On the effectiveness of backjumping and trivial truth in quantified boolean formulas satisfiability*

E. Giunchiglia[1], M. Narizzano[1], and A. Tacchella[2]

[1] DIST, Università di Genova, Viale Causa, 13 – 16145 Genova, Italy
[2] CS Dept., Rice University, 6100 Main St. MS 132, 77005-1892 Houston, Texas

Abstract. Trivial truth and backjumping are two optimization techniques that have been proposed for deciding quantified boolean formulas (QBFs) satisfiability. Both these techniques can greatly improve the overall performance of a QBF solver, but they are the expression of opposite philosophies. On one hand, trivial truth is a "look-ahead" policy: it is applied when descending the search tree to (try to) prune it. On the other hand, backjumping is a "look-back" policy: it is applied when backtracking to (try to) avoid useless explorations. Neither of these optimizations subsumes the other: it is easy to come up with examples in which trivial truth behaves much better than backjumping, and the other way around. In this paper we experimentally evaluate these two optimizations both on randomly generated and on real world test cases.

1 Introduction

Trivial truth [1, 2] and backjumping [3] are two optimization techniques that have been proposed for deciding quantified boolean formulas (QBFs) satisfiability. Both these techniques can greatly improve the overall performance of a QBF solver, but they are the expression of two opposite philosophies. On one hand, trivial truth is a "look-ahead" policy: it is applied when descending the search tree to (try to) prune it. On the other hand, backjumping is a "look-back" policy: it is applied when backtracking to (try to) avoid useless explorations. Neither of these two optimizations subsumes the other: it is easy to come up with examples in which trivial truth behaves much better than backjumping, and the other way around.

In this paper we experimentally compare trivial truth and backjumping. The comparison is done using the QUBE system [4]. While trivial truth is implemented by various QBF solvers like EVALUATE [1] and QSOLVE [5], QUBE is –as far as we know– the only system with a backjumping optimization built in. The experimental analysis is performed on a combination of random kQBFs (see [2]), generated according to "model A" proposed by Gent and Walsh in [6], and on the real world instances proposed by Rintanen in [7]. The overall conclusions are that

- on random kQBFs, backjumping becomes more effective as the value of k increases, while trivial truth is more effective for specific values of k;

* We wish to thank Marco Cadoli, Rainer Feldmann, Theodor Lettman, Jussi Rintanen, Marco Schaerf, Stefan Schamberger, for providing us with their systems and helping us to figure them out during our experimental analisys.

– on real world QBFs, trivial truth does not improve the performances, while backjumping enables QuBE to cope with some instances which could not be solved otherwise.

All the aforementioned issues are the subject of the next three sections. In particular, Section 2 is devoted to the preliminaries, which include a formal presentation of QBFs; a presentation of QuBE decision algorithm(s), including a high-level description of backjumping and trivial truth; and a presentation of the testing methodology. Section 3 is devoted to the experimental evaluation of backjumping and trivial truth. In Section 4 we experiment with QuBE, QKN [8], EVALUATE [1], DECIDE [7], and QSOLVE [5] in order to assess the impact of QuBE's optimizations given the current state of the art in QBF solvers. We end the paper in Section 5 by sorting out the clues of our experimental analisys from the issues that still require more investigation.

2 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) $\overline{l}$ 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 a p-ary $(p \geq 0)$ disjunction of literals such that no atom occurs twice in C. A *propositional formula* is an m-ary $(m \geq 0)$ 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

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

where every Q_i $(1 \leq i \leq n)$ is a quantifier, either existential $\exists$ or universal $\forall$; $x_1, \dots, x_n$ are pairwise distinct atoms in P; and Φ is a propositional formula in the atoms $x_1, \dots, x_n$. $Q_1 x_1 \dots Q_n x_n$ 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 $\{l, l_1, \dots, l_k\}$ 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).
– *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. For example, the empty clause (i.e., the clause with no literals) is contradictory.

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).

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

 3 function Simplify() {
 4   do {
 5     φ' = φ;
 6     if (⟨a contradictory clause is in φ⟩)
 7       return FALSE;
 8     if (⟨the matrix of φ is empty⟩)
 9       return TRUE;
10     if (⟨l is unit in φ⟩)
11       { |l|.mode = UNIT; Extend(l); }
12     if (⟨l is monotone in φ⟩)
13       { |l|.mode = PURE; Extend(l); }
14   } while (φ' != φ);
15   return UNDEF; }

