Dec 8, 2008

A Solver for Quantified Formula Problem Q-ALL SAT

Anja Remshagen anja@ewestga.edu
Department of Computer Science,
University of West Georgia

Klaus Truemper klaus@utdallas.edu
Department of Computer Science,
University of Texas at Dallas

Abstract

Problem Q-ALL SAT demands solving a quantified Boolean formula that involves two propositional formulas in conjunctive normal form (CNF). When the first formula has no clauses and thus is trivial, Q-ALL SAT becomes the standard quantified Boolean formula (QBF) at the second level of the polynomial hierarchy. In general, Q-ALL SAT can be converted to second-level QBF by well-known transformations. A number of application problems can be formulated as instances of Q-ALL SAT. Thus, solution of the problem is of practical importance.

This paper describes a solution algorithm for Q-ALL SAT called QRSsat3. The method is a significant improvement over an algorithm called QRSsat that was described in an earlier paper. Algorithm QRSsat3 relies on backtracking search just as QRSsat does. The improvement over the predecessor is due to an enhanced learning process and a heuristic for the satisfiability problem SAT of CNF formulas.

Computational results are reported for three sets of instances including a robot problem and a game problem. To compare the performance of QRSsat3 with other solvers, we have converted the test instances into QBF format required by QBF solvers. For these test instances, QRSsat3 has uniformly low solution times and is substantially faster than QRSsat, which in turn was already much faster than state-of-the-art QBF solvers.

The problems SAT and Q-ALL SAT are part of a previously defined hierarchy of quantified formulas that we call constrained quantified formulas (CQFs). The paper includes some complexity results for the hierarchy of specially structured CQFs and thus for specially structured Q-ALL SAT cases.

Keywords: *learning, QBF, Q-ALL SAT*

1. Introduction

Define Q-ALL SAT to be the following problem. Given are disjoint variable sets Q, X, and Y, and two *conjunctive normal form* (CNF) formulas R and S. Formula R has variable set $Q \cup X$, and S has variable set $Q \cup Y$ as shown in Figure 1. We allow for the special case where R has no clauses and thus is *trivial*. Define a truth assignment to Q to be *R-acceptable* (resp. *R-unacceptable*) if the assignment can be extended to a satisfying solution of R. The analogous definition applies to the terms *S-acceptable* and *S-unacceptable*. Problem Q-ALL SAT demands the following. Either one determines that all R-acceptable truth assignments to Q are also S-acceptable, or one obtains an R-acceptable assignment to Q that is S-unacceptable. Thus, each instance of Q-ALL SAT requires evaluation of the

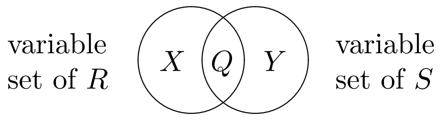


Figure 1. Variable sets of a Q-ALL SAT instance.

quantified formula

$$\forall Q (\exists X \; R \rightarrow \exists Y \; S) \tag{1}$$

Applications of Q-ALL SAT abound. Consider an expert system for medical diagnosis and treatment, as for example sketched by Moreland, Remshagen, and Riehl [18]. The system repeatedly asks for symptom values until the presence of a disease can be proven. Suppose that the disease is not present. Then this will be established only when all possible symptom questions have been asked and answered. Yet, one may be able to draw the same conclusion much earlier, by proving that, no matter which of the so-far-unexplored symptoms are present, one will be unable to establish the disease. The possible symptoms are represented by the variable set Q. The CNF formula R models the relationships among the symptoms. Formula S describes the relationships among the symptoms and diseases, and includes a clause that declares the disease of interest to be absent. Thus, unsatisfiability of S under a truth assignment to the Q variables proves presence of the disease. To decide whether a proof of the disease is possible, we find an R-acceptable and S-unacceptable assignment to Q, or we conclude that such an assignment does not exist.

The same problem occurs in other question-and-answer processes where a system queries a user for information until a desired conclusion can be established. Straach and Truemper [31] describe an expert system for regulatory compliance where Q-ALL SAT instances must be solved during a question-and-answer process, and where optimization versions must be solved for learning effective questioning. In that situation, the Q-ALL SAT instances typically are not so difficult, while the optimization versions are very hard. In ongoing work, we use the results reported here to solve these optimization versions.

Other applications arise in planning and games. For example, consider a game with two adversarial players, each of which attempts to reach some goal. We want to determine whether the first player can reach the goal while preventing the second player from reaching the goal. In this case, the truth assignments to the Q variables correspond to the moves of the first player. Formula R models all valid moves, and formula S models all scenarios in which the second player reaches the goal. That is, unsatisfiability of S under a truth assignment to the Q variables models a situation in which the second player cannot reach the goal. See Section 4 for details of such a game. Applications also arise outside Artificial Intelligence. For example, Mneimneh and Sakallah [17] compute the eccentricity of vertices, used in formal hardware verification, by evaluating a sequence of Q-ALL SAT instances. See also Eiter and Gottlob [9] for further applications.

A predecessor paper (Remshagen and Truemper [25]) proposes a direct attack on Q-ALL SAT based on backtracking search. The resulting solver, called QRSsat, outperformed three search-based QBF solvers on two problem classes.

However, on some test instances QRSsat still required up to almost an hour on a Sun ULTRA 5 (400 MHz) and thus was not yet suitable for practical applications. A significantly improved version called QRSsat3 can now solve that instance in less than two minutes on the same machine. This paper describes QRSsat3 that uses a substantially enhanced learning scheme for the computation of conflict clauses when satisfiability of the CNF formula S is detected. The scheme sharpens the learned clauses by a heuristic that detects satisfiability of CNF formulas, and by a limited enumeration of certain subcases. The heuristic takes advantage of the similarity of satisfying solutions for neighboring nodes of the search tree. For the first time, we have achieved uniformly low run times on all instances of the two problem classes described in the predecessor paper. We also have created additional, substantially harder, instances of the two problem classes to explore the limitations of QRSsat3. Here, too, QRSsat3 substantially outperforms QRSsat. The implementation of QRSsat3 employs the same data structures used for QRSsat. Indeed, the data structures are very simple, and the coding of subroutines is straightforward. Thus, the short run times are solely due to algorithmic efficiency. QRSsat3 is described in Section 3. The test problems are summarized in Section 4, and the test results are given in Section 5.

Q-ALL SAT isn't an isolated problem among quantified Boolean formulas. Chapter 4 of Truemper [33] introduces an entire problem hierarchy of quantified Boolean formulas that builds on SAT and Q-ALL SAT and that directly represents many applications. We call the formulas of the hierarchy *constrained quantified formulas (CQFs)*. Section 6 summarizes the CQF hierarchy and includes preliminary complexity results for specially structured CQFs, which thus imply certain complexity results for Q-ALL SAT.

2. Related Work

The predecessor paper includes a detailed discussion of prior work. We discuss more recent work in a terse summary. Most prior work related to Q-ALL SAT considers quantified Boolean formulas (QBFs) of the form (4), where all quantifiers precede a CNF formula.

Search-based QBF solvers that are based on a variation of the Davis-Putnam procedure DPLL include, for example, Quaffle by Zhang and Malik [35], [36], QuBERel1.3 by Giunchiglia, Narrizano, and Tacchella [12], [13], Semprop by Letz [16], and the solver by Rintanen [27]. In particular, Zhang and Malik [35] learn DNF clauses that are used to augment the CNF formula. Learning of these DNF clauses corresponds to learning of S-conflict clauses in our solver. However, we use a different sharpening process for these clauses.

