Quantifier structure in search based procedures for QBFs [*]

Enrico Giunchiglia Massimo Narizzano Armando Tacchella
DIST - Università di Genova Viale Causa 13, 16145 Genova, Italy
E-mail: enrico,mox,tac@star.dist.unige.it

Abstract

The best currently available solvers for Quantified Boolean Formulas (QBFs) process their input in prenex form, i.e., all the quantifiers have to appear in the prefix of the formula separated from the purely propositional part representing the matrix. However, in many QBFs deriving from applications, the propositional part is intertwined with the quantifier structure. To tackle this problem, the standard approach is to first convert them in prenex form, thereby loosing structural information about the prefix.

In this paper we show that conversion to prenex form is not necessary, i.e., that it is relatively easy to extend current search based solvers in order to exploit the original quantifier structure, i.e., to handle non prenex QBFs. Further, we show that the conversion can lead to the exploration of search spaces bigger than the space explored by solvers handling non prenex QBFs. To validate our claims, we implemented our ideas in the state-of-the-art search based solver QUBE*, and conducted an extensive experimental analysis. The results show that very substantial speedups can be obtained.*

1. Introduction

The use of Quantified Boolean Formulas (QBFs) to encode problems arising from various application domains, expcecially from Formal Verifications, has attracted increasing interest in recent years (see, e.g., [15, 1, 13]). The application-driven quest for efficiency has in turn propelled the research on decision procedures in order to deal with the size and the complexity of the QBF encodings (see [3] for a recent account on the state of the art in QBF reasoning). Considering the best currently available solvers, all of them assume that the input QBF

1. is in prenex form, i.e., all the quantifiers have to appear in the prefix of the formula separated from the purely propositional part; and
2. is in conjunctive normal form (CNF), i.e., the propositional part of the formula (called matrix) consists of a set of clauses.

However, in many QBFs deriving from applications, the propositional part is intertwined with the quantifiers structure and the matrix is not in CNF. The situation is simpler in the propositional satisfiability (SAT) case, corresponding to QBFs in which all the quantifiers are existential: in SAT, the first problem does not show up, and several papers have been dedicated to efficient and effective conversions to CNF and/or to the implementation of SAT solvers able to handle non CNF formulas (see, e.g., [16, 7] for two recent papers on these issues). The solutions devised in SAT to handle non CNF formulas can be easily lifted to the more complex QBF case. Still, in the QBF case we are left with the first issue. Indeed, the standard solution is to convert any non prenex QBF into a prenex one using standard quantifier rewriting rules like

$$(\exists x\Phi(x) \wedge \forall y\Psi(y)) \mapsto \exists x\forall y(\Phi(x) \wedge \Psi(y))$$

or

$$(\exists x\Phi(x) \wedge \forall y\Psi(y)) \mapsto \forall y\exists x(\Phi(x) \wedge \Psi(y)).$$

However, in the resulting QBF, the information that x and y are not one in the scope of the other is lost. Further, as the above simple example shows, there can be more than one rule applicable at each step and the result may vary depending on which rule is applied. In general, given a non prenex QBF φ, there can be exponentially many QBFs (i) in prenex form, (ii) equivalent to φ, and (iii) each of them obtainable from φ using the above mentioned rewriting rules. Thus, it is not clear which of these exponentially many QBFs is best, i.e., leads to the best performances when coupled with a QBF solver. Egly, Seidl, Tompits, Woltran and Zolda [6] define four strategies which are guaranteed to be optimal in the sense that the resulting QBF is guaranteed to belong to the lowest possible complexity class in the polynomial hierarchy. Their experimental analysis, conducted on a series of instances encoding knowledge representation problems and involving the best QBF solvers based on search, showed that

[*]This work is partially supported by MIUR

the strategy delivering the best performances depends both on the kind of instances and on the internals of the QBF solver.

In this paper we show that conversion to prenex form is not necessary, i.e., that it is relatively easy to extend current search based solvers in order to exploit the original quantifier structure, i.e., to handle non prenex QBFs. Further, we show that the conversion can have severe drawbacks on the heuristic and pruning techniques of the solvers, leading to the exploration of search spaces bigger than the space explored by solvers handling non prenex QBFs. To validate our claims, we implemented our ideas in the state-of-the-art search based solver QUBE, and conducted an extensive experimental analysis. The results show that very substantial speedups can be obtained.