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

24 function QubeSolver() {
25   do {
26     res = Simplify();
27     if (res == UNDEF) l = ChooseLiteral();
28     else  l = Backtrack(res);
29     if (l != NULL)  Extend(l);
30   } while (l != NULL);
31   return res; }
```

Fig. 1. The pseudo-code of QUBE.

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., [9]) it is always possible to rewrite a QBF into an equisatisfiable one satisfying our restrictions.

2.1 QUBE

QUBE is implemented in C on top of SIM [10], an efficient library for propositional satisfiability developed by our group. A C-like high-level description of QUBE is shown in Figure 1. Inspection of QUBE's main routine (*QubeSolver* in Figure 1) reveals that the solver works in the usual way, trying to simplify φ as much as possible (line 26) before choosing how to branch on truth values (line 27), or backtracking to some previous choice point (line 28) whenever a contradictory clause or an empty formula is reached. Clearly, the order in which *ChooseLiteral* picks its choice depends on how literals are arranged in the prefix of φ. Outermost literals are selected first, unless they appear in a contiguous sequence of literals under the scope of the same kind of quantifier: in this case we are free to instantiate the literals in such a sequence with an arbitrary order. In Figure 1 we used the following conventions and notations:

- 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 the input QBF, (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.

QUBE relies on a set of primitives on φ (and *Stack*) to solve the input QBF. The basic services have the purpose of assigning a truth value to a literal and of undoing such assignments. In particular:

- *Extend*(l) first pushes l and φ in the stack; then deletes the clauses of φ in which l occurs, and removes $\overline{l}$ from the others.
- *Retract*() pops the literal and the corresponding QBF that are on top of the stack: the literal is returned, while the QBF is assigned to φ.

Extend() and *Retract*() are thus the finest grained operations that can be performed on QUBE's data structure. At a coarser level, QUBE relies on:

- *Simplify*() (detailed in Figure 1) simplifies φ till a contradictory clause is generated (line 6), or the matrix of φ is empty (line 8), or no more simplifications involving unit clauses and monotone literals are 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*) (detailed in Figure 1) pops all the literals and the 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 $\overline{l}$ is returned (line 22). If no such literal exists, NULL is returned (line 23).

The trivial truth optimization is implemented in QUBE as follows. 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 [2]. In practice, this is done by triggering a flag which instructs QUBE's main algorithm to work as a SAT solver, i.e., to ignore universal literals when it comes to simplify the formula or to choose a branching variable. This optimization can produce dramatic speed-ups, particularly on randomly generated QBFs (see, e.g. [2]). Notice that ours is an optimized version of trivial truth as described in [2], where the check is performed at each branching node. In our version, such a check is not performed after branches on existential variables.

Backjumping comes into play when QUBE reaches a leaf node in the search tree, be it a contradictory clause or an empty formula. At this point, QUBE builds a reason for the current status, i.e., the set of literals directly causing it; then, while backtracking, QUBE refines such reason and tries to understand the branches that indirectly determined the satisfiability status in the leaf. In this way, QUBE can skip the other side of branches on existential literals which would lead again to a contradictory clause: this is the well known conflict-directed backjumping (CBJ) schema as used both in SAT (see, e.g., [11]) and CSP (see, e.g., [12]). QUBE is –as far as we know– the first QBF solver to implement such technique. In addition to performing CBJ, QUBE also skips the other side of branches on universal literals that would lead again to an empty formula: this is called *solution-directed* backjumping (SBJ) in [3] where such technique is introduced. In the following, we refer to the combination of CBJ and SBJ simply as "backjumping".