Further work that improves search-based solvers includes binary clause reasoning by Samulowitz and Bacchus [30]. Using a backtracking search based SAT solver, the QBF solver by Samulowitz and Bacchus [29] can alleviate the variable ordering constraint imposed by quantifier nesting. The QBF solver profits from the clauses learned by the SAT solver and vice versa. Gent and Rowley [10] test two new techniques for solution learning that can improve the performance of QBF solvers.

We list additional recent work that explores alternative methods. Biere [7] uses Q-resolution and expansion on universally quantified variables. His solver Quantor eliminates variables starting at the innermost quantified variables until the problem can be solved by a SAT solver. The variables to be eliminated are chosen dynamically with the goal to keep the size of the formula as small as possible. The solvers QMRES and QBDD by Pan and Vardi [20] are based on ZDDs and OBDDs, respectively. GhasemZadeh, Klotz, and Meinel [11] introduce the solver QZSAT, which is also based on ZDDs. Ayari and Basin [3] reduce a quantified Boolean formula to a SAT instance and solve the resulting instance by a SAT solver. The reduction process uses mini-scoping, quantifier expansion, and non-clausal simplification techniques. The solver sKizzo by Benedetti [4] is an innovative hybrid approach that is based on symbolic skolemization and that adapts known techniques, like the DPLL procedure, SAT-based reasoning, and symbolic representations. Based on the Skolem theorem, sKizzo obtains a symbolic representation of a SAT instances that is equivalent to the original QBF instance. Using symbolic simplifications and decomposition into smaller problems, sKizzo solves the SAT instance.

A transformation from Q-ALL SAT to QBF is possible, but increases the size of the formula. Benedetti [5] acknowledges that a conversion into standardized QBF may result in a loss of structure. The paper introduces quantifier trees to recover at least parts of the original structure of an instance. Zhang [34] argues that the transformation to CNF limits the treatment of satisfaction clauses. His solver IQTest transforms the original Boolean formula into logically equivalent CNF and DNF formulas in order to handle conflicts and satisfactions symmetrically. Although there is some similarity between the treatment of the CNF and DNF formula and the treatment of R and S by our approach, his solver still relies on the equivalence on the CNF and DNF formula while R and S are typically not related. His solver is based on QBFs in prenex form.

Research on QBF with one quantifier alteration, called 2QBF, has been conducted by Ranjan, Tang, and Malik [22]. They compare the performance of QBF solvers and specialized 2QBF solvers on 2QBF instances. Their experiments indicate that specialized solvers for 2QBF can be more effective than general QBF solvers.

3. The Solution Algorithm

This section provides details of Algorithm QRSsat3.

Section 3.1 covers the basic scheme, which is based on backtracking search. This material is treated in detail in the predecessor paper, so we just include a summary.

Section 3.2 introduces a two-step approach to learn conflict clauses based on satisfiability of S. The steps employ a heuristic for Q-ALL SAT, a heuristic for the satisfiability problem SAT of CNF formulas, and a limited enumeration of certain subcases.

Section 3.3 gives details of the heuristic for SAT.

Section 3.4 describes the rule for selecting the next variable in the backtracking search.

3.1 Basic Algorithm

The backtracking algorithm Solve_Qallsat below is the basic scheme of QRSsat3. It enumerates all possible assignments to the Q variables. We denote the CNF formula resulting from a CNF formula T by assigning the truth value α to variable x by $T(x = \alpha)$. A variable to which a truth value has been assigned is also called *fixed*. A variable that has not been fixed is called *free*.

```
Solve_Qallsat($Q$,$R$,$S$)
(1)  Do unit-resolution in $R$ on all variables. All $Q$ variables fixed
     in $R$ by unit-resolution are fixed in $S$ as well.
(2)  Do unit-resolution in $S$ on the $Y$ variables only.
(3)  If ($Q = \emptyset$)
(4)      If ($R$ is satisfiable and $S$ is unsatisfiable)
(5)          Return False;
(6)      Else
(7)          Return True;
(8)      Endif
(9)  Endif
(10) Select a variable $q \in Q$;
(11) If (Solve_Qallsat($Q \setminus \{q\}, R(q = True), S(q = True)$) = False)
(12)     Return False;
(13) Else
(14)     If (Solve_Qallsat($Q \setminus \{q\}, R(q = False), S(q = False)$) = False)
(15)         Return False;
(16)     Else
(17)         Return True;
(18)     Endif
(19) Endif
```

In Steps (1) and (2), unit-resolution assigns *True* (resp. *False*) to a variable if it occurs nonnegated (resp. negated) as the single literal in a clause. Observe that, if unit-resolution is applied in R to all variables and in S to the Y variables only, then the reduced search space still contains all possible solution. However, unit-resolution in S on the Q variables is in general not permissible.

Any complete SAT solver may be used in Step (4). In our implementation, we have applied a simple solver based on backtracking search. The solver terminates as soon as all clauses are satisfied and thus may return a partial assignment in case of a satisfiable instance.

For the selection of a Q variable in Step (10), we use a version of the MOMS (Maximum Occurrences in Minimum Size clauses) heuristic which chooses a variable occurring in a maximum number of short clauses in R and in S combined. Details are discussed in Section 3.4.

The search tree is pruned if S is satisfied regardless of the values assigned to the free Q variables. For example, assume that the given Q-ALL SAT instance contains two Q variables q_1 and q_2, and that the search tree sets q_1 to *True* and q_2 to *False*. Further assume that the formula S is satisfiable under that assignment. Then the assignment for q_1 and q_2 cannot be a solution of the Q-ALL SAT instance, and hence every solution must set q_1 to *False* or q_2 to *True*. In other words, a solution must satisfy the clause $\neg q_1 \vee q_2$. Such a clause is called an *S-conflict clause*. If a conflict clause is detected, QRSsat3 tries to sharpen the clause by removing some literals in a two-step process. The sharpened clause is added to R.

Sharpening of clauses requires finding a minimal assignment to the Q variables that, when combined with every possible fixing of the unassigned Q variables, remains S-acceptable. This problem is related to finding a minimal satisfying solution starting from a given solution. However, since we want to minimize the assignment to a subset of the Q variables only, algorithms for the minimal satisfying solution problem, as discussed, for example, in Ravi and Somenzi [23], cannot be expected to perform well on the problem at hand. The next section describes the sharpening process.

3.2 Learning S-conflict Clauses

Why should we expect a sharpening of clauses by the removal of some literals to be possible? We answer the question using a typical example application.

Consider an expert system for medical diagnosis where symptoms are utilized to infer various diseases. The relationships are encoded in a CNF formula S'. The symptom variables make up a set Q. Relationships among the variables of Q are encoded in a CNF formula R. If we are interested in symptoms that prove a suspected disease d, we define $S = S' \wedge \neg d$ and solve the Q-ALL SAT instance given by Q, R, and S. Typically, some symptoms are closely related to disease d, while other symptoms are only tangentially or not at all related. Therefore, some subset of the symptoms is usually irrelevant to prove the suspected disease d. An S-conflict clause can be sharpened by removing the variables that correspond to these irrelevant symptoms. Appropriately, we call these variables *irrelevant*. The sharpened S-conflict clause can then be added to R as an additional restriction on the Q variables and hence reduces the search space.

We show how irrelevant variables may be detected in a two-step process. For the discussion, suppose that S remains satisfiable when all Q variables have been fixed. Let I_+ (resp. I_-) be the index set of the $q_i \in Q$ that have been fixed to *True* (*False*). Since these values do not constitute a solution for the Q-ALL SAT instance, we have the S-conflict clause $(\bigvee_{i \in I+} \neg q_i) \vee (\bigvee_{i \in I_-} q_i)$.