2 The logic of QBFs

To focus on the problem we deal with, we consider QBFs in which the quantifiers may be not in prenex form, but in which the matrix is in CNF.

Consider a set P of *variables*. A *literal* is a variable or the negation $\overline{z}$ of a variable z. In the following, for any literal l,

- $|l|$ is the variable occurring in l; and
- $\overline{l}$ is $\overline{l}$ if l is a variable, and is $|l|$ otherwise.

A *clause* is a finite disjunction of literals. Finally,

- if $c_1, \ldots, c_n$ are clauses, $(c_1 \wedge \ldots \wedge c_n)$ is a QBF,
- if Φ is a QBF and z is a variable, $Qz\Phi$ is a QBF, where Q is either the existential quantifier "$\exists$" (in which case we say that z and $\overline{z}$ are *existential*) or the universal quantifier "$\forall$" (in which case we say that z and $\overline{z}$ are *universal*). In $Qz\Phi$, Φ is called the *scope* of Qz, and z is the variable *bound* by Q.
- if $\Phi_1, \ldots, \Phi_n$ are QBFs, $(\Phi_1 \wedge \ldots \wedge \Phi_n)$ is a QBF.

For simplicity, we restrict our attention to *closed QBFs*, i.e., to QBFs in which each variable is bound by a quantifier.

For example,

$$\begin{array}{c}\exists x_0(\forall y_1 \exists x_1 \exists x_2((x_0 \vee \overline{x}_1 \vee \overline{x}_2) \wedge (y_1 \vee \overline{x}_1 \vee x_2) \wedge \\ (x_1 \vee \overline{x}_2) \wedge (x_0 \vee x_1 \vee x_2)) \wedge \\ \forall y_2 \exists x_3 \exists x_4((\overline{x}_0 \vee \overline{x}_3 \vee \overline{x}_4) \wedge (y_2 \vee \overline{x}_3 \vee x_4) \wedge \\ (x_3 \vee \overline{x}_4) \wedge (\overline{x}_0 \vee x_3 \vee x_4)))\end{array} \tag{1}$$

is a closed QBF. Further, we assume that in a QBF there are no two distinct quantifiers that bind the same variable. With this assumption, we can represent any QBF as a pair

- the *prefix*, being a partially ordered set in which (i) each element of the set has the form $\langle quantifier, boundvariable \rangle$; and (ii) two elements $\langle Q_1 z_1 \rangle$ and $\langle Q_2 z_2 \rangle$ in the set are in partial order (and we write $z_1 \prec z_2$) if and only if $Q_2 z_2$ occurs in the scope of $Q_1 z_1$, and
- the *matrix*, consisting of a set of clauses.

Since we will use x_i (resp. y_i) to denote an existentially (resp. universally) quantified variable,[1] we can simply represent the prefix with the partial order. For example, the prefix of (1) corresponds to the transitive closure of

$$x_0 \prec y_1, y_1 \prec x_1, x_1 \prec x_2, \ \ x_0 \prec y_2, y_2 \prec x_3, x_3 \prec x_4. \tag{2}$$

Notice that for a QBF in which no variable is in the scope of another, our representation of the prefix will be empty.

About the matrix, we use the standard SAT notation, and we represent each clause as the set of literals in it. Thus, the matrix of (1) is written as

$$\begin{array}{c}\{\{x_0, \overline{x}_1, \overline{x}_2\}, \{y_1, \overline{x}_1, x_2\}, \{x_1, \overline{x}_2\}, \{x_0, x_1, x_2\}, \\ \{\overline{x}_0, \overline{x}_3, \overline{x}_4\}, \{y_2, \overline{x}_3, x_4\}, \{x_3, \overline{x}_4\}, \{\overline{x}_0, x_3, x_4\}\}\end{array} \tag{3}$$