It turns out that, exactly as in SAT and CSP, backjumping has some overhead, but it can produce significant speed-ups [3].

Currently QuBE is distributed in two fashions: QuBE-BT, featuring standard chronological backtracking, and QuBE-BJ featuring backjumping. The choice between these two versions determines whether we are running the backjumping optimization or not. As for the trivial truth optimization, it is performed by default in both versions: we denote with "QuBE-BT -tt" and "QuBE-BJ -tt" the procedures corresponding, respectively, to QuBE-BT and QuBE-BJ with trivial truth turned off. QuBE's options other than backjumping and trivial truth are set according to their default values. As matter of fact, all the experiments have been run with the same version of *ChooseLiteral*, i.e., Böhm's heuristic [13] for SAT, modified as in [5] to consider existential literals only. Other configurations of QuBE, including those that we do not take into account here, are described in [4] to which we refer the reader for further details.

2.2 Testing Methodology

To experiment with various settings of QuBE and to compare it with other state-of-the-art solvers,[1] we have considered sets of random QBFs generated according to the "model A" as described in [6]. With this model, each QBF has the following 4 properties: (i) the prefix consists of k sequences, each sequence has n quantifiers, and each two quantifiers in the same 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. In our experiments we fix $h = 5$ and we run the solvers with (some of) the settings $k = 2, 3, 4, 5$ and $n = 50, 100, 150$, in order to understand how the effectiveness of backjumping and trivial truth is related to a change of those parameters. The choice of $h = 5$ is motivated by the analysis presented in [6]: here the authors show that "model A" instances with $h = 5$ turn out to be challenging on average. On the other hand, smaller values of h are not quite as good: for instance, according to [6], in the case of $h = 3$ only a few samples turn out to be really difficult for the current state-of-the-art solvers.

For every problem, i.e., every specific value of k and n, the range of l is experimentally chosen to cover at least the 100%-satisfiable - 100% unsatisfiable transition. In order to meet this requirement and to compare performances among different settings, we fixed $1 \leq l/n \leq 20$ for all the problems. Given a point, i.e., a specific value of k and l/n, we generate 100 samples with the program used in [14] to compare modal provers, and modified by us to halt and output the QBF samples before they are translated to modal formulas. For each sample and for all the solvers we measure the running time in CPU seconds, and for QuBE we also count the number of times that *ChooseLiteral* is called, i.e., the number of nodes in the search tree. Notice that when trivial truth is enabled, this includes the nodes explored by QuBE when running the "look-ahead" search. In order to save time, the execution of a solver on a sample is timed out after 1200s. Even more, for any point $k, l/n$, the execution of a solver is stopped after it exceeds the timeout on more than 50% of the samples, or:

[1] Experiments run on several identical Pentium III, 600MHz, 128MBRAM equipped with Linux Suse 6.x, kernel release 2.2.14. QuBE is compiled with GNU C release 2.95.2.

– up to 6 were attempted so far and at least 5 timed out, or
– up to 10 were attempted so far and at least 8 of them timed out, or
– up to 20 were attempted so far and at least 13 of them timed out.

