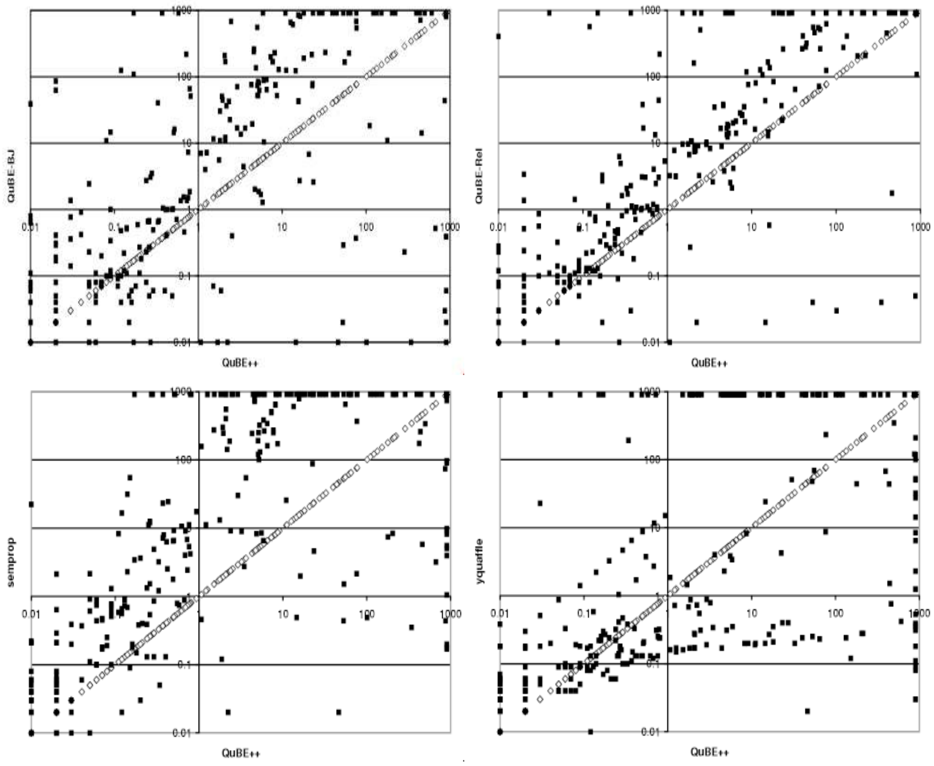

Fig. 2. Correlation plots of QUBE++ vs. state-of-the-art QBF solvers.

The intuition behind the heuristic is to favor existential literals which eliminate a lot of clauses and simplify a lot of terms, thereby incrementing the chances of finding a solution; conversely, the heuristic favors universal literals which simplify a lot of clauses and eliminate a lot of terms, thereby incrementing the chances of finding a conflict. The fact that the scores are periodically slashed helps the solver to keep focus on more recent learned constraints (a similar technique is employed by ZCHAFF [15] for SAT instances).

7 Experimental results

In order to validate QUBE++ and tune its performances we have run several experiments using real-world instances extracted from those available at QBFLIB [21]. Here 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 [9]. In particular, we selected the three top performers on this kind of instances: SEMPROP [22], QUBEBJ [18], and QUBEREL [13]. Since at the time of the evaluation the performances of QUAFFLE could not be fairly assessed because of a parser problem, we decided to include also its new version YQUAFFLE in our analysis. For all the solvers considered, we report the results of their most recent versions available at the time this paper was submitted for review (April 2004). To run the comparison, we have used the same set of 450 verification and planning (synthesis) benchmarks that constituted part of the QBF evaluation: 25% of these instances are from verification problems [1, 2], and the remaining are from planning domains [3, 4]. All the experiments were run on a farm of PCs, each one equipped with a Pentium IV 2.4GHz processor, 1GB of RAM, and running Linux RedHat 7.2.