In the first step, each Q variable q_i is processed as follows. The variable q_i is deleted from the current S'. If the reduced formula is unsatisfiable, the variable q_i is added back again. Otherwise, S' denotes the reduced formula, and the literal of q_i in the S-conflict clause is removed from that clause.

The order in which the variables q_i are processed, does matter. We argue that it is best to process first the variables that were fixed last in the search tree. Indeed, the selection rules of variables in the tree search as described in the subsequent Section 3.3 assure that the likelihood of relevance of variables decreases as we proceed deeper into the tree. Thus, processing the most recently fixed variables first means that we try to remove variables first that most likely are irrelevant.

We carry out the first step for an example. Let the CNF formula S have the following clauses.

$$\begin{aligned}
&\neg q_1 \vee q_2 \vee \neg y_1\\
&q_1 \vee \neg y_2\\
&q_1 \vee \neg q_2 \vee y_2\\
&\neg q_3 \vee y_3\\
&q_4 \vee y_3\\
&\neg q_3 \vee q_4 \vee \neg y_3\\
&q_3 \vee \neg q_4 \vee \neg y_3
\end{aligned}$$

We have $Q = \{q_1, q_2, q_3, q_4\}$. Suppose in the tree search we have fixed $q_1 = \mathit{True}$, $q_2 = \mathit{False}$, $q_3 = \mathit{False}$, and $q_4 = \mathit{True}$, in that order. That fixing leaves S satisfiable. Thus, we have the S-conflict clause $\neg q_1 \vee q_2 \vee q_3 \vee \neg q_4$.

We try to sharpen the S-conflict clause. We initialize S' as S. First, we remove q_4 tentatively from all clauses of S'. We obtain the clauses

$$\begin{aligned}
&\neg q_1 \vee q_2 \vee \neg y_1\\
&q_1 \vee \neg y_2\\
&q_1 \vee \neg q_2 \vee y_2\\
&\neg q_3 \vee y_3\\
&y_3\\
&\neg q_3 \vee \neg y_3\\
&q_3 \vee \neg y_3
\end{aligned}$$

Since the reduced formula is unsatisfiable under $q_1 = \mathit{True}$, $q_2 = \mathit{False}$, and $q_3 = \mathit{False}$, we add q_4 again to the clauses of S' and remove q_3. Similarly, the resulting formula is unsatisfiable, and we add q_3 again to S'. Now we remove q_2 from S', getting

$$\begin{aligned}
&\neg q_1 \vee \neg y_1\\
&q_1 \vee \neg y_2\\
&q_1 \vee y_2\\
&\neg q_3 \vee y_3\\
&q_4 \vee y_3\\
&\neg q_3 \vee q_4 \vee \neg y_3\\
&q_3 \vee \neg q_4 \vee \neg y_3
\end{aligned} \tag{2}$$

Since the reduced formula is satisfiable under the assignment $q_1 = \mathit{True}$, $q_3 = \mathit{False}$, and $q_4 = \mathit{True}$, we remove the literal q_2 from the S-conflict clause, thus getting the sharpened clause $\neg q_1 \vee q_3 \vee \neg q_4$. The formula S' remains unchanged and thus is given by (2). In the next step, we tentatively remove q_1 from S'. The resulting formula is unsatisfiable, and the S-conflict clause is not changed. This completes the processing by the first step. The resulting S-conflict clause is $\neg q_1 \vee q_3 \vee \neg q_4$.

The second step attempts to remove additional variables from the S-conflict clause obtained in the first step. Let the S-conflict clause have k literals. For $1 \leq i \leq k$, define Q_i to be the subset of Q containing the i most recently fixed variables for which literals occur in the S-conflict clause. According to the above discussion linking the order of fixing variables and the likelihood of relevance, the definition of Q_i guarantees that the i variables of Q_i are more likely to be irrelevant than the remaining variables of the S-conflict clause. For the above example, we have $Q_1 = \{q_4\}$, $Q_2 = \{q_3, q_4\}$, and $Q_3 = \{q_1, q_3, q_4\}$.

For $1 \leq i \leq k$, we check certain sufficient conditions discussed momentarily. If these conditions are satisfied, then deletion of the literals of the variables of Q_i from the S-conflict clause is justified. Let i^* be the largest index i for which the conditions are satisfied. Then we sharpen the S-conflict clause by removing the literals of the variables of Q_{i*}. At that point, the second step stops.

The sufficient conditions and the related check for a given index i are as follows. First, we delete from S all Q variables that were determined to be irrelevant in the first step. Let S' result. Second, in S', we fix all Q variables of the S-conflict clause that do not occur in Q_i, to the current assignment values of the tree search. Let S'' result. Third, we test whether, for each possible assignment to the variables of Q_i, S'' remains satisfiable. If this is the case, then all such assignments would lead to backtracking in the search tree, and we are justified to remove the variables of Q_i from the S-conflict clause.

Since the Q_i are nested, the assignment cases considered for index i can be viewed as a subset of the cases for $i + 1$. Hence, using $k^* = |Q_{i^*}|$, at most 2^{k^*+1} satisfiability problems derived from S must be solved. In our implementation, we limit the process to values of i that do not exceed some parameter i_{max}. So far, we have found that the value $i_{max} = 8$ produces attractive results, and we have not had the need to devise a heuristic that dynamically chooses or modifies i_{max}.

We carry out the second step for the earlier example problem. Recall that the first step produces the S-conflict clause $\neg q_1 \vee q_3 \vee \neg q_4$. We have $Q_1 = \{q_4\}$, $Q_2 = \{q_3, q_4\}$, and $Q_3 = \{q_1, q_3, q_4\}$.

Deletion of the Q variables that do not occur in the S-conflict clause, reduces S to the following formula S'.

$$\begin{aligned}
&\neg q_1 \vee \neg y_1 \\
&q_1 \vee \neg y_2 \\
&q_1 \vee y_2 \\
&\neg q_3 \vee y_3 \\
&q_4 \vee y_3 \\
&\neg q_3 \vee q_4 \vee \neg y_3 \\
&q_3 \vee \neg q_4 \vee \neg y_3
\end{aligned}$$

Since $Q_1 = \{q_4\}$, we fix $q_1 = True$ and $q_3 = False$ to obtain S''. Then we enumerate all cases for q_4. In the tree search, we already tried $q_4 =$*True*, so we only need to check the case $q_4 = False$. For that situation, S'' remains satisfiable.

Next, we process Q_2. Here, fixing $q_1 = True$ defines S'' from S', and the two assignments $q_3 = True$, $q_4 = True$ and $q_3 = True$, $q_4 = False$ have not yet been examined. It turns out that the assignment $q_3 = \textit{True}$, $q_4 = \textit{False}$ makes S'' unsatisfiable. Thus, we have $i^* = 1$. Accordingly, we remove from the S-conflict clause $\neg q_1 \vee q_3 \vee \neg q_4$ the single variable q_4 of Q_1 and get $\neg q_1 \vee q_3$ as final sharpened clause. That clause is added to R.

Evidently, the two sharpening steps may require solution of a substantial number of satisfiability problems, and the process may seem to be a considerable computational burden if the SAT instance S is hard. But the SAT instances that must be solved are derived from S by deleting or fixing Q variables, and the resulting CNF formulas tend to be rather easy even if the original instance is hard. In addition, we employ a simple but effective heuristic that frequently detects satisfiability without applying the actual SAT solver. This heuristic is described in the next section.