Consider a QBF φ with prefix $\prec$ and matrix Φ. The semantics of φ can be defined recursively as follows. Define the *prefix level* of a variable z as the length of the longest chain $z_1 \prec z_2 \prec z_n \prec z$ ($n \geq 0$) in the prefix such that z_i and z_{i+1} are differently quantified. For instance, in (1) the prefix level of x_0 is 1, while both x_1 and x_2 have prefix level 3. A variable z to be *top in* φ if it has prefix level 1. If the matrix of φ is empty, then φ is true. If the matrix of φ contains an empty clause, then φ is false. If z is top in φ and z is existential (respectively universal), φ is true if and only if the QBF φ_z or (respectively and) $\varphi_{\overline{z}}$ are true. If l is a literal, φ_l is the QBF

- whose matrix is obtained from Φ by (i) eliminating the clauses C such that $l \in C$, and eliminating $\overline{l}$ from the other clauses in Φ; and
- whose prefix is obtained from $\prec$ by removing the pairs $|l|, z$ such that $|l| \prec z$ or $z \prec |l|$.

3 *Q-DLL*

Most of the available QBF solvers assume that the input formula is in prenex form. For us, a QBF φ is in *prenex form* if its prefix is a total order.

Consider a QBF φ in prenex form, with prefix $\prec$ and matrix Φ.

A simple procedure for determining the value of φ, starts with φ and recursively simplifies the current φ to φ_z and/or $\varphi_{\overline{z}}$, where z is a heuristically chosen top variable in φ, till either the empty clause or the empty set of clauses are produced: on the basis of the values of φ_z and $\varphi_{\overline{z}}$, the value of φ can be determined according to the semantics of QBFs.

Cadoli, Giovanardi and Schaerf [5] introduced various improvements to this basic procedure.

[1] From a formal point of view, this amounts to divide the set P of variables in two disjoint sets $P_x = \{x, x_1, x_2, \ldots\}$ and $P_y = \{y, y_1, y_2, \ldots\}$, being respectively the set of existentially and universally quantified variables.

```
0 function Q-DLL($\varphi$)
1   if (⟨a contradictory clause is in $\varphi$⟩) return FALSE;
2   if (⟨the matrix of $\varphi$ is empty⟩) return TRUE;
3   if (⟨$l$ is unit in $\varphi$⟩) return Q-DLL($\varphi_l$);
4   $l$ := ⟨a top literal in $\varphi$⟩;
5   if (⟨$l$ is existential⟩) return Q-DLL($\varphi_l$) or Q-DLL($\varphi_{\overline{l}}$);
6   else return Q-DLL($\varphi_l$) and Q-DLL($\varphi_{\overline{l}}$).
```

Figure 1. The algorithm of *Q-DLL*.

The first improvement is that we can directly conclude about the value of φ if Φ contains a contradictory clause. A clause C is *contradictory* if it contains no existential literal. An example of a contradictory clause is the empty clause.

The second improvement allows us to directly simplify φ to φ_l if l is unit in φ. A literal l is *unit* in φ if l is existential and for some $m \geq 0$,

- a clause $\{l, l_1, \ldots, l_m\}$ belongs to Φ; and
- each literal l_i ($1 \leq i \leq m$) is universal and such that $|l_i| \not\prec |l|$, i.e., it is not the case that $|l_i| \prec |l|$.

With such improvements, the resulting procedure, called *Q-DLL*, is essentially the one presented in [5], which extends the famous Davis-Logemann-Loveland procedure *DLL* for (SAT). Figure 1 is a simple, recursive presentation of *Q-DLL*. In the figure, given a QBF φ,

1. FALSE is returned if a contradictory clause is in the matrix of φ (line 1); otherwise
2. TRUE is returned if the matrix of φ is empty (line 2); otherwise
3. at line 3, φ is recursively simplified to φ_l if l is unit; otherwise
4. at line 4 a top literal l is chosen (and we say that l has been *assigned as a branch*) and
   - if l is existential (line 5), the "**or**" of the results of the evaluation of φ_l and $\varphi_{\overline{l}}$ is returned;
   - otherwise (line 6), l is universal, and the "**and**" of the results of the evaluation of φ_l and $\varphi_{\overline{l}}$ is returned.

Q-DLL is correct: it returns TRUE if the input QBF is true and FALSE otherwise.