In Figure 2 we compare the performances of QUBE++ with the other state-of-the-art QBF solvers. In the plots of Figure 2, each solid-fill square dot represents an instance, QUBE++ solving time is on the x-axis (log scale), QUBEBJ (top-left), QUBEREL (top-right), SEMPROP (bottom-left) and YQUAFFLE (bottom-right) solving times are on the y-axes (log scale). The diagonal (outlined diamond boxes) represents the solving time of QUBE++ against itself and serves the purpose of reference: the dots above the diagonal are instances where QUBE++ performs better than its competitors, while the dots below are the instances where QUBE++ performances are worse than the other solvers. By looking at Figure 2 we can see that QUBE++ performances compare favorably with its competitors. Considering the number of instances solved both by QUBE++ and each of its competitors, on 268/342, 301/342, 259/308 and 198/301 instances QUBE++ is as fast as, or faster than QUBEBJ, QUBEREL, SEMPROP and YQUAFFLE respectively. A more detailed analysis reveals that QUBE++ is at least one order of magnitude faster than QUBEBJ (resp. SEMPROP) on 112 (resp. 163) problems, and it is at least one order of magnitude slower on 36 (resp. 37) problems, i.e., about 44% (resp. 60%) of the instances where QUBEBJ (resp. SEMPROP) is faster than QUBE++. QUBE++ is also at least one order of magnitude faster than QUBEREL on 79 instances, and one order of magnitude slower on 15 instances only. In the case of YQUAFFLE the plot thickens, since QUBE++ and YQUAFFLE results disagree on 54 instances. Noticeably, (i) for each one of these instances the satisfiability result of QUBE++ is independently confirmed by at least one solver among QUBEBJ, QUBEREL and SEMPROP, and (ii) some of the instances where YQUAFFLE reports a satisfiability (resp. unsatisfiability) result have been declared as unsatisfiable (resp. satisfiable) by the benchmark author. However, since a certificate of satisfiability (or unsatisfiability) in QBF is not as easy to produce and to check as it is in SAT, none of the solvers that we have tried produces a witness that can be used to check its result. Therefore we must rely on a majority argument to declare YQUAFFLE "wrong", and to consider its 54 mismatching answers as if the solver failed to complete within the time limit. With this proviso, on the remaining 396 instances (of which 360 solved both by QUBE++ and YQUAFFLE) QUBE++ is strictly faster than YQUAFFLE on 212 instances, and on 115 of them it is at least one order of magnitude faster. On the other hand, YQUAFFLE is at least one order of magnitude faster than QUBE++ on 56 instances only. In conclusion, it is fair to say that QUBE++ is more robust and also faster than YQUAFFLE, since the above extrapolated data correspond to the best-case scenario in which YQUAFFLE results agree with all the other solvers, and fixing it did not hurt its performances.

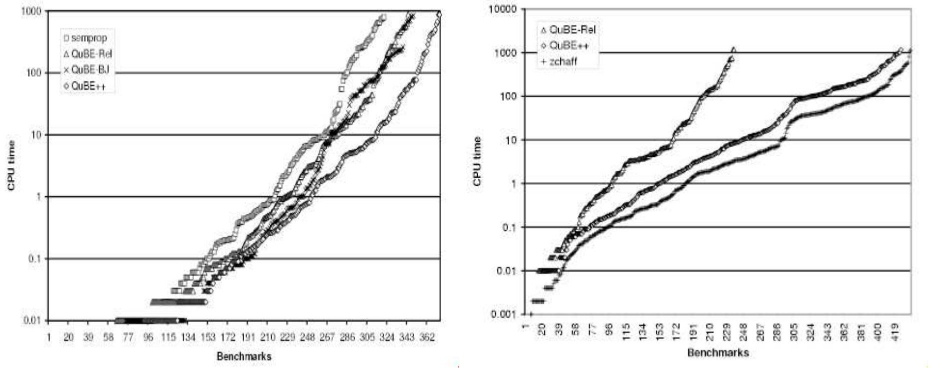

Fig. 3. Runtime distributions of QuBE++ vs. state-of-the-art QBF solvers (left) and on SAT instances (right).

In Figure 3 (left) we compare the runtime distribution of QuBERel, QuBEbj and semprop[1] with respect to QuBE++: the x-axis is an ordinal in the range (1-375), and the y-axis is CPU seconds on a logarithmic scale. The plot in Figure 3 is obtained by ordering the results of each solver independently and in ascending order. We show only the part of the plot where at least one of the solvers completed within the time limit of 900 seconds. By looking at Figure 3 (left) it is evident that QuBE++ advances considerably the state of the art in the solution of real-world QBF instances. Within the time limit, QuBE++ solves 22, 33, and 61 more instances than QuBERel, QuBEbj, and semprop, respectively. A quantitative analysis on the data shown in Figure 3 (left) is obtained by comparing QuBE++ with the *SOTA solver*, i.e., semprop, QuBEbj and QuBERel running in parallel on three different processors. Focusing on a subset of 273 nontrivial benchmarks, i.e., those where the run time of either QuBE++ or the SOTA solver exceeds 10^{-1} seconds, we see that:

- On 215 instances, QuBE++ is as fast as, or faster than, the SOTA solver and, in a median-to-median comparison of run times, QuBE++ is about five times faster; on 49 of these instances the gap is more than one order of magnitude.
- On 58 instances, QuBE++ is slower than the SOTA solver: on 18 of these instances, QuBE++ (and QuBERel) run time exceeds the time limit, while semprop and QuBEbj manage to solve 8 and 11 instances, respectively; on the remaining 40, QuBE++ is a factor of 7.5 slower than the SOTA solver, but, individually, it is only a factor of 1.9, 4.8 and 1.5 slower than semprop, QuBEbj and QuBERel, respectively (all the factors obtained comparing median run times).