In the following, we say that a point cannot be solved whenever any of the above conditions is met. For each point we consider the Q*-percentiles* for $Q = 50, 60, 70, 80, 90, 100$, the Q-percentile being the value such that $Q\%$ of the observations falls at or below it [15]. Our main statistical tool is the 50-percentile – best known as the *median* value – which we use as a resistant estimate for the average behaviour of the solvers on each point. The percentiles for $Q \geq 60$ serve the purpose of showing the behaviour on the hardest instances: in particular the 100-percentile corresponds to the worst case behaviour of the solvers on each point. When a point cannot be solved, the values of all the Q-percentiles default to 1200s.

Our experimental analysis includes the 38 problems contributed by Rintanen in [7].[2] The problems are translations from planning problems into the language of QBFs. They are interesting because:

– none of them can be solved by all the solvers at hand, and
– they share some "structure" with the original planning problem.

The second point is particularly relevant when comparing results on Rintanen's problems with results on "model A" instances. In SAT (see, e.g., [10]) it is often the case that the performance of some heuristics and optimizations is quite different when changing from randomly generated instances to real world ones. Therefore, although there is not yet any evidence in this direction for QBF, we included real world instances to make our evaluation of QuBE and other state-of-the-art solvers as wide as possible.

3 Backjumping vs. Trivial Truth

We start the review of our experimental results with the comparison showed in Figure 2 among the four versions of QuBE giving all the possible combinations of backjumping and trivial truth, including QuBE-BT -tt, i.e, QuBE's "vanilla" algorithm. The plots are arranged in two columns: running times are reported on the left and nodes are reported on the right. Going from top to bottom, we show the results of problems $k = 2, 3, 4, 5$ with $n = 50$. In every plot, the x-axis is the range of l/n, and the y-axis is either the running time or the number of nodes, both in logarithmic scale. The percentage of satisfiable samples is represented by the dotted curve in the background. Notice that in the plots we omit the points which could not be solved.

By looking at the plots in Figure 2 and focusing on the running times for $k = 2$ (top-left), we immediatly notice that trivial truth causes at least 3 orders of magnitude speed-up for the hardest formulas, i.e., those with $l/n = 4$. As a matter of fact, QuBE-BT median running time peaks at slightly less than 1s, while QuBE-BT -tt cannot solve more than 50% of the samples within the time limit. Still with $k = 2$, but looking at the number of nodes (Fig. 2 top-right), we see that the good performance of trivial truth is readily explained by a similar order-of-magnitude difference in the number of

[2] Available at `www.informatik.uni-freiburg.de/~rintanen/qbf.html`.

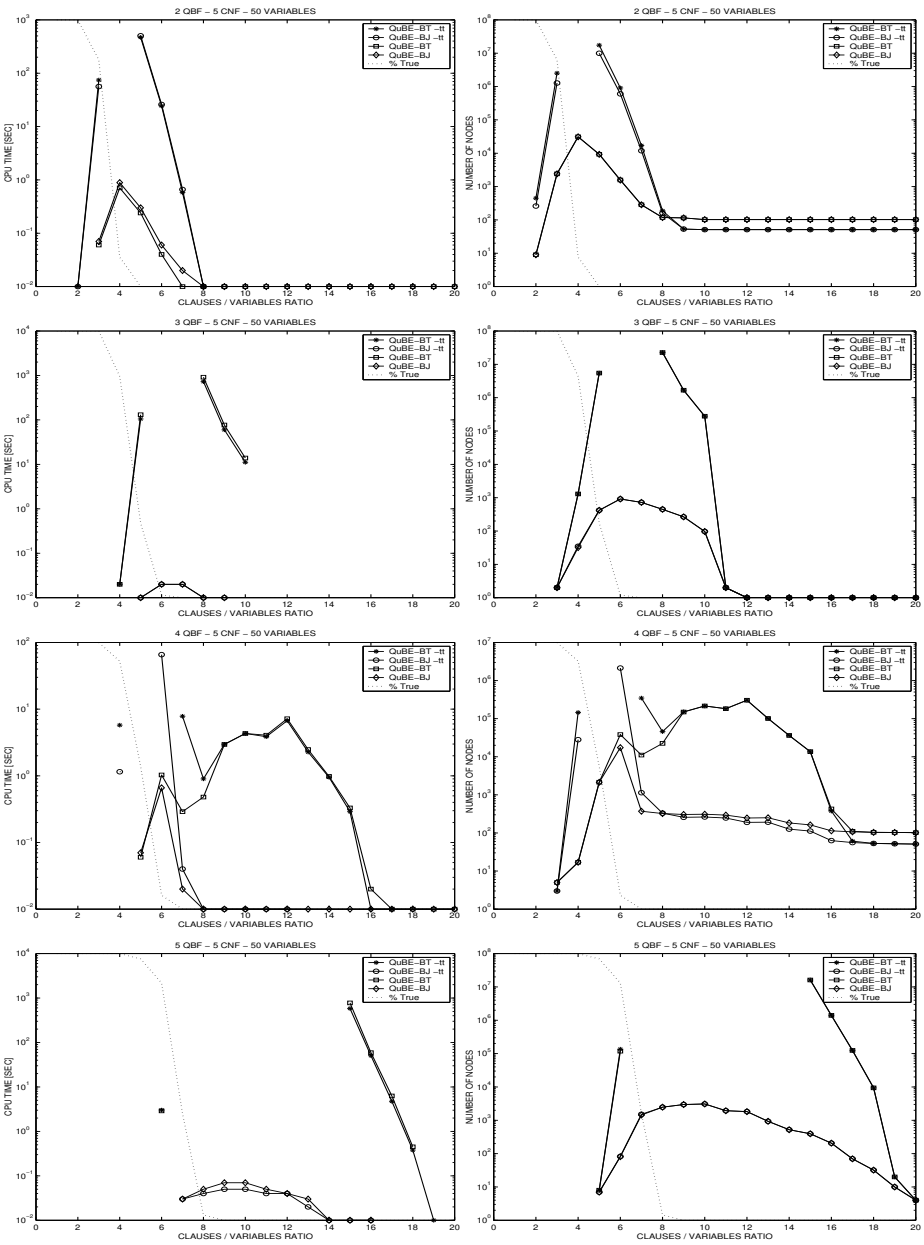

Fig. 2. QUBE median running time (left) and number of nodes (right) with $n = 50$, $k = 2, 3, 4, 5$.

nodes: the difference is even more impressive if we consider that such number includes the nodes explored by trivial truth. On the other hand, backjumping does not seem to pay off: although the median number of nodes for QUBE-BJ (resp. QUBE-BJ -tt) is

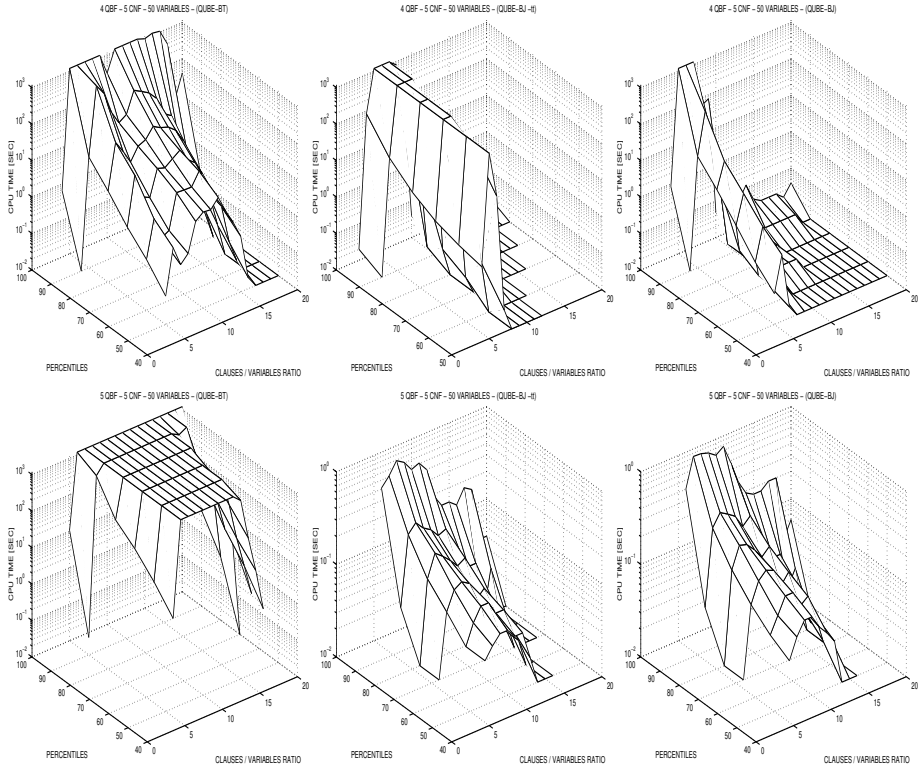

Fig. 3. QUBE Q-percentile plots for $n = 50$ $k = 4, 5$.

always less than or equal to the same statistic for QUBE-BT (resp. QUBE-BJ -tt), the difference is not enough to pay off the additional overhead. Indeed, it turns out that backjumping alone (QUBE-BT -tt) is not able to decrease the median running time of QUBE below the 1200s threshold for $l/n = 4$.

The situation is quite different for all values $k \geq 3$. Indeed, trivial truth has a much more limited impact, in particular for $k = 3$ (Fig. 2 second row) and $k = 5$ (Fig. 2 bottom row), while backjumping is delivering order-of-magnitude speed-ups on the same settings. This is utterly evident by looking at the running times for $k = 5$ (Fig. 2 bottom-left), where QUBE-BJ and QUBE-BJ -tt curves both peak at less than 0.1s, while QUBE-BT and QUBE-BT -tt both peak above the timeout threshold. Again, it is a similarly large reduction in the search space size (Fig. 2 bottom-right) that determines the difference in performance.

