QuBE: A system for deciding Quantified Boolean Formulas Satisfiability ⋆

Enrico Giunchiglia, Massimo Narizzano, and Armando Tacchella

DIST, Università di Genova, Viale Causa 13, 16145 Genova – Italy

1 Introduction

Deciding the satisfiability of a Quantified Boolean Formula (QBF) is an important research issue in Artificial Intelligence. Many reasoning tasks involving planning [1], abduction, reasoning about knowledge, non monotonic reasoning [2], can be directly mapped into the problem of deciding the satisfiability of a QBF.

In this paper we present QuBE, a system for deciding QBFs satisfiability. We start our presentation in § 2 with some terminology and definitions necessary for the rest of the paper. In § 3 we present a high level description of QuBE's basic algorithm. QuBE's available options are described in § 4. We end our presentation in § 5 with some experimental results showing QuBE effectiveness in comparison with other systems. QuBE, and more information about QuBE, are available at `www.mrg.dist.unige.it/star/qube`.

2 Formal preliminaries

Consider a set P of propositional letters. An *atom* is an element of P. A *literal* is an atom or the negation of an atom. For each literal l, (i) $\bar{l}$ is x if $l = \neg x$, and is $\neg x$ if $l = x$; (ii) $|l|$ is the atom occurring in l. A *clause* C is an n-ary $(n \geq 0)$ disjunction of literals such that no atom occurs twice in C. A *propositional formula* is a k-ary $(k \geq 0)$ conjunction of clauses. As customary, we represent a clause as a set of literals, and a propositional formula as a set of clauses.

A *QBF* is an expression of the form

$$Q_1 x_1 \ldots Q_n x_n \Phi, \qquad (n \geq 0) \tag{1}$$

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

Consider a QBF of the form (1). A literal l occurring in Φ is:

- *existential* if $\exists |l|$ belongs to the prefix of (1), and is *universal* otherwise.
- *unit* in (1) if l is existential, and, for some $k \geq 0$,
  - a clause $\{l, l_1, \ldots, l_k\}$ belongs to Φ, and
  - each expression $\forall |l_i|$ $(1 \leq i \leq k)$ is at the right of $\exists |l|$ in the prefix of (1).

⋆ We wish to thank Marco Cadoli, Rainer Feldmann, Theodor Lettman, Jussi Rintanen, Marco Schaerf and Stefan Schamberger for providing us with their systems and helping us to figure them out during our experimental analisys. This work has been partially supported by ASI and MURST.

```
φ = ⟨the input QBF⟩;
Stack= ⟨the empty stack⟩;

function Simplify() {
  do {
    φ' = φ;
    if (⟨a contradictory clause is in φ⟩)
      return FALSE;
    if (⟨the matrix of φ is empty⟩)
      return TRUE;
    if (⟨l is unit in φ⟩)
      { |l|.mode = UNIT; Extend(l); }
    if (⟨l is monotone in φ⟩)
      { |l|.mode = PURE; Extend(l); }
  } while (φ' != φ);
  return UNDEF; }

function Backtrack(res) {
  while (⟨Stack is not empty⟩) {
    l = Retract();
    if ((|l|.mode == L-SPLIT) &&
        ((res == FALSE && |l|.type == ∃) ||
         (res == TRUE && |l|.type == ∀)))
      {|l|.mode = R-SPLIT; return l̄; }}
  return NULL; }

function QubeSolver() {
  do {
    res = Simplify();
    if (res == UNDEF) l = ChooseLiteral();
    else   l = Backtrack(res);
    if (l != NULL)   Extend(l);
  } while (l != NULL);
  return res; }
```

Fig. 1. The algorithm of QuBE.

- *monotone* if either l is existential, l occurs in Φ, and $\overline{l}$ does not occur in Φ; or l is universal, l does not occur in Φ, and $\overline{l}$ occurs in Φ.

A clause C is *contradictory* if no existential literal belongs to C.