On the remaining 177 trivial instances, all the solvers perform equally well. Summing up, QuBE++ performances are individually much better than semprop, QuBEbj and

[1] We have discarded yquaffle from this analysis because we cannot fully rely on its answers and, consequently, on its solving times.

QuBEREL, and are even better than the SOTA solver obtained by combining all the three of them.

As we said in the introduction, [9] reports that QBF solvers are still lagging behind SAT solvers in terms of efficiency and sophistication. In Figure 3 (right) we checked the standing of QuBE++ with respect to this issue using a set of 483 challenging real-world SAT instances (described in [16]). We compared QuBE++, QuBEREL, the best state-of-the-art solver on real-world QBFs according to our experiments, and ZCHAFF, the winner of SAT 2002 competition [23] and one of the best solvers in SAT 2003 competition [24] on real-world SAT instances. The results show that QuBE++ is only a factor of two slower than ZCHAFF, while QuBEREL is one order of magnitude slower than ZCHAFF (median-to-median comparison on the run times). 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 and related work

In this paper we have described QuBE++, an efficient QBF solver QBF instances. QuBE++ owes its efficiency to the propagation scheme based on lazy data structures tied with the UIP-based learning method and a lookback-based search heuristic. To the extent of our knowledge, QuBE++ is the first QBF solver to feature such a powerful combination of techniques. On real-world QBF instances, QuBE++ shows order-of-magnitude improvements with respect to SEMPROP, QuBEBJ, QuBEREL, and YQUAFFLE. On a test set of challenging real-world SAT benchmarks, QuBE++ bridged the gap with SAT solvers as it was comparable to ZCHAFF, while QuBEREL lagged severely behind both ZCHAFF and QuBE++.

Considering other state-of-the-art QBF solvers adopting some form of smart lookback scheme, we have SEMPROP [22] that features a different form of learning, non-lazy data structures and a different heuristic; QuBEBJ [18] and QuBEREL [13] that feature non-lazy data structures, a different heuristic and lookback methods on which those of QuBE++ are based, but with different implementations; QUAFFLE [17] restricted learning to conflicts, and featured a non-lazy data structure; as for YQUAFFLE, no detailed description of its relevant features is available to the extent of our knowledge; finally, WATCHEDCSBJ (see, e.g., [12] for a description of its implementation) features lazy data structures restricted to clauses, and CBJ/SBJ without learning. We did not report about QUAFFLE and WATCHEDCSBJ, since QUAFFLE has been replaced by YQUAFFLE, while WATCHEDCSBJ turned out to be slower than SEMPROP on real-world QBF instances according to our experiments.

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. J. Rintanen. Constructing conditional plans by a theorem prover. *Journal of Artificial Intelligence Research*, 10:323–352, 1999.
4. C. Castellini, E. Giunchiglia, and A. Tacchella. Sat-based planning in complex domains: Concurrency, constraints and nondeterminism. *Artificial Intelligence*, 147(1):85–117, 2003.
5. Guoqiang Pan and Moshe Y. Vardi. Optimizing a BDD-based modal solver. In *Proceedings of the 19th International Conference on Automated Deduction*, 2003.
6. A. Biere, A. Cimatti, E. Clarke, and Y. Zhu. Symbolic Model Checking without BDDs. In *Proceedings of TACAS*, volume 1579 of *LNCS*, pages 193–207. Springer-Verlag, 1999.
7. F. Copty, L. Fix, Ranan Fraer, E. Giunchiglia, G. Kamhi, A. Tacchella, and M. Y. Vardi. Benefits of Bounded Model Checking at an Industrial Setting. In *Proc. of CAV*, LNCS. Springer-Verlag, 2001.
8. P. A. Abdulla, P. Bjesse, and N. Eén. Symbolic Reachability Analisys Based on SAT-Solvers. In *Proceedings of TACAS*, volume 1785 of *LNCS*, pages 411–425. Springer-Verlag, 2000.
9. 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.
10. M. Davis and H. Putnam. A computing procedure for quantification theory. *Journal of the ACM*, 7(3):201–215, 1960.
11. M. Davis, G. Logemann, and D. Loveland. A machine program for theorem proving. *Communications of the ACM*, 5(7):394–397, 1962.
12. 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.
13. 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.
14. 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.
15. 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.
16. 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.
17. 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.
18. 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.
19. 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.
20. 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.

21. E. Giunchiglia, M. Narizzano, and A. Tacchella. Quantified Boolean Formulas satisfiability library (QBFLIB), 2001. `www.qbflib.org`.
22. 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.
23. L. Simon, D. Le Berre, and E. A. Hirsch. The SAT2002 Competition, 2002.
24. 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.