The plots with $k = 4$ (Fig. 2 third row) are even more interesting, since trivial truth and backjumping show to be incomparable even considering a single setting of k and l/n. Neither backjumping alone (QUBE-BJ -tt) nor trivial truth alone (QUBE-BT) are able to achieve optimum performances for $k = 4$. Looking at running times (left) we see that in the first part of the plot, up to $l/n = 6$, QUBE-BT is faster than QUBE-BJ -tt, but the situation is reversed in the second half of the plot, with the two curves rejoining

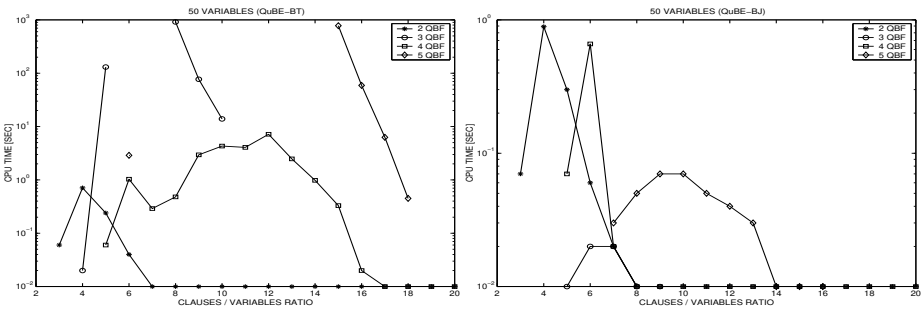

Fig. 4. QUBE-BT and QUBE-BJ versions with $n = 50$.

only for $l/n \geq 17$. The situation is similar when looking at the number of nodes (right), with the additional remark that QUBE-BJ (resp. QUBE-BJ -tt) always explores less search space than QUBE-BT (resp. QUBE-BT -tt). Interestingly, QUBE-BT median running time does not exceed the timeout threshold, while QUBE-BJ -tt peaks above it; nevertheless, the two optimizations can interact positively, since QUBE-BJ is able to solve all the points.

We now deepen our analysis on QUBE, by considering in more detail the problems $n = 50$ with $k = 4, 5$. In Figure 3 we show the Q-percentile plots for the running times of QUBE-BT, QUBE-BJ -tt and QUBE-BJ, i.e., all the versions of QUBE with at least one optimization turned on. In every plot, the x-axis is the range of l/n, the y-axis is the percentile range (from 50 to 100), and the z-axis is the time in CPU seconds on a logarithmic scale. Notice that when QUBE is exceeding the time limit we assign it the default value of 1200s: this corresponds to the presence of "plateaus" in the plots shown in Figure 3. Looking at the first column of Figure 3 (QUBE-BT) we immediatly see that trivial truth is more effective with $k = 4$ rather than with $k = 5$, but still the hardest samples (100-percentile) for $k = 4$ cannot be solved within the time limit. Interestingly, such samples seems to be located towards higher values of l/n where they are almost certainly unsatisfiable: this is to be expected, since trivial truth only helps in discovering trivially *satisfiable* subtrees. In the second column of Figure 3 (QUBE-BJ-tt) we observe that backjumping confirms its effectiveness for higher values of k, and in particular on unsatisfiable instances. Including the third column (QUBE-BJ) we also notice that, other things being equal, increasing k from 4 to 5 makes the problem easier for QUBE-BJ (notice the change of scale from top-row plots to bottom-row plots). As a final remark on the plots of Figure 3, we notice that the difference from the median running times (50-percentile) to the hardest samples (100-percentile) is contained in within one order of magnitude for $k = 5$ while it grows up to 3 orders of magnitude for $k = 4$. In other words, QUBE with all optimizations turned on, finds problems with $k = 4$ substantially more difficult than problems with $k = 5$. Although for the lack of space we do not show the results here, the same pattern can be observed between $k = 2$ and $k = 3$.

To better understand the variation in QUBE's search cost according to our optimizations and to different values of k, we rearranged our data as shown in Figure 4: QUBE-BT (left) and QUBE-BJ (right) median running times are plotted for each of

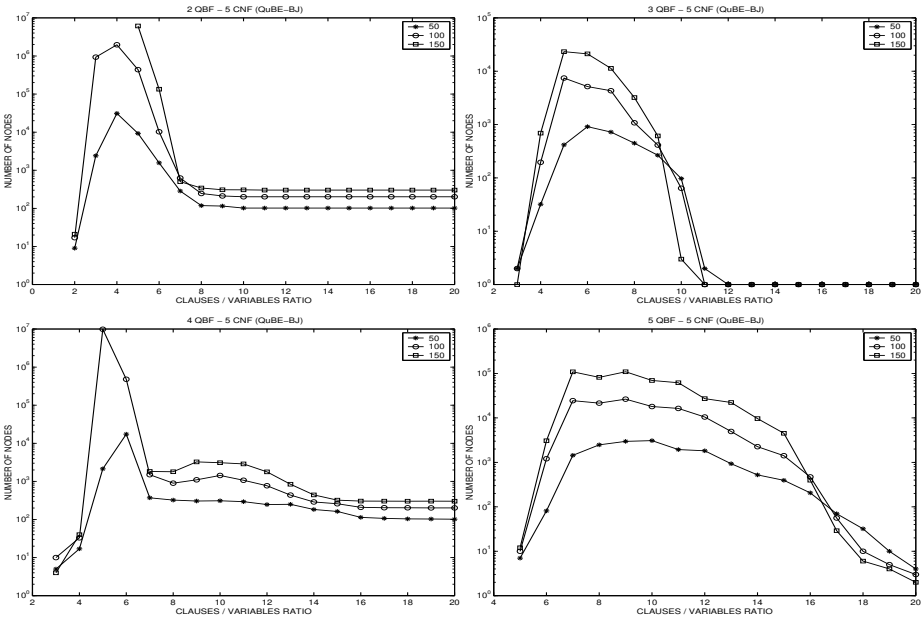

Fig. 5. QuBE-BJ on $k = 2, 3, 4, 5$ and $n = 50, 100, 150$ problems.

the values of k that we tried ($2 \leq k \leq 5$). Notice that it is not possible to compare the plots in a pointwise fashion, since instances with different value of k and same value of l/n can be of different hardness. This is due to the fact that the transition between 100%-satisfiable to 100%-unsatisfiable instances drifts towards the right for increasing values of k and the plots of QuBE's search cost seem to follow the same pattern. Therefore, saying here that a problem setting is "harder" than another one, corresponds to observing a higher (expected) peak time and/or a greater number of unsolved points. Back to Figure 4, by looking at QuBE-BT (left), we would be tempted to claim that instances with $k = 2i$ are easier than