As it is the case in SAT, real implementations of *Q-DLL* extend the basic algorithm by allowing for more powerful simplification rules (e.g., pure literal fixing), intelligent backtracking (e.g., nogood and/or good learning), heuristics for deciding on which literal to branch on. Examples of solver featuring the above characteristics are QUBE [11], YQUAFFLE [17], and SEMPROP [12]: see the respective papers for more details.

4 Partial order vs Total order prefixes

Consider a QBF φ, with prefix $\prec$ and matrix Φ, and assume that $\prec$ is arbitrary, i.e., not necessarily in prenex form.

As we already anticipated in the introduction, in order to decide the value of φ, the standard approach is to first convert φ into prenex form, and then use one of the available solvers. This is the approach followed, e.g., in [6, 13]. The conversion can be easily done by simply extending the prefix till we get a total order. However, this can have some serious drawbacks detailed in the following.

The first important observation is that *Q-DLL* in Figure 1 does not rely on $\prec$ to be a total order. In other words, *Q-DLL* maintains its correctness even when the prefix of $\prec$ is not a total order. A possible execution of *Q-DLL* on (1) is represented by the tree in Figure 2. In the Figure, each node of the tree is labeled with a set of clauses and is numbered according to the order in which *Q-DLL* explores the search space; the root node has the input matrix (3) as label, and the other nodes contain the matrices resulting from the simplifications performed along the path from the root to each of them; each leaf is marked with $\{\{\}\}$ to denote that the resulting set of clauses contains at least an empty clause; branches in the tree correspond to the choice of a literal whose both values have to be tried; straight lines stand for unit literals or branching literals that are not subject to backtracking. As it can be seen from the figure, *Q-DLL* may, e.g., assign x_1 as a branch without having assigned y_2 before (and this in a total order setting would imply $x_1 \prec y_2$) and assign y_2 later[2] as a branch without having assigned x_1 before (and this in a total order setting would imply $y_2 \prec x_1$): since it is not possible to have both $x_1 \prec y_2$ and $y_2 \prec x_1$, the search tree showed in Figure 2 cannot be explored by *Q-DLL* if run on a QBF with the same matrix and a total order prefix extending (2).

Even if *Q-DLL* can work with QBFs in non prenex form, the advantage of having a totally ordered vs a partially ordered prefix is that the former is simpler to handle than the latter. However there can be exponentially many, pairwise non equivalent, prefixes extending the prefix of φ. Two prefixes are *equivalent* if removing from them the pairs z, z' such that both z and z' are either existential or universal leads to the same set of pairs. Given this fact, it is not clear which of these prefixes is best, i.e., leads to the best performances once coupled with the desired QBF solver. In [6], the authors define four strategies which are optimal in the sense that each strategy leads to an optimal prefix: there is no prefix extending $\prec$ with a smaller number of alterna-

[2] Considering the QBF (1), it can be objected that both y_1 and y_2 could be eliminated during the preprocessing since they are pure literal: a slightly more complicated example in which this critique does not apply and all the considerations we make still hold, can be obtained by simply adding the two clauses $\{\overline{y}_1, \overline{x}_1, x_2\}$ and $\{\overline{y}_2, \overline{x}_3, x_4\}$ to the matrix.