We also use the above process when just some of Q variables have been fixed to values so that these values plus some values for the Y variables satisfy all clauses of S. In that case, we define the initial S-conflict clause from the fixed Q variables as described above. However, before we carry out the two sharpening steps, we first delete from S all free Q variables.

3.3 Detecting Satisfiability

Backtracking and learning of S-conflict clauses is initiated whenever all clauses of S are satisfied under the current assignment for the fixed Q variables and the fixed Y variables. In this case, S is satisfiable regardless of any assignment to the free Q variables. The same conclusion applies if the fixed Q variables and the fixed Y variables satisfy just some clauses of S, and if the remaining clauses can be satisfied by suitable values for the free Y variables. We describe a heuristic that tries to detect the latter situation with little computational effort.

The heuristic is based on the following observation. The assignment to the Q variables changes only slightly within a few steps of the search process. Thus, an assignment to the Y variables that satisfies all clauses of S after fixing some Q variables in the tree search, may still satisfy all clauses after a slight modification of the assignment for the Q variables. Accordingly, we store the assignment to the Y variables whenever S is solved in Step (4) or when S is solved to learn S-conflict clauses. Note that the assignment possibly does not assign truth values to all Y variables.

We summarize the heuristic to determine whether S is satisfiable. Let α denote the assignment to the Y variables of the most recent SAT solution computed during the tree search or during learning of S-conflict clauses. Whenever the next Q variable is fixed in S in Algorithm Solve_Qallsat, we determine the CNF formula S' resulting from S when Q variables are fixed according to the current assignment and when Y variables are fixed according to assignment α. If there are some free Y variables, we invoke a greedy algorithm that fixes the free Y variables one by one to the truth value that satisfies as many clauses in S' as possible. If the assignment for Q variables, the assignment α for Y variables, and the assignment by the greedy algorithm together satisfy all clauses in S', then S is satisfiable under the current assignment to Q no matter how the free Q variables are fixed. Since only minor computational effort is required to track α and to apply the greedy algorithm, the heuristic is efficient.

3.4 Selecting the Next Variable

The selection rule for the next Q variable in Step (10) of Solve_Qallsat is motivated by the fact that Solve_Qallsat can backtrack if R is determined to be unsatisfiable or if S is shown to be satisfiable no matter how the free Q variables are fixed. Accordingly, the rule considers all clauses in R not yet satisfied, as well as a certain subset of the clauses of S, say S'. The latter subset consists of the clauses not satisfied by the most recently determined assignment α of the SAT heuristic, provided that the current node of the search tree is close to the node where that heuristic was most recently successful in deciding satisfiability. If the current node is not close to the latter node, then the subset S' is the entire set of clauses of S. We measure closeness indirectly, by deciding that the current node is not close if the SAT heuristic has not found a solution in 10 consecutive cases since the most recent successful application. We derived that measure via a small set of test instances that are not part of the benchmarks described later. In addition to these considerations, the selection rule borrows from the MOMS (Maximum Occurrences in Minimum Size clauses) heuristic that has been applied in backtracking-based SAT solvers, for instance by Böhm and Speckenmeyer [8].

Details of the rule are as follows. Let S' be the clause set defined above. For each Q variable q, we count the nonnegated occurrences of q in the current R and the negated occurrences in S' in all clauses of length i. Let this number be $p_q(i)$. Analogously, let $n_q(i)$ be the number of negated occurrences of q in all clauses of the current R of length i and nonnegated occurrences of q in S' in clauses of length i. Define a vector h_q whose ith element is

$$h_q(i) = p_q(i) + n_q(i) - \frac{1}{3} \cdot |p_q(i) - n_q(i)| \tag{3}$$

We select the variable q with the lexicographically largest vector h_q. Thus, variables are first compared with respect to $h_q(1)$, ties are broken using $h_q(2)$, and so on. The selected variable q is set to *True* if the sum of all $p_q(i)$ is larger than the sum of all $n_q(i)$; otherwise, it is set to *False*. The selected variable and truth value aim at inducing unsatisfiability in R and satisfiability in S.

We pause for a moment and compare the above rule with selection rules of SAT solvers. The currently most successful rules are versions of VSIDS (Variable State Independent Decaying Sum) introduced by Moskewicz, Madigan, Zhao, Zhang, and Malik [19]. These rules differ from the MOMS heuristic by giving preference to the most recently learned clauses. A very successful SAT solver by Goldberg and Novikov [14], called BerkMin, carries the idea of VSIDS so far that it only selects a variable that occurs in the most recently learned clause. The MOMS-based selection rule used here also considers learned clauses, since they are added to R and thus enter the computation of (3). But in contrast to VSIDS, the computation does not emphasize the recently learned clauses and instead considers all learned clauses and the original R and S clauses. That choice is based on extensive trials with many selection rules. For example, in one alternate version, QRSsat3 always selected a variable from the most recently learned clause as BerkMin does. Another alternate selection rules considered several most recently learned clauses (e.g. the 5 most recently learned ones) only. Both selection rules, as well as a number of variants tried by us, resulted in very poor performance on our test instances. A possible, intuitive, explanation for the poor performance is as follows. The learned clauses of SAT solvers are logic implications of the original formula that are directly related to the satisfiability problem to be solved and thus are able to guide the selection process. On the other hand, the S-conflict clauses computed here encode relationships derived about satisfiability of R and S that have a more complex link to the Q-ALL SAT problem. By themselves, these clauses do not contain enough information to guide the selection process.

In the experiments, we also explored rules that used particular subsets of clauses of R and S. For example, one such rule considered only the learned clauses in R and all clauses of S, using a normalization to account for the different cardinalities of the two clause sets. For instance, let $n^*(q)$ be the number of learned clauses that contain variable q, and let $n_S(q)$ be the number of clauses in S that contain q. Let the total number of learned clauses be n^*, and let the total number of clauses in S be n_S. Then the count for q is $n^*(q)/n^* + n_S(q)/n_S$. The variable with highest count was selected by this rule. In other experiments, the selection rule considered only clauses of S or only clauses of R. None of the tested rules resulted in the same uniformly low run times on our test instances as the above described rule. Thus, at present, it seems to be most effective to include all clauses of the current R and of S with equal weight in the selection rule. Of course, this does not mean that there isn't some better rule. But we do know that such a rule cannot be a VSIDS version or closely related rule, since all such cases produced poor performance.

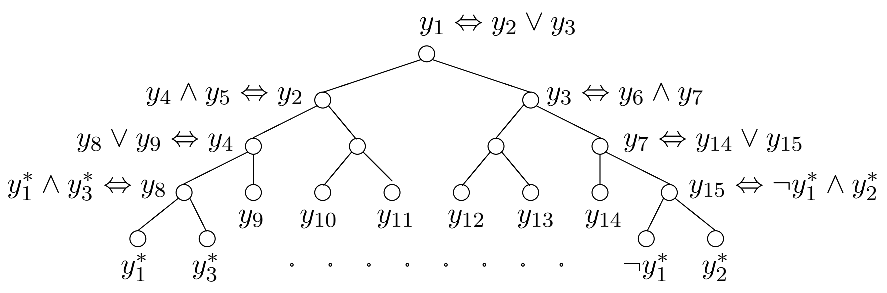
4. The Test Problems

We have tested QRSsat3 using five sets of benchmark problems. The first two sets were introduced in the predecessor paper. They represent structured problems that, by their design, are closer to real-world problems than randomly generated instances. For a detailed description of the problem sets, see the predecessor paper. A summary follows.