those with $k = 2i + 1$; nevertheless, instances with $k = 2i$ (resp. $k = 2i + 1$) are more difficult than $k = 2j$ (resp. $k = 2j + 1$) whenever $i < j$. In other words, trivial truth seems to be more effective when k is even, i.e., the prefix looks like $\forall \ldots \exists$, and less effective when k is odd, i.e., the prefix looks like $\exists \ldots \exists$. By looking at QuBE-BJ, (Fig. 4 right) we see a different trend, since the problems with $k = 2i$ are harder than those with $k = 2i + 1$; moreover, the problems with $k = 2i$ are harder than those with $k = 2j$ when $i < j$, while problems with $k = 2i + 1$ follow the same pattern as with QuBE-BT, i.e, they get harder with increasing values of i.

A complete picture of QuBE-BJ's performance on 12 different problems corresponding, respectively, to $k = 2, 3, 4, 5$ and $n = 50, 100, 150$ is presented in Figure 5: going from left to right, top to bottom, we show the median number of nodes of QuBE-BJ for different values of k and, in each plot, for different values of n. By contrasting the plots on the left column ($k = 2, 4$) and those on the right one ($k = 3, 5$) we immediatly confirm what we noticed by looking at Figure 4, i.e., QuBE-BJ finds problems

		QuBE-BT -tt	QuBE-BJ -tt	QuBE-BT	QuBE-BJ
Test File	Value	Time	Time	Time	Time
B..3ii.4.3	0	-	5.690	-	7.900
B..3ii.5.2	0	-	97.450	-	128.460
B..3iii.4	0	-	1.630	-	2.540
C..21v.22	1	140.630	163.950	-	-
C..22v.23	1	292.900	341.330	-	-
C..23v.24	1	605.050	701.300	-	-
T..6.1.iv.11	0	32.500	6.940	40.220	8.460
T..6.1.iv.12	1	14.610	1.830	19.330	2.330
T..7.1.iv.13	0	753.020	139.720	941.300	144.280
T..7.1.iv.14	1	311.830	28.910	416.700	36.090

Table 1. QUBE optimizations on (some) real world instances.

with an even k harder than those with an odd k, and this is independent from the number of variables. In all the plots, i.e., for every value of k, increasing the number of variables, increases the hardness of the instances; if k is even, QUBE-BJ is not able to solve some of the points: $l/n = 3, 4$ with $k = 2$, and $l/n = 5, 6$ with $k = 4$. If k is odd, QUBE-BJ is able to solve all the points, and the peak in the median number of nodes is in the order of 10^5 at most for $k = 5$, $n = 150$.

In Table 1 we show the results obtained by running QUBE on some of the real world samples that we discussed in Subsection 2.2. In particular we omit those instances (there are 28 of them) in which QUBE is either able (11) or not able (17) to solve all the instances, no matter what optimizations are tried. The remaning 10 instances are shown in Table 1, divided in three classes: "blocks-world" ("B" instances), "chain" ("C" instances), and "bomb in the toilet" problems ("T" instances). In the columns of Table 1, besides the satisfiability status, we report for each version of QUBE the running time in seconds: instances exceeding the time limit are denoted with a dash. In the case of "B" instances, backjumping is the optimization of choice: QUBE-BJ -tt as well as QUBE-BJ are able to solve all the three instances, while QUBE-BT -tt and QUBE-BT are not. We also notice that trivial truth is causing a performance degradation in QUBE-BJ which turns out to be slower than QUBE-BJ -tt. With "C" instances, even QUBE's vanilla version is able to solve the corresponding QBFs, while backjumping does not seem to pay off. Noticeably, the versions featuring trivial truth are not able to solve the instances within the time limit. With "T" instances, QUBE-BJ -tt is again the fastest on all the problems, thus confirming the effectiveness of backjumping.

4 QUBE and state-of-the-art QBF solvers

In Figure 6 we review the experiments comparing state-of-the-art solvers with QUBE. Going from left to right, top to bottom, we show the problems $k = 2, 3, 4, 5$ with $n = 100$. The plots are arranged in the same way as in Figure 2. We run QUBE-BJ and QSOLVE [5] with all the settings, while for $k = 2$ we also tested DECIDE [7],

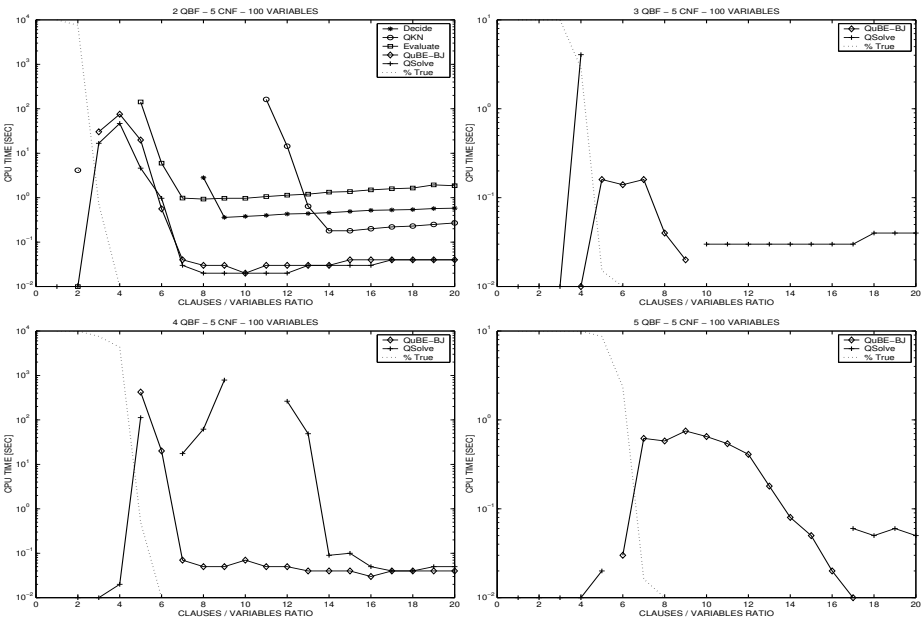

Fig. 6. QUBE and state-of-the-art solvers on $k = 2, 3, 4, 5$ and $n = 100$ problems.