The semantics of a QBF φ can be defined recursively as follows. If φ contains a contradictory clause then φ is not satisfiable. If the matrix of φ is empty then φ is satisfiable. 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 deleting the clauses in which l occurs, and removing $\overline{l}$ from the others. It is easy to see that if φ is a QBF without universal quantifiers, the problem of deciding φ satisfiability reduces to propositional satisfiability (SAT).

Notice that we allow only for propositional formulas in conjunctive normal form (CNF) as matrices of QBFs. Indeed, by applying standard CNF transformations (see, e.g., [3]) it is always possible to rewrite a QBF into an equisatisfiable one satisfying our restrictions.

3 QuBE algorithm

QuBE is implemented in C on top of SIM, an efficient SAT decider developed by our group. A C-like high-level description of QuBE is shown in Figure 1. In this Figure,

- φ is a global variable initially set to the input QBF.
- *Stack* is a global variable storing the search stack, and is initially empty.
- $\exists$, $\forall$, FALSE, TRUE, UNDEF, NULL, UNIT, PURE, L-SPLIT, R-SPLIT are pairwise distinct constants.
- for each atom x in Φ, (i) $x.mode$ is a variable whose possible values are UNIT, PURE, L-SPLIT, R-SPLIT, and (ii) $x.type$ is $\exists$ if x is existential, and $\forall$ otherwise.

- *Extend*(l) first pushes l and φ in the stack; then deletes the clauses of φ in which l occurs, and removes $\bar{l}$ from the others.
- *Retract*() pops the literal and corresponding QBF that are on top of the stack: the literal is returned, while the QBF is assigned to φ.
- *Simplify*() simplifies φ till a contradictory clause is generated (line 6), or the matrix of φ is empty (line 8), or no simplification is possible (lines 5, 14).
- *ChooseLiteral*() returns a literal l occurring in φ such that for each atom x occurring to the left of $|l|$ in the prefix of φ, x does not occur in φ, or x is existential iff l is existential. *ChooseLiteral*() also sets $|l|.mode$ to L-SPLIT.
- *Backtrack*(*res*): pops all the literals and corresponding QBFs (line 18) from the stack, till a literal l is reached such that $|l|.mode$ is L-SPLIT (line 19), and either (i) l is existential and *res* = FALSE (line 20); or (ii) l is universal and *res* = TRUE (line 21). If such a literal l exists, $|l|.mode$ is set to R-SPLIT, and $\bar{l}$ is returned (line 22). If no such literal exists, NULL is returned (line 23).

QUBE returns TRUE if the input QBF is satisfiable, and FALSE otherwise. It is easy to see that QUBE, like other QBF procedures (see, e.g., [4–6]), is a generalization of the Davis, Logemann, Loveland procedure (DLL) for SAT: QUBE and DLL have the same behavior on QBFs without universal quantifiers.

4 QUBE options

Consider Figure 1. QUBE ver. 1.0 features backjumping, trivial truth, six different branching heuristics, i.e., implementations of *ChooseLiteral*, and other control options.

The backjumping procedure implemented in QUBE [7] is a generalization of the conflict-direct backjumping procedure as implemented in SAT solvers. As far as we know, QUBE is the only QBF solver with backjumping. Because of the potential overhead, backjumping has to be enabled when compiling the system, while all the other heuristics and optimizations can be enabled/disabled using QUBE's command line.

QUBE's command line is:

```
qube [-tt] [-heuristics unit|bohm|jw2] [-length exists|all]
     [-verbose] [-timeout <n1>] [-memout <n2>] <file-name>.
```

By default, after the simplifications following the branch on an universal variable have been performed, QUBE checks whether the formula obtained from φ by deleting universal literals is satisfiable. If it is, then φ is satisfiable [4]. This optimization can produce dramatic speed-ups, particularly on randomly generated QBFs (see, e.g. [4]). The option `-tt` disables this check. Notice that ours is an optimized version of "trivial truth" as described in [4], where the check is performed at each branching node.