The first set models a robot navigation problem where a robot has to navigate through a grid with obstacles. The positions of most obstacles are not known. One has to decide whether the robot can reach its destination no matter how the obstacles are placed according to some given constraints. Specifically, each satisfying solution of the formula R models a path of the robot to the destination. The path is represented by the truth assignment for the Q variables. The formula S handles the aspect of obstacles indirectly, as follows. An assignment for the Q variables is S-unacceptable if it is impossible to place obstacles that interfere with the corresponding path of the robot. The instances differ by the grid size and by randomly defining the possible positions for obstacles.

The first test set contains a total of 18 instances. They have 64 to 100 Q variables, 64 to 100 X variables, and 128 to 200 Y variables. Formula R contains 924 to 1683 clauses, and formula S has 595 to 1125 clauses.

The second test set models the search for a successful strategy in a game. The first of two players tries to decide on certain choices that prevent the second player from reaching a goal. Formula S models the game in the form of an and/or tree. In the example tree of Figure 2, the y_i and y_j^* constitute the set Y of variables. Each internal node of the tree has one associated y_i variable that is linked to the variables of the children by the given relation. Each leaf node has one associated literal of a y_j^* variable. For clarity, Figure 2 shows the variables and relations just for a subset of the nodes. If y_1 is true, then the second



$y_1 \Leftrightarrow y_2 \vee y_3$
$y_4 \wedge y_5 \Leftrightarrow y_2$
$y_3 \Leftrightarrow y_6 \wedge y_7$
$y_8 \vee y_9 \Leftrightarrow y_4$
$y_7 \Leftrightarrow y_{14} \vee y_{15}$
$y_1^* \wedge y_3^* \Leftrightarrow y_8$
$y_{15} \Leftrightarrow \neg y_1^* \wedge y_2^*$
y_9
y_{10}
y_{11}
y_{12}
y_{13}
y_{14}
y_1^*
y_3^*
$\neg y_1^*$
y_2^*

Figure 2. And/or tree of a game instance

player has reached the goal. Toward this, the second player is allowed to set the values of the literals at the leaf nodes of the tree.

The first player wants to prevent the second player from reaching the goal. As a strategy, the first player is allowed to manipulate the values assigned to the y_i variables of some of the internal nodes of the tree, via the assignment of *True* to some Q variables. Such an assignment must be made to at least one Q variable. The formula R consists of just one clause, which enforces that condition. The set of X variables is empty.

In the instances of the test set, the leaf nodes are randomly assigned literals. The instances have 15 to 25 Q variables, no X variables, and 275 to 405 Y variables. Formula R has just one clause, and formula S has 781 to 841 clauses. The second test set contains 12 game sets, each containing 12 instances.

The third test set consists of ten randomly generated instances. It turned out that almost all randomly generated instances are *False* and very easy. To avoid this result, we forced the set to contain three *True* instances. The first five instances contain 20 Q variables, and the other five instances contain 25 Q variables. All instances have 50 X and 50 Y variables. The CNF formulas R and S of all instances have 200 clauses. Each clause contains 2 to 10 literals.

The fourth test set contain larger versions of the robot navigation problem of the first test set. The instances have 144 to 196 Q variables, 144 to 196 X variables, and 288 to 392 Y variables. The formula R has 2746 to 4169 clauses, and the formula S has 1899 to 2973 clauses.

The fifth test set has larger versions of the game problem of the second test set. The instances have 30 Q variables, no X variables, and 275 to 405 Y variables. The formula R has just one clause, and the formula S has 796 to 856 clauses.

For a comparison of our computational results with other solvers, we converted the instances of the first three test sets into the Q-DIMACS format required by QBF solvers; see QBFLIB [21] for details about that format. The transformation, described in detail in the predecessor paper, introduces a new variable for each clause in R. Alternatively, one could transform the negation of a Q-ALL SAT instance to the standard format, which results in a new variable for each clause in S. However, the resulting instances have two quantifier alterations instead of only one. Experiments apparently indicated that the instances resulting from the alternate formulation are more difficult for QBF solvers. Due to the transformation, the number of clauses may increase substantially. For example, the

Q-ALL SAT instance robot_8_1 of Table I has a total of $|Q|+|X|+|Y| = 256$ variables, formula R has 925 clauses, and formula S has 596 clauses. Thus, R and S together have 1521 clauses. The corresponding instance in the Q-DIMACS format has $256 + 925 + 1 = 1182$ variables and 3211 clauses. On the other hand, the increase in size for the game instances is insignificant since formula R consists of just one clause.

5. Computational Results

QRSsat3 has been implemented in the programming language C. The following simple data structures and subroutines are used. For each CNF formula, a list of its clauses is maintained. A clause is represented by a list of its literals. For each literal, a list of the clauses in which that literal occurs is stored. These lists are not dynamically modified as the CNF formula is reduced by variable fixing. That is, literals and clauses are not removed from the corresponding lists if they are fixed or satisfied, respectively. All lists and statistics are implemented by arrays. We use a simple SAT solver in Step (4) of Solve_Qallsat and when S-conflict clauses are learned. The SAT solver is based on chronological backtracking. That is, no backjumping or learning is utilized by that solver. As the implementation does not rely on sophisticated data structures and subroutines, the short run times reported later must be due to the efficiency of the algorithm.

We used the first three test sets to compare QRSsat3 with the following three search-based QBF solvers: yQuaffle by Yu and Malik [32], QuBERel1.3 by Giunchiglia, Narrizano, and Tacchella [12], [13], and Semprop by Letz [16]. These solvers were selected since they performed very well at the SAT 2004 QBF Evaluation. For details about these QBF solvers and their evaluation, see QBFLIB [21]. The computational results for the solvers yQuaffle, QuBERel1.3, Semprop, and QRSsat3 were obtained on a Sun ULTRA 5 (400 MHz) workstation. We also ran two more recent solvers: sKizzo by Benedetti [4] and Quantor by Biere [7], which uses PicoSAT as SAT solver. See Section 2 for details on sKizzo and Quantor. The experiments for sKizzo and Quantor were run on different platforms due to hardware and software constraints of the Sun ULTRA 5. Specifically, the tests for sKizzo were performed on a 3GHz Intel(R) Pentium(R) D CPU with 4GB RAM running Cygwin on Windows XP. The tests for Quantor were performed on a Sun Enterprise 250 with two UltraSparc II 400MHz processors and 2GB RAM running the Solaris 9 operating system. We have carried out some run time comparisons with QRSsat3 to gauge the speed differences of the two platforms compared with the Sun ULTRA 5. The Sun Enterprise 250 roughly had the same speed as the Sun ULTRA 5. But the Pentium processor was more than 6 times faster than the Sun ULTRA 5. This means that the execution times reported in the tables below for Quantor may be directly compared with the run times for yQuaffle, QuBERel1.3, Semprop, and QRSsat3. However, the run times reported in the tables for sKizzo should be scaled up if a reasonable comparison with the other run times is to be made. We purposely omit such scaling since it may be controversial. At any rate, we shall see that, even without scaling of run times, QRSsat3 is much faster than sKizzo.