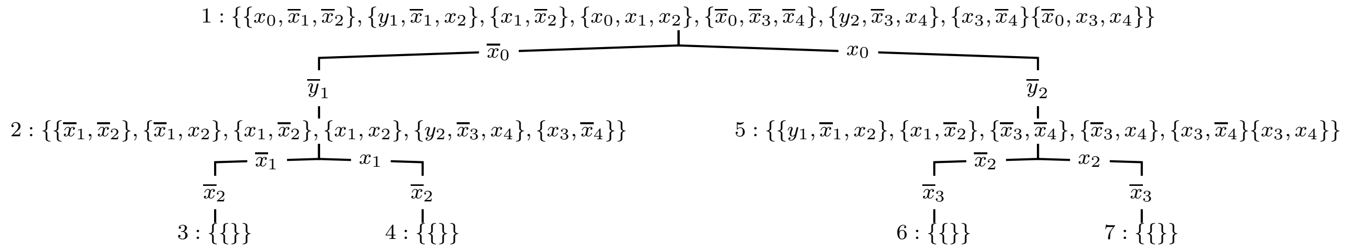

$1 : \{\{x_0, \overline{x}_1, \overline{x}_2\}, \{y_1, \overline{x}_1, x_2\}, \{x_1, \overline{x}_2\}, \{x_0, x_1, x_2\}, \{\overline{x}_0, \overline{x}_3, \overline{x}_4\}, \{y_2, \overline{x}_3, x_4\}, \{x_3, \overline{x}_4\}\{\overline{x}_0, x_3, x_4\}\}$
$\overline{x}_0$
x_0
$\overline{y}_1$
$\overline{y}_2$
$2 : \{\{\overline{x}_1, \overline{x}_2\}, \{\overline{x}_1, x_2\}, \{x_1, \overline{x}_2\}, \{x_1, x_2\}, \{y_2, \overline{x}_3, x_4\}, \{x_3, \overline{x}_4\}\}$
$5 : \{\{y_1, \overline{x}_1, x_2\}, \{x_1, \overline{x}_2\}, \{\overline{x}_3, \overline{x}_4\}, \{\overline{x}_3, x_4\}, \{x_3, \overline{x}_4\}\{x_3, x_4\}\}$
$\overline{x}_1$
x_1
$\overline{x}_2$
x_2
$\overline{x}_2$
$\overline{x}_2$
$\overline{x}_3$
$\overline{x}_3$
$3 : \{\{\}\}$
$4 : \{\{\}\}$
$6 : \{\{\}\}$
$7 : \{\{\}\}$

Figure 2. Search tree of *Q-DLL* on the QBF with the matrix at the root and the prefix corresponding to $x_0 \prec y_1, y_1 \prec x_1, x_1 \prec x_2, x_0 \prec y_2, y_2 \prec x_3, x_3 \prec x_4$.

tions.[3] In the case of the QBF (1) the optimal prefixes are the ones satisfying (2) and also: $y_1 \prec x_3,\ y_2 \prec x_1$. Obtaining a QBF with a minimal number of alternations is an important property, at least theoretically: a QBF with k alternations belongs to a complexity class which is contained in the complexity class to which a QBF with $k+1$ alternations belong, see [14]. However, in general there can be exponentially many strategies which are optimal in the above sense, i.e., there can be exponentially many, pairwise non equivalent, and with the minimal number of alternations prefixes extending a given prefix $\prec$. Further, the experimental analysis conducted in [6] shows that even restricting to the 4 optimal strategies there defined, the strategy delivering the best performances depends both on the kind of instances and on the internals of the QBF solver.

Furthermore, no matter which strategy is used, be it optimal or not, imposing a total order on the prefix can have substantial drawbacks if a solver based on *Q-DLL* is used:

1. when deciding which literal to assign as a branch, the selection is restricted among the top literals: imposing a total order on the prefix can severely limit the choice up to the point that the heuristic becomes static. Considering, e.g., an instance φ with three variables x_1, x_2 and y and prefix either $x_1 \prec x_2$ or $y \prec x_2$: the prefix according to which $x_1 \prec y \prec x_2$ imposes a fixed, static ordering on the atoms to branch on. In the case of the QBF (1), we already pointed out that there is no total order allowing to explore the search tree in Figure 2. Given that the search tree in the Figure is optimal (any other search tree possibly explored by *Q-DLL* on (1) has a bigger number of literals assigned as branches) it trivially follows that extending (1) to a total order will necessarily cause the exploration of a search tree bigger than that in the figure.
2. when checking if a literal l is unit, we search for clauses in which l is the only existential literal, and all the other literals l' are such that $|l'| \not\prec |l|$: in the case we extend the prefix to a total order, for each pair of distinct literals l and l', either $|l| \prec |l'|$ or $|l'| \prec |l|$, and as consequence some literals may no longer be detected as unit. In the case of the QBF (1), if we consider the clause $\{y_1, x_2, x_3, x_4\}$ obtained by resolving the second, fourth and last of the clauses in (3), once, e.g., x_2 and x_3 are assigned to false, x_4 can be propagated as unit: This would not happen if the partial order is extended with $y_1 \prec x_4$. These kind of clauses can be generated either in the preprocessing and/or during the search if the solver implements nogood and/or good learning.[4]

[3]The *number of alternations* in a QBF φ is the maximal of the prefix levels of the variables in φ minus 1. The number of alternations in (1) is 2.