QUBE branching heuristics have been inspired by the SAT literature. Our current version includes `böhm`, `jw2` and `unit` heuristics. The behavior of these heuristics depends on the notion of "length" of a clause. QUBE features two definitions of length of a clause C: the number of literals in C (`-length all`) as in [4, 5], and the number of existential literals in C (`-length exists`) as in [6]. By combining these options, six different branching heuristics are possible.

The `böhm` and `jw2` heuristics are, respectively, a generalization of Böhm's heuristic [8] and "two-sided Jeroslow-Wang" heuristic [9] for SAT. The idea behind `böhm` and `jw2` is to choose literals that occur as often as possible in the shortest clauses of φ. The hope is that by assigning such literals, we will have the largest amount of simplification. The `unit` heuristic is based on the one implemented in SATZ [10]. As opposed to `böhm` and `jw2`, the `unit` heuristic tentatively assigns truth values to atoms in order to get the exact amount of simplification caused by such assignments.

Independently from the particular branching heuristic used, if the selected atom x is existential (resp. universal), QUBE tries first x if x has more (resp. less) positive than negative occurrences in the matrix of φ. The idea is to maximize the chances of showing φ satisfiability (resp. unsatisfiability) in the first branch.

The `-verbose` options enables printing search information during the execution, including the variable x being assigned, its mode (whether is a unit, a pure, $\ldots$), and –in case it is a L-SPLIT– whether x or $\neg x$ is tried first.

The `-timeout <n1>` and `-memout <n2>` options are used to limit the amount of resources used by QUBE. Whenever QUBE exceeds `<n1>` seconds of CPU time (resp. `<n2>` megs of RAM), its execution is halted.

5 QUBE performances

To evaluate QUBE performances, we compare it with DECIDE [5], EVALUATE [4], QKN [11], and QSOLVE [6]. According to our preliminary experimental results, the options `-heuristics bohm -length exists` give good performances on all the problems, and are thus the default. The tests run on a Pentium III, 600MHz, 128MBRAM.

We consider sets of randomly generated QBFs. We generate QBFs according to model A as described in [12]. In this model, each QBF has the following 4 properties: (i) the prefix consists of k sequences, each sequence has n quantifiers, and any two quantifiers in a sequence, are of the same type, (ii) the rightmost quantifier is $\exists$, (iii) the matrix consists of l clauses, (iv) each clause consists of h literals of which at least 2 are existential. The Figure 2 shows the median, out of 100 samples, of the CPU times when $k = 2$, $n = 100$ (left) and $k = 3$, $n = 100$ (right). We fixed $h = 5$ because it yields harder QBFs than $h < 5$ (see [12]), and l (on the x-axis) is varied in such a way to cover the "100% satisfiable – 100% unsatisfiable" transition (shown in the background). Notice the logarithmic scale on the y-axis.

Looking at Figure 2 (left) we immediatly see that QUBE and QSOLVE perform roughly the same (with QSOLVE being slightly better than QUBE) and better than all the other solvers that we tested. Still in Figure 2, for $k = 3$ we further observe that QUBE is always faster than QSOLVE, sometimes by orders of magnitude. Since QSOLVE runs trivial truth (and trivial falsity), but no backjumping, we take this as an evidence on the effectiveness of backjumping.

Our experimental analysis includes the 38 problems contributed by Rintanen in [5]. They are translations from planning problems into the language of QBFs and the best solver overall turns out to be DECIDE with 33 problems solved and 42.28s average running time (on solved samples), followed by QUBE, with 18 problems solved and 73.21s, and QSOLVE with 11 problems solved and 149.29s. QKN and EVALUATE with, respectively, 1 and 0 problems solved, trail the list. In this regard, we point out that DECIDE features "inversion of quantifiers" and "sampling" mechanisms which are particularly effective on these benchmarks .