EVALUATE [1] and QKN [8]. Looking at Figure 6 (top 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: EVALUATE, QKN and DECIDE all fail to solve the hardest points. Still in Figure 6, for $k \geq 3$ 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 further evidence on the effectiveness of backjumping for values of $k \geq 3$. In particular, we remark the qualitative similarity between QSOLVE with $k = 4$ and $n = 100$ in Figure 6 and QUBE-BT with $k = 4$ and $n = 50$ in Figure 2, and the fact that all the considerations that we did on $n = 50$ seems to carry over to $n = 100$.

In Figure 7 we take a closer look to QUBE-BJ (left) and QSOLVE (right): in the plots we show the median running times for $n = 100$ and the values $k = 2, 3, 4, 5$. QSOLVE finds problems with an even k somewhat easier than those with an odd k, and it is not able to solve all the points for $k \geq 3$; on the other hand, QUBE-BJ finds problems with $k \geq 3$ easier than those with $k = 2$. As with $n = 50$ in Figure 4, the comparison above is not pointwise, because also for $n = 100$ the crossover between 100%-satisfiable to 100%-unsatisfiable instances shifts to the right: both QUBE-BJ and QSOLVE median running times follow the same pattern. Notice that, in spite of the different implementation and the use of trivial falsity, QSOLVE qualitative behaviour on $n = 100$ closely resembles the one of QUBE-BT on $n = 50$.

We conclude the comparison between QUBE and the other state-of-the-art solvers with the results presented in Table 2. In the Table, we report the running times of QUBE and other solvers on 33 out of 38 problems: the remaining instances could not be solved

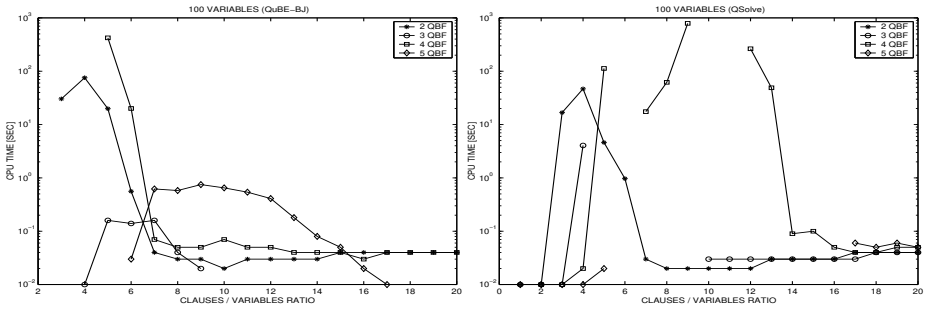

Fig. 7. QUBE and QSOLVE on $k = 2, 3, 4, 5$ and $n = 100$ problems.

by any of the systems. As we can see from Table 2, the best solver overall turns out to be DECIDE with 33 problems solved and 40.22s average running time (on solved samples), followed by QUBE, with 21 problems solved and 78.48s, and QSOLVE with 11 problems solved and 142.28s. 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. Athough QUBE does not feature such mechanisms its performances are quite good on these tests: on "C" instances this is entirely due to its heuristics and efficient implementation, while backjumping turns out to be the crucial factor in the solution of "B" and "T" instances.

5 Conclusions

Our experimental evaluation of backjumping and trivial truth confirms that the two optimizations are of incomparable strengths, although they both help in obtaining substantial speed-ups. The analysis of Section 3 suggests the following clues:

- on random kQBFs, backjumping is more effective with $k \geq 3$ while trivial truth is very effective with $k = 2$, and less effective with $k \geq 3$; once combined, the two optimizations either interact positively, or their relative overhead is small;
- on real world QBFs, trivial truth does not improve the performances, while backjumping enables QUBE to solve some "blocks world" and "bomb in the toilet" instances which could not be solved otherwise.

Therefore, at least for the instances that we analyzed, the bottom line is: the combination of backjumping and trivial truth pays off on "model A" instances, and backjumping alone is better on real world instances. Still, our experimental analysis does leave some questions unsettled, which we would like to investigate further:

- rather unintuitively, QUBE finds "model A" instances with $k = 2i$ more difficult than those with $k = 2i + 1$: it is not clear whether this phenomenon is specific to QUBE and the way its optimizations are implemented, or if it happens in combination with a particular choice of samples and/or generation method;

		QuBE-BJ -tt	**QSolve**	**Evaluate**	**QKN**	**Decide**
Test File	Value	Time	Time	Time	Time	Time
B..3i.4.4	0	-	-	-	-	0.11
B..3i.5.3	0	-	-	-	-	49.72
B..3i.5.4	1	-	-	-	-	7.53
B..3ii.4.3	0	5.690	-	-	-	0.07
B..3ii.5.2	0	97.450	-	-	-	0.41
B..3ii.5.3	1	-	-	-	-	1.80
B..3iii.4	0	1.630	-	-	-	0.02
B..3iii.5	1	-	-	-	-	0.56
B..4i.6.4	0	-	-	-	-	3.72
B..4ii.6.3	0	-	-	-	-	3.10
B..4ii.7.2	0	-	-	-	-	8.79
B..4iii.6	0	-	-	-	-	1.51
B..4iii.7	1	-	-	-	-	729.11
C..12v.13	1	0.220	6.45	-	-	0.07