5 Exploiting Quantifier Structure in QUBE

We implemented the algorithm described in the previous section on top of the state-of-the-art solver QUBE [11]. QUBE reads instances in prenex form and features state-of-the-art backtracking techniques, heuristics and data structures. To describe QUBE's features prior to this work, we use QUBE(TO) to denote the old version of QUBE solving QBF instances in prenex form, and QUBE(PO) to denote the version of QUBE modified in order to exploit the quantifier structure.

QUBE(PO) main difference with respect to QUBE(TO) is in the heuristic. The heuristic in QUBE(TO) is implemented by associating a counter to each literal storing number of constraints c such that $l \in c$. Each time a constraint is added, the counter is incremented; when a learned constraint is removed, the counter is decremented. In order to choose a branching literal, QUBE(TO) stores the literals in a priority queue according to (i) the prefix level of the corresponding atom, (ii) the score and (iii) the numeric ID. Initially the score of each literal is set to the value of the associated counter. Periodically, QUBE(TO) rearranges the priority queue by updating the score of each literal l: this is

[4]It can be objected that once x_2 and x_3 are assigned, so it is also y_1. This is due to the extreme simplicity of our example. It is relatively easy to build a more complex one, with more variables and clauses, in which x_2 and x_3 will be assigned as unit.

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. In QuBE(PO), the ordering of the priority queue cannot be maintained using the same set of conditions $(i - iii)$ above. However, the condition that top-priority literals in the queue must correspond to top atoms in the current QBF can be enforced by modifying the score as follows. First, we consider the set S of *bottom* atoms, i.e., all the atoms $|l|$ such that there is no $|l'|$ where $|l| \prec |l'|$ in the prefix, and we assign them the basic score. Then, we consider the set S' of all the atoms $|l|$ such that $|l| \prec |l'|$ precisely when $|l'| \in S$, i.e., $|l|$ is bottom: for each such literal l, we add to the basic score the maximum score among the literals l' such that $|l| \prec |l'|$. We repeat the process, each time by letting $S = S'$ and computing the new S' as above. In this way, we guarantee that any two literals l, l' that are incomparable (i.e., such that $|l| \not\prec |l'|$ and $|l'| \not\prec |l|$) are selected according to their heuristic scores, at the same time respecting the condition that each branching literal l corresponds to a top atom $|l|$.

The other essential modification has been the implementation of a data structure allowing for efficiently checking whether two atoms z and z' are in partial order. This check is indeed at the basis of the unit detection procedure.

6. Experimental Analysis

To evaluate the effectiveness of QuBE(PO) vs QuBE(TO), we first considered the same benchmarks used in [6]. These QBFs are appealing since they can be automatically generated not in prenex form, and/or in prenex form according to the 4 different optimal strategies defined in [6] and denoted with $\exists\uparrow\forall\uparrow$, $\exists\downarrow\forall\downarrow$, $\exists\downarrow\forall\uparrow$, $\exists\uparrow\forall\downarrow$. The generator takes four parameters ⟨DEP, VAR, CLS, LPC⟩ which have been set as follows: DEP is fixed to 6; VAR is varied in $\{4, 8, 16\}$; CLS is varied in such a way to have the ratio CLS/VAR in $\{1, 2, 3, 4, 5\}$; LPC is varied in $\{3, 4, 5\}$. For each setting of ⟨DEP, VAR, CLS, LPC⟩ we have generated 100 problems, and for each problem we obtained 4 different prenex QBF and one non prenex QBF. Finally we have run QuBE(TO) and QuBE(PO) on the prenex and non prenex instances respectively, on a farm of 10 identical rack-mount PCs, each one equipped with a 3.2Ghz PIV processor, 1GB of main memory and running Debian/GNU Linux. The timeout after which a solver is stopped has been set to 600s.