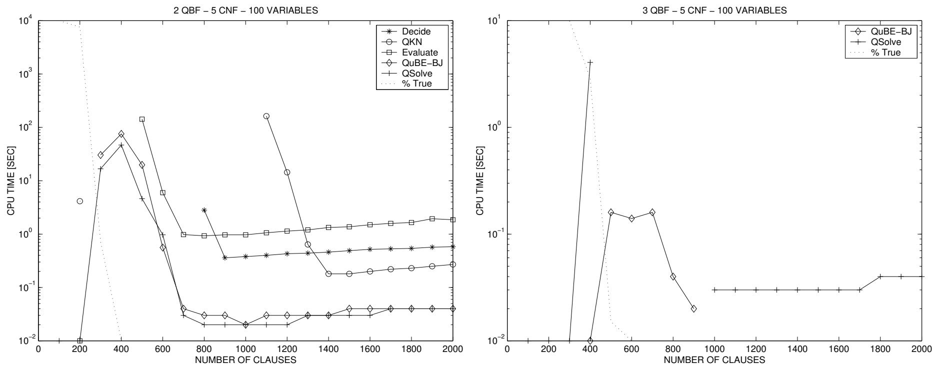

Fig. 2. CPU times, median, 100 samples/point. Background: satisfiability percentage. Left: $k = 2, n = 100$, Right: $k = 3, n = 100$,

References

1. Jussi Rintanen. Constructing conditional plans by a theorem prover. *Journal of Artificial Intelligence Research*, 10:323–352, 1999.
2. U. Egly, T. Eiter, H. Tompits, and S. Woltran. Solving advanced reasoning tasks using quantified boolean formulas. In *Proc. AAAI*, 2000.
3. D.A. Plaisted and S. Greenbaum. A Structure-preserving Clause Form Translation. *Journal of Symbolic Computation*, 2:293–304, 1986.
4. M. Cadoli, M. Schaerf, A. Giovanardi, and M. Giovanardi. An algorithm to evaluate quantified boolean formulae and its experimental evaluation. *Journal of Automated Reasoning*, 2000. To appear. Reprinted in [13].
5. Jussi T. Rintanen. Improvements to the evaluation of quantified boolean formulae. In Dean Thomas, editor, *Proceedings of the 16th International Joint Conference on Artificial Intelligence (IJCAI-99-Vol2)*, pages 1192–1197, S.F., July 31–August 6 1999. Morgan Kaufmann Publishers.
6. R. Feldmann, B. Monien, and S. Schamberger. A distributed algorithm to evaluate quantified boolean formulae. In *Proc. AAAI*, 2000.
7. Enrico Giunchiglia, Massimo Narizzano, and Armando Tacchella. Backjumping for quantified boolean logic satisfiability. In *Proc. of the International Joint Conference on Artificial Intelligence (IJCAI'2001)*, 2001.
8. M. Böhm and E Speckenmeyer. A fast parallel SAT-solver – efficient workload balancing. *Annals of Mathematics and Artificial Intelligence*, 17:381–400, 1996.
9. Robert G. Jeroslow and Jinchang Wang. Solving propositional satisfiability problems. *Annals of Mathematics and Artificial Intelligence*, 1:167–187, 1990.
10. Chu Min Li and Anbulagan. Heuristics based on unit propagation for satisfiability problems. In *Proceedings of the 15th International Joint Conference on Artificial Intelligence (IJCAI-97)*, pages 366–371, San Francisco, August 23–29 1997. Morgan Kaufmann Publishers.
11. Kleine-Büning, H. and Karpinski, M. and Flögel, A. Resolution for quantified boolean formulas. *Information and computation*, 117(1):12–18, 1995.
12. Ian Gent and Toby Walsh. Beyond NP: the QSAT phase transition. In *Proc. AAAI*, pages 648–653, 1999.
13. Ian P. Gent, Hans Van Maaren, and Toby Walsh, editors. *SAT2000. Highlights of Satisfiability Research in the Year 2000*. IOS Press, 2000.