C..13v.14	1	0.470	20.1	-	-	0.10
C..14v.15	1	0.970	63.47	-	-	0.12
C..15v.16	1	2.020	199.31	-	-	0.16
C..16v.17	1	4.200	624.97	-	-	0.19
C..17v.18	1	8.920	-	-	-	0.23
C..18v.19	1	18.480	-	-	-	0.28
C..19v.20	1	38.590	-	-	-	0.31
C..20v.21	1	85.400	-	-	-	0.36
C..21v.22	1	163.950	-	-	-	0.40
C..22v.23	1	341.330	-	-	-	0.48
C..23v.24	1	701.300	-	-	-	0.56
T..10.1.iv.20	1	-	-	-	-	6.30
T..16.1.iv.32	1	-	-	-	-	43.15
T..2.1.iv.3	0	0.000	0	-	0.147	0.01
T..2.1.iv.4	1	0.000	0	-	-	0.02
T..6.1.iv.11	0	6.940	11.81	-	-	36.17
T..6.1.iv.12	1	1.830	8.53	-	-	0.91
T..7.1.iv.13	0	139.720	426.93	-	-	428.20
T..7.1.iv.14	1	28.910	203.53	-	-	1.64

Table 2. QUBE and state-of-the-art solvers on (some) real world instances.

– the number of real world benchmarks available at the moment is not yet large enough to make conclusive claims about the effectiveness of the optimizations on these kind of instances.

To settle the first issue, we need to learn more about the distinguishing features of "model A" samples, and the way QUBE and other QBF solvers deal with them. Also, it might be interesting to pursue different models of generation and to investigate their properties in comparison with "model A". About the second issue, QBF reasoning is rapidly moving from theoretical investigation to practical applications and, under this

perspective, the goal of increasing the real world benchmark base does not seem to be out of reach.

As a concluding remark, we wish to point out that QUBE is very competitive with the current state-of-the-art solvers, mainly because of its optimizations: this is true on "model A" instances for all values of k and n that we tried, but in particular with $k = 3, 4, 5$ and $n = 100$; on real world instances, QUBE with backjumping comes second best after DECIDE which is indeed geared towards these problems. We take these results as a first promising evidence of QUBE's implementation and algorithmic strengths, and of its effectiveness as a research and development tool for QBF applications.

References

1. M. Cadoli, A. Giovanardi, and M. Schaerf. An algorithm to evaluate quantified boolean formulae. In *Proc. of AAAI*, 1998.
2. M. Cadoli, M. Schaerf, A. Giovanardi, and M. Giovanardi. An Algorithm to Evaluate Quantified Boolean Formulae and its Experimental Evaluation. In *Highlights of Satisfiability Research in the Year 2000*. IOS Press, 2000.
3. E. Giunchiglia, M. Narizzano, and A. Tacchella. Backjumping for Quantified Boolean Logic Satisfiability. In *Proc. of IJCAI*, 2001. To appear.
4. E. Giunchiglia, M. Narizzano, and A. Tacchella. QuBE: a system for Deciding Quantified Boolean Formulas Satisfiability. In *Proc. of IJCAR*, 2001.
5. R. Feldmann, B. Monien, and S. Schamberger. A Distributed Algorithm to Evaluate Quantified Boolean Formulae. In *Proc. of AAAI*, 2000.
6. Ian Gent and Toby Walsh. Beyond NP: the QSAT phase transition. In *Proc. of AAAI*, pages 648–653, 1999.
7. J. T. Rintanen. Improvements to the Evaluation of Quantified Boolean Formulae. In *Proc. of IJCAI*, pages 1192–1197, 1999.
8. H. Kleine-Büning and M. Karpinski and A. Flögel. Resolution for quantified boolean formulas. *Information and Computation*, 117(1):12–18, 1995.
9. D.A. Plaisted and S. Greenbaum. A Structure-preserving Clause Form Translation. *Journal of Symbolic Computation*, 2:293–304, 1986.
10. E. Giunchiglia, M. Maratea, A. Tacchella, and D. Zambonin. Evaluating search heuristics and optimization techniques in propositional satisfiability. In *Proc. of IJCAR*, LNCS. Springer Verlag, 2001. To appear.
11. 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.
12. Patrick Prosser. Hybrid algorithms for the constraint satisfaction problem. *Computational Intelligence*, 9(3):268–299, 1993.
13. M. Buro and H. Kleine-Büning. Report on a SAT competition. Technical Report 110, University of Paderborn, November 1992.
14. F. Massacci and F. Donini. Design and Results of TANCS-2000 Non-classical (Modal) Systems Comparison. In *Proc. of TABLEAUX*, volume 1847 of *LNAI*, pages 52–56. Springer Verlag, 2000.
15. D. S. Moore and G. P. McCabe. *Introduction to the Practice of Statistics*. W. H. Freeman and Co., 1993.