Table I lists the results for the 18 robot instances of the first test set. The second and third column displays the run times of QRSsat3 and QRSsat, respectively, in seconds. The run times for yQuaffle, QuBERel1.3 are given in columns three, four, and five, respectively. The solution process was aborted if the run time exceeded 60 min. A run time of 0 indicates

Table 1. Results for the robot instances

instance	QRSsat3 time (sec)	QRSsat time (sec)	yQuaffle time (sec)	QuBE time (sec)	Semprop time (sec)
robot_8_1	0	0	39	26	>3600
robot_8_2	1	0	30	18	>3600
robot_8_3	0	1	41	49	2
robot_8_4	1	2	–*	41	>3600
robot_8_5	1	0	27	8	>3600
robot_8_6	0	1	29	38	>3600
robot_9_1	1	0	96	520	>3600
robot_9_2	0	1	58	122	>3600
robot_9_3	1	1	122	257	81
robot_9_4	1	3	73	129	>3600
robot_9_5	1	1	62	45	>3600
robot_9_6	1	2	–*	178	>3600
robot_10_1	0	0	179	252	>3600
robot_10_2	1	1	115	250	>3600
robot_10_3	1	1	1341	506	7
robot_10_4	3	8	–*	201	>3600
robot_10_5	1	2	–*	92	>3600
robot_10_6	2	4	130	499	>3600

* The solver terminated with an error.

a time below 0.1 sec. The results for sKizzo and Quantor are not listed in the table since both schemes could not solve any of the robot instances within the 60 min limit.

All robot instances were solved by QRSsat3 within 3 sec. For these test instances, the heuristic to detect satisfiability of S reduced the run time of QRSsat by a factor of about 2. The two-step process for sharpening S-conflict clauses did not reduce the run times further. Due to the structure of these test instances, QRSsat could already shorten each S-conflict clause to a unit-clause. Roughly speaking, QRSsat3 is twice as fast as the predecessor QRSsat on the robot instances.

The data for the three search-based QBF solvers can be summarized as follows. yQuaffle solved 14 instances within 27–1341 sec and terminated with an error on the other four instances. QuBERel1.3 solved each instance within 8–520 sec. With three exceptions, Semprop could not solve any instance within 60 min; the exceptions required 2–81 sec. Compared with each of the five QBF solvers, QRSsat3 is, on average, faster by a factor of at least 200.

Table II and III display the results for the 12 game problem sets of the second test set. Recall that each of the 12 problem sets contains 12 instances. The columns of Table II are arranged analogously to Table I. Similarly, Table III lists the results for Quantor and sKizzo. For easier comparison, the results for QRSsat3 are listed again. The clock speed of the corresponding platform is listed below each solver name to emphasize that different platforms were used.

Table 2. Results of QRSsat3, yQuaffle, QuBERel1.3, and Semprop for the game instances

	QRSsat3		QRSsat		yQuaffle		QuBE		Semprop	
problem set	time (sec)	t/o	time (sec)	t/o	time (sec)	t/o	time (sec)	t/o*	time (sec)	t/o
game_15_20	0.3	0	4	0	76	0	13	0	23	0
game_15_50	0.4	0	4	0	56	0	34	2	33	0
game_15_100	0.5	0	2	0	58	0	70	0	30	0
game_15_150	0.6	0	3	0	43	0	129	2	26	0
game_20_20	2.8	0	70	0	631	2	193	2	492	0
game_20_50	1.1	0	14	0	103	0	63	1	276	0
game_20_100	0.7	0	3	0	498	1	649	2	400	2
game_20_150	0.9	0	3	0	132	0	823	2	186	0
game_25_20	11.4	0	91	1	1500	5	1374	4	1500	5
game_25_50	1.8	0	37	0	900	3	643	2	1140	3
game_25_100	1.0	0	4	0	812	2	1204	4	1403	4
game_25_150	1.0	0	3	0	389	1	1098	3	625	2

* The number of instances that were not solved within 60 min includes 10 cases in which QuBERel1.3 terminated with an error.

Due to the large number of test instances, we list the average run time for each of the 12 problem sets. After each column with the average run times, the next column labelled t/o displays how many instances could not be solved within 60 min. In the calculation of the average run times, we assume a run time of 60 min for all instances that are not solved within the time limit of 60 min on the corresponding platform. The cases resulting in an error were disregarded in the average run times.

QRSsat3 solves 95% of the game instances within 2 sec. The highest solution time of QRSsat3 for any one of the instances is 81 sec. In comparison, the original QRSsat solved only 56% of the game instances within 2 sec and required more than 53 min in one case. Overall, QRSsat3 is more than 10 times faster than QRSsat. Both the more elaborate learning scheme for S-conflict clauses and the heuristic to detect satisfiability improve the run times by a factor of about 3 to 4 each.

The time limit of 60 min was exceeded by the solver yQuaffle in 14 cases, by QuBE in 14 cases (10 cases resulted in an error), by Semprop in 16 cases, by sKizzo in 16 cases, and by Quantor 16 cases. In general, all QBF solvers needed significantly more run time on *True* instances. For example, four instances in problem set game_25_20 evaluate to *True*. All five QBF solvers exceeded 60 min on these four instances, except for QuBERel1.3, which could solve one of these instances in about 34 min. QRSsat solved three of the four instances within 76 sec and one in over 53 min. In contrast, QRSsat3 solved one instance in 3 sec, two in 35 sec, and one in 81 sec. However, on some *True* instances that were very hard for the three search-based solvers sKizzo and Quantor performed very well. For example, all three search-based QBF solvers exceeded 60 min on the four *True* instances in set game_25_100, except for yQuaffle, which could solve two instances in about 134 sec and

Table 3. Results of QRSsat3, sKizzo, and Quantor for the game instances

	QRSsat3 (400MHz)		sKizzo (3GHz)*		Quantor (2x400MHz)**	
problem set	time (sec)	t/o	time (sec)	t/o	time (sec)	t/o
game_15_20	0.3	0	237.7	0	188.1	0
game_15_50	0.4	0	903.4	1	193.7	0
game_15_100	0.5	0	1.0	0	0.1	0
game_15_150	0.6	0	0.6	0	0.1	0
game_20_20	2.8	0	1500.7	5	150.2	5
game_20_50	1.1	0	601.1	2	687.9	2
game_20_100	0.7	0	1.1	0	300.1	1
game_20_150	0.9	0	0.6	0	0.0	0
game_25_20	11.4	0	1500.8	5	1500.1	5
game_25_50	1.8	0	901.0	3	900.3	3
game_25_100	1.0	0	1.6	0	0.1	0
game_25_150	1.0	0	0.6	0	0.1	0

* Processor is about 6 times faster than the one used for QRSsat3.
** Processor has about the same speed as the one used for QRSsat3.

40 min, respectively. sKizzo solved each instance in less than 2 sec and Quantor solved each instance within 0.1 sec. The predecessor QRSsat solved each of the three instances within 8 sec, while QRSsat3 solved each instance within 1 sec.

The comparison between QRSsat3 and the other five QBF solvers is complicated by the fact that we do not know how long the cases that were stopped after 60 min, would take. But if one assigns a solution time of 60 min for each such case, and if one ignores the higher speed of the Pentium processor used for sKizzo, then QRSsat3 on average is faster than each of the five QBF solvers by a factor of at least 200.

Table IV contains the results for the 10 random instances of the third test set. The results for sKizzo and Quantor are not listed since both solvers could not complete any random instance within 60 min. QRSsat3 solved one instance in 1 sec and all other instances within 0.1 sec. QRSsat3 solved two instances in 1 sec and the other instances within 0.1 sec. Of the three QBF solvers, yQuaffle performed best, requiring at most 13 sec. The other two solvers had times ranging from less than 0.1 sec to cases where 60 min is exceeded. When one compares the results for the random instances with those for the robot and game instances, it becomes clear that random instances are not challenging and therefore are not good test cases for quantified formulas of the type considered here.