On all these instances QuBE(PO) compares very well with respect to QuBE(TO), especially if considering the non trivial instances (i.e., with a running time $\geq$ 0.1s). The first 4 rows of Table 1 gives a summary of the results. In the table,

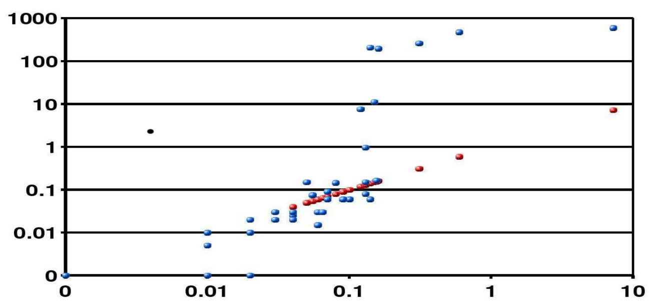
	$>$	$<$	$=$	$\gg$	$\ll$	$\bowtie$	$>10\times$	$10\times<$
$\exists\uparrow\forall\uparrow$	746	7	5247	370	1	1323	587	1
$\exists\downarrow\forall\downarrow$	1061	0	4939	441	0	1324	847	0
$\exists\downarrow\forall\uparrow$	1001	0	4999	425	0	1324	758	0
$\exists\uparrow\forall\downarrow$	999	0	5001	425	0	1324	757	0
$\exists\uparrow\forall\uparrow$	627	208	70	68	43	44	190	0

Table 1. QuBE(TO) **vs** QuBE(PO)

- "$>$" (resp. "$<$") is the number of instances for which QuBE(TO) is slower (resp. faster) than QuBE(PO) of more than 1s;
- "$=$" is the number of instances for which QuBE(TO) is within 1s from QuBE(PO);
- "$\gg$" (resp. "$\ll$") is the number of instances for which QuBE(TO) (resp. QuBE(PO)) times out while QuBE(PO) (resp. QuBE(TO)) does not;
- "$\bowtie$" is the number of instances for which both QuBE(TO) and QuBE(PO)) exceed the timeout;
- "$>10\times$" (resp. "$10\times<$") is the number of instances which are solved by both systems, but for which QuBE(TO) is at least 1 order of magnitude slower (resp. faster) than QuBE(PO).

As it can be seen, QuBE(PO) outperforms QuBE(TO) no matter which prenexing strategy is used. To further highlights QuBE(PO) good performances, figure 3 left shows the comparison between QuBE(PO) vs QuBE(TO) when considering the best prenexing strategy for that instance (in other words, for each problem we consider the minimum of the QuBE(TO) running times when using the 4 different prenexing strategies). In the plot in Figure 3 left, each solid-fill square dot represents a setting of the parameters, QuBE(PO) median solving time is on the x-axis (log scale), while QuBE(TO) median solving time, calculated as above specified, is on the y-axis (log scale). The diagonal (outlined diamond boxes) represents the solving time of QuBE(PO) against itself and serves as reference: the dots above the diagonal are settings where QuBE(PO) performs better than QuBE(TO), while the dots below are the settings where QuBE(PO) is worse than QuBE(TO). Even in such disadvantageous scenario, QuBE(PO) is competitive with QuBE(TO): QuBE(TO) median time exceeds the timeout for some setting of the parameters while this is never the case for QuBE(PO).

We also considered 905 formal verification problems coming from the application described in [8], where QBF reasoning is applied to model checking of early requirements. Each problem corresponds to a non prenex QBF. As before, the non prenex QBF has been converted to a prenex one using our implementation of the optimal prenexing strategy $\exists\uparrow\forall\uparrow$which, according to the results in the first four rows of Table 1, gives the best performances. The results are summarized in the last row in Table 1 and in the



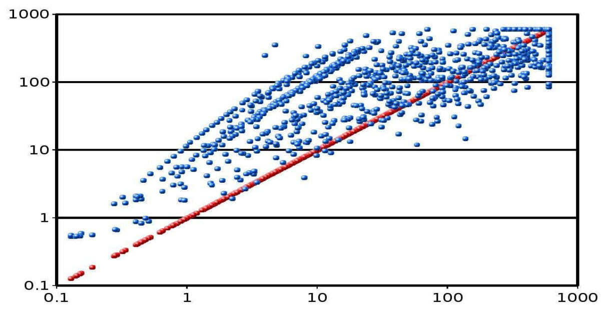


Figure 3. QUBE(TO) **vs** QUBE(PO)**. On the y(x) axis there are the** QUBE(TO)**(**QUBE(PO)**) times**

plot in Figure 3 right. As it can be seen, the results are very positive also in this case, even though not as impressive as before: this is due to the particular structure of these instances, which feature a few universal variables and a small number of alternations. Still, QUBE(PO) performs better than QUBE(TO) of more than one order of magnitude on 258 problems, compared to the 43 where the opposite happens (counting the instances solved by only one system).