We turn to the fourth and fifth test sets, which contain harder versions of the instances of the first and second test sets. We used these test sets to explore the limitations of QRSsat3 and to obtain additional comparison data for that solver and the original QRSsat. We did not consider it fruitful to apply the other five QBF solvers to these test sets since QRSsat3

Table 4. Results for the random instances of the third test set

	QRSsat3	QRSsat	yQuaffle	QuBERel1.3	Semprop
instance	time (sec)	time (sec)	time (sec)	time (sec)	time (sec)
random_20_1	1	1	2	1	>3600
random_20_2	0	0	6	1082	379
random_20_3	0	0	1	0	0
random_20_4	0	0	1	1	8
random_20_5	0	0	10	678	>3600
random_25_1	0	0	0	0	0
random_25_2	0	0	0	0	0
random_25_3	0	1	13	>3600	>3600
random_25_4	0	0	0	0	0
random_25_5	0	0	5	0	1

Table 5. QRSsat3 and QRSsat on robot instances of the fourth test set

	QRSsat3	QRSsat
instance	time (sec)	time (sec)
robot_12_1	1	1
robot_12_2	2	5
robot_12_4	14	50
robot_12_6	6	18
robot_14_1	2	3
robot_14_2	5	14
robot_14_4	55	207
robot_14_6	16	59

already outperforms them on the easier instances of the first two test sets. Tables V and VI contain the results for the fourth and fifth test set.

According to Table V, QRSsat3 performs better than QRSsat on the robot instances of the fourth test set by a factor of about 3. In Table VI, every row provides the average time over a set of 12 game instances of the fifth test set. Here, QRSsat3 reaches its limits on some instances since the highest run time, 2144 sec, is no longer practical. QRSsat3 outperforms QRSsat on all instances except for one of the 48 instances, where QRSsat is marginally faster with 162 sec compared with 170 sec for QRSsat3. On average, QRSsat3 is faster by a factor of about 8 on these problem sets. More importantly, QRSsat3 outperforms QRSsat by a factor of about 100 on the hardest instances. For example, QRSsat3 has the run times 13 sec, 8 sec, and 0.1 sec on three instances in game_30_50 while QRSsat needs 1039 sec, 1174 sec, and 538 sec, respectively.

Table 6. QRSsat3 and QRSsat on game instances of the fifth test set

	QRSsat3		QRSsat	
problem	time (sec)	t/o	time (sec)	t/o
game_30_20	235.5	0	1710.2	4
game_30_50	2.3	0	222.3	0
game_30_100	1.1	0	8.3	0
game_30_150	1.1	0	4.9	0

6. A Hierarchy of Quantified Boolean Formulas

At present, the standard QBF format

$$\forall X_1 \exists X_2 \forall X_3 \ldots \exists X_n S \tag{4}$$

is well accepted, where S is a propositional logic formula in conjunctive normal form with variable set $X_1 \cup X_2 \cup \cdots \cup X_n$. While it may be theoretically convenient to force each problem into the QBF format, it may be an unwise choice to encode all applications that way. Indeed, for many applications, the underlying knowledge is composed of several components, each of which can be represented by a separate propositional logic formula. A natural representation then combines these formulas using quantification, implications, conjunctions, and disjunctions as required. Such a formulation often is not in the format of QBF, and transformation into the QBF format frequently hides or even destroys problem structure that could and should be exploited. These considerations have been voiced elsewhere as well. See, for instance, Ansotegui, Gomes, and Selman [1]. For an example, consider an ad hoc network management problem at the third level of the polynomial hierarchy. The problem demands that we find a possible network configuration such that, no matter how the environment changes, the network is able to adapt accordingly. Here, a Boolean formula R_2 models feasible network configurations, a second formula R_1 represents possible changes in the environment, and a formula S models suitable responses to changes. The resulting formulation is

$$\exists Q_2 \,(\exists X_2 \, R_2 \wedge \forall Q_1 \,(\exists X_1 \, R_1 \;\rightarrow\; \exists Y \, S)) \tag{5}$$

All formulas share the variable set Q_2 representing the network configurations. R_1 and S share the variable set Q_1 that represents the changes in the environment.

Chapter 4 of Truemper [33] introduces SAT, Q-ALL SAT and (5) as the first three levels of an entire hierarchy. An exercise, with appropriate hint, asks the reader to define the higher levels of the hierarchy. We call the formulas of that hierarchy *constrained quantified formulas (CQFs)*. The reference also includes several optimization versions motivated by applications. For the description and illustration of the CQF hierarchy, we consider a game where we want to decide on the next move looking several moves ahead. Every additional move we look ahead adds one layer of complexity. Suppose that the moves in the ith step are constrained by a CNF formula R_i, and that the relationships between moves and winning states are represented by a CNF formula S. The possible moves are given by the R_i-acceptable assignments to a subset Q_i of the variables of R_i. In addition, R_i contains variable set X_i, and S contains variable set Y. Then the resulting hierarchy of CQFs is defined as follows.

$$\begin{aligned} F_1 &= \exists Y S \\ F_i &= \begin{cases} \forall Q_i((\exists X_i R_i) \rightarrow F_{i-1}) & \text{if } i > 1, i \text{ even} \\ \exists Q_i((\exists X_i R_i) \wedge F_{i-1}) & \text{if } i > 1, i \text{ odd} \end{cases} \end{aligned} \tag{6}$$

In general, each CNF formula in F_i contains the variable set Q_i. Note that Q-ALL SAT is the case $i = 2$ of (6), and formula (5) is the case $i = 3$.

In subsequent work, Benedetti, Lallouet, and Vautard [6] use the same hierarchy structure for the constraint satisfaction problem and then, apparently not aware of the above reference, propose the above formulation as an alternative to QBF. Sabharwal, Ansotegui, Gomes, Hart, and Selman [28] motivate an alternative formulation for QBF using the example of a game. In their formulation, all quantifiers precede one Boolean formula that is composed of CNF and DNF formulas.

In ongoing work, we are investigating the complexity of the CQF hierarchy. We cite results relevant here. It is easy to prove that, in the general case, evaluating formula F_i is at the ith level of the polynomial hierarchy. In particular, F_i is Π_i^{P}-complete if i is even and Σ_i^{P}-complete if i is odd. For special cases of practical interest, a reduced complexity can be proved. We cite two cases for CQFs with CNF *Horn* formulas, where each clause in a CNF formula contains at most one nonnegated variable, or with *2CNF* formulas where each clause contains at most two literals. Specifically, if all CNF formulas R_i and the formula S are Horn (resp. 2CNF), then the complexity of evaluating F_i is reduced by one level (resp. two levels) when compared with the general case. This result implies that Q-ALL SAT with R and S Horn (resp. 2CNF) is $\mathcal{NP}$-complete (resp. in $\mathcal{P}$). For proofs, see Remshagen [24]. For complexity results concerning optimization versions of Q-ALL SAT, see Remshagen and Truemper [26].

It is instructive to compare the above results with corresponding prior results for QBF. Specifically, Aspvall, Plass, and Tarjan [2] prove that QBF is in $\mathcal{P}$ in case of 2CNF. Kleine Büning, Karpinski, and Flögel [15] show that QBF is also reduced to $\mathcal{P}$ in case of Horn formulas. Evidently, these special cases do not capture the sharply rising difficulty of the polynomial hierarchy, or rather, the assumptions completely eliminate that aspect. In contrast, the above results for the CQF hierarchy show that Horn and 2CNF formulas have a rather limited effect in reducing the computational complexity.

7. Summary and Future Work

This paper describes algorithm QRSsat3 for problem Q-ALL SAT, where each instance is a quantified formula involving two propositional CNF formulas. The solver is a significant enhancement of the predecessor QRSsat. Computational results on the instances of two problem classes that partially were difficult for the predecessor QRSsat, are uniformly low for QRSsat3. Indeed, while QRSsat required up to 53 min on a Sun ULTRA 5 for some instances, QRSsat3 can solve all instances within 81 sec. In addition, 95% of the test instances are now solved within 2 sec. But on two additional classes with much harder instances, the limit of QRSsat3 is sometimes reached and run times, though much smaller than for QRSsat, grow to several minutes for some instances.

When compared with the performance of three search-based and two other state-of-the-art QBF solvers on QBF versions of the problem classes, the direct attack on Q-ALL SAT with the solver QRSsat3 is substantially faster. Indeed, for two test sets representing structured problems, QRSsat3 is faster than each of the five QBF solvers by a factor of at least 200.

The paper also includes some results for the complexity of the CQF hierarchy, including specially structured cases, which imply complexity results for Q-ALL SAT. In ongoing work, we aim for exact solution algorithms that exploit the special structure, as well as for methods handling optimization versions motivated by applications.

References

[1] C. Ansotegui, C.P. Gomes, and B. Selman The Achilles' Heel of QBF. *Proceedings of the Twentieth National Conference on Artificial Intelligence* (2005) 275–281.

[2] B. Aspvall, M.F. Plass, and E. Tarjan, A linear-time algorithm for testing the truth of certain quantified Boolean formulas. *Information Processing Letters* **8** (1979), no. 3, 121–123.

[3] A. Ayari and D. Basin, QUBOS: Deciding Quantified Boolean Logic Using Propositional Satisfiability Solvers. *Proceedings of the Fourth International Conference on Formal Methods in Computer-Aided Design* (2002).

[4] M. Benedetti, Evaluating QBFs via Symbolic Skolemization. *Proceedings of the Eleventh International Conference on Logic for Programming Artificial Intelligence and Reasoning* (2006).

[5] M. Benedetti, Quantifier Trees for QBFs. *Proceedings of the Eighth International Conference on Theory and Applications of Satisfiability Testing* (2005), 378–385.

[6] M. Benedetti, A. Lallouet, and J. Vautard, QCSP made Practical by Virtue of Restricted Quantification. *Proceedings of the 20th International Joint Conference on Artificial Intelligence* (2007).

[7] A. Biere, Resolve and Expand. Proceedings of the Seventh International Conference on Theory and Applications of Satisfiability Testing (2004), 238–246.

[8] M. Böhm and E. Speckenmeyer, A fast parallel SAT-solver – efficient workload balancing. *Annals of Mathematics and Artificial Intelligence* **17** (1996), no. 2, 381–400.

[9] T. Eiter and G. Gottlob, The Complexity of Logic-based Abduction. *Journal of the Association for Computing Machinery* **42** (1995), 3–42.

[10] I.P. Gent and A.G.D. Rowley, Local and Global Complete Solution Learning Methods for QBF. *Proceedings of the Eighth International Conference on Theory and Applications of Satisfiability Testing* (2005) 91–105.

[11] M. GhasemZadeh, V. Klotz, and C. Meinel, Embedding Memorization to the Semantic Tree Search for QBF. *Proceedings of the 17th Australian Joint Conference on Artificial Intelligence* (2004).

[12] E. Giunchiglia, M. Narrizano, and A. Tacchella, Learning for Quantified Boolean Logic Satisfiability. *Proceedings of the 18th National Conference on Artificial Intelligence* (2002).

[13] E. Giunchiglia, M. Narrizano, and A. Tacchella, Backjumping for Quantified Boolean Logic satisfiability. *Artificial Intelligence* **145** (2003), no. 1, 99–120.

[14] E. Goldberg and Y. Novikov, BerkMin: a Fast and Robust Sat-Solver. *Design Automation and Test in Europe* (2002).

[15] H. Kleine Büning, M. Karpinski, and A. Flögel, Resolution for quantified Boolean formulas. *Information and Computation* **117** (1995), no. 1, 12–18.

[16] R. Letz, Lemma and Model Caching in Decision Procedures for Quantified Boolean Formulas. in *Automated Reasoning with Analytic Tableaux and Related Methods*, Springer-Verlag, *Lecture Notes in Comput. Sci.* **2381** (2002), 160–175.

[17] M. Mneimneh and K. Sakallah, Computing Vertex Eccentricity in Exponentially Large Graphs: QBF Formulation and Solution. *Proceedings of the Sixth International Conference on Theory and Applications of Satisfiability Testing* (2003).

[18] K. Moreland, A. Remshagen, and K. Riehl, An Intelligent System for Medical Diagnosis. *Grace Hopper Celebration of Women in Computing* (2006).

[19] M. Moskewicz, C. Madigan, Y. Zhao, L. Zhang, and S. Malik, Chaff: Engineering an Efficient SAT Solver. *Proceedings of the 39th Design Automation Conference* (2001).

[20] G. Pan and Y. Vardi, Symbolic Decision Procedures for QBF. *Proceedings of the Tenth International Conference on Principles and Practice of Constraint Programming* (2004).

[21] E. Giunchiglia, M. Narizzano, and A. Tacchella Quantified Boolean Formulas satisfiability library (QBFLIB). http://www.qbflib.org (2001).

[22] D. Ranjan, D. Tang, and S. Malik, A Comparative Study of 2QBF Algorithms. *Proceedings of the Seventh International Conference on Theory and Applications of Satisfiability Testing* (2004).

[23] K. Ravi and F. Somenzi, Minimal Assignments for Bounded Model Checking. *Proceedings of the Tenth International Conference on Tools and Algorithms for the Construction and Analysis of Systems* (2004), 31–45.

[24] A. Remshagen, On the Complexity of the CQF Hierarchy. Working paper (2007).

[25] A. Remshagen and K. Truemper, An Effective Algorithm for the Futile Questioning Problem. *Journal of Automated Reasoning* **34** (2005), 31–47.

[26] A. Remshagen, and K. Truemper, The Complexity of Futile Questioning. *Proceedings of the International Conference on Foundations in Computer Science* (2007), 132–138.

[27] J. Rintanen, Constructing conditional plans by a theorem prover. *Journal of Artificial Intelligence* **10** (1999), 323–352.

[28] A. Sabharwal, C. Ansotegui, C.G. Gomes, J.W. Hart, and B. Selman, QBF Modeling: Exploiting Player Symmetry for Simplicity and Efficiency. *Proceedings of the Ninth International Conference on Theory and Applications of Satisfiability Testing* (2006).

[29] H. Samulowitz and F. Bacchus, Using SAT in QBF. *Proceedings of the Eleventh International Conference on Principles and Practice of Constraint Programming* (2005).

[30] H. Samulowitz and F. Bacchus, Binary Clause Reasoning. *Proceedings of the Ninth International Conference on Theory and Applications of Satisfiability Testing* (2006).

[31] J. Straach and K. Truemper, Learning to Ask Relevant Questions. *Artificial Intelligence* **111** (1999), 301–327.

[32] Y. Yu and S. Malik, Solver Description for yQuaffle. *Proceedings of the Seventh International Conference on Theory and Applications of Satisfiability Testing Addendum* (2004).

[33] K. Truemper, Design of Logic-based Intelligent Systems. Wiley (2004).

[34] L. Zhang, Solving QBF with Combined Conjunctive and Disjunctive Normal Form. *Twenty-First National Conference on Artificial Intelligence* (2006).

[35] L. Zhang and S. Malik, Towards a Symmetric Treatment of Satisfaction and Conflicts in Quantified Boolean Formula Evaluation. *Proceedings of the 8th International Conference on Principles and Practice of Constraint Programming* (2002).

[36] L. Zhang and S. Malik, Conflict Driven Learning in a Quantified Boolean Satisfiability Solver. *Proceedings of the International Conference on Computer Aided Design* (2002).