7. Conclusions and related work

The main points of the paper can be summarized as:

- The basic search algorithm of QBF solvers can be extended to take into account the quantifier structure.
- The conversion of QBF instances exhibiting quantifier structure into prenex form can have dramatic impacts (i) on the effectiveness of the heuristic, and (ii) on the detection of unit literals.
- Our experiments reveal that by taking into account the quantifier structure we can get dramatic improvements in the performance of the QBF solver.

The work mostly related to ours is [2]. In this work, the author tries to re-construct the original non prenex structure of the formula starting from the instance in total order. A similar thing is also done in [4]. The essential difference between [2, 4] and our work is that the solver we use is based on search, while the solvers in [2] and [4] are mainly based on quantifier elimination. For solvers based on quantifier elimination, recovering or keeping the original quantifier structure is fundamental in order to reduce the size of each quantifier elimination operation. Notice that the solver SKIZZO described in [2] is not entirely based on quantifier elimination, since it uses different strategies –including search– for trying to solve each problem. However, search is the last attempted and thus the least used strategy, and it is not clear how SKIZZO uses the quantifier structure during the search. Finally, this is the first paper that we know of, clearly addressing the quantifier structure problem and giving clear evidence that keeping the original quantifier structure pays off, at least when using search based solvers.

References

[1] A. Ayari and D. Basin. Bounded model construction for monadic second-order logics. In *Proc. CAV'00*.

[2] M. Benedetti. Quantifier Trees for QBFs. In *Proc. SAT'05*.

[3] D. Le Berre, L. Simon, and A. Tacchella. Challenges in the QBF arena: the SAT'03 evaluation of QBF solvers. In *Proc. SAT'03*.

[4] A. Biere. Resolve and Expand. In *Proc. SAT'04*.

[5] M. Cadoli, A. Giovanardi, and M. Schaerf. An algorithm to evaluate quantified Boolean formulae. In *Proc. AAAI'98*.

[6] U. Egly, M. Seidl, H. Tompits, and M. Zolda. Comparing Different Prenexing Strategies for QBFs. In *Proc. SAT'03*.

[7] Z. Fu, Y. Yu, and S. Malik. Considering circuit observability don't cares in cnf satisfiability. In *Proc. DATE'05*.

[8] A. Fuxman, L. Liu, J. Mylopoulos, M. Pistore, M. Roveri, and P. Traverso. Specifying and analyzing early requirements in Tropos. *Requirements Engineering*, 9(2):132–150, 2004.

[9] E. Giunchiglia, M. Narizzano, and A. Tacchella. Learning for Quantified Boolean Logic Satisfiability. In *Proc. AAAI'02*.

[10] E. Giunchiglia, M. Narizzano, and A. Tacchella. Backjumping for Quantified Boolean Logic Satisfiability. *Artificial Intelligence*, 145:99–120, 2003.

[11] E. Giunchiglia, M. Narizzano, and A. Tacchella. QUBE: an Efficient QBF solver. In *Proc. FMCAD'04*.

[12] R. Letz. Lemma and model caching in decision procedures for QBFs. In *Proc. Tableaux'02*.

[13] M. Mneimneh and K. Sakallah. Computing Vertex Eccentricity in Exponentially Large Graphs: QBF Formulation and Solution. In *Proc. SAT'03*.

[14] C. H. Papadimitriou. *Computational Complexity*. Addison-Wesley, 1994.

[15] C. Scholl and B. Becker. Checking equivalence for partial implementations. In *Proc. DAC'01*.

[16] D. Sheridan. The optimality of a fast CNF conversion and its use with SAT. In *Proc. SAT'04*.

[17] L. Zhang and S. Malik. Conflict driven learning in a quantified Boolean satisfiability solver. In *Proc. ICCAD'02*.