

An Effective QBF Solver for Planning Problems

Charles Otwell¹, Anja Remshagen¹, and Klaus Truemper²

¹ State University of West Georgia, Carrollton GA 30118, USA,
anja@westga.edu

² University of Texas at Dallas, Box 830688, Richardson, TX 75083-0688, USA,
klaus@utdallas.edu

Abstract. A large number of applications can be represented by quantified Boolean formulas (QBF). Although evaluating QBF is NP-hard and thus very difficult, there has been significant progress in the development of QBF solvers. These solvers require the quantified Boolean formula to be in a standard format. We have encountered a large class of problems whose representation as QBF is not in that standard format. If we apply current state-of-the-art QBF solvers, the required transformation into standard format increases the size of the formula and tends to hide structural properties of the problem class. We suggest a direct attack of the problem. The solution algorithm is based on backtracking search and on a new form of learning clauses. We have tested a first implementation of the algorithm on a class of planning problems. The tests show that the approach is significantly faster than current state-of-the-art QBF solvers.

1 Introduction

We have encountered the following planning problem arising in a game. In the game, two adversarial players try to reach a destination position while preventing the opponent from reaching his destination. We want to determine a plan for one of the players, say, for the first player, so that he can reach his destination and while the opponent cannot do so regardless of the moves the opponent may choose. Or we want to determine that such a plan does not exist.

A direct representation of the problem as quantified Boolean formula (QBF) has the following structure. The possible moves of the first player are represented by a set Q of logic variables. Every truth assignment to Q corresponds to a plan for the first player. If a truth assignment to Q can be extended to a satisfying solution of a Boolean formula R with variable sets Q and X , then the corresponding plan leads the first player to his destination position. We call such an assignment to Q *R-acceptable*. If an assignment to Q cannot be extended to a satisfying solution of R , the assignment is called *R-unacceptable*. A second Boolean formula S shares the variable set Q with R . In addition, S contains the variables of a set Y that represent the possible moves of the opponent. If an assignment to Q is *S-acceptable*, that is the assignment can be extended to a satisfying solution of S , then the opponent can reach his goal under the corresponding plan for the first player. Thus, the problem demands to find an *R-acceptable* and *S-unacceptable* assignment to Q , or to determine that such an assignment does not exist. Using a quantified Boolean formula, we have to determine whether

$$\exists Q (\exists X R \wedge \forall Y \neg S) \tag{1}$$

evaluates to *True*. We consider here only formulas R and S in *conjunctive normal form* (CNF). That is, formulas that are conjunctions of *clauses* each of which is a disjunction of *literals*. A literal is a variable or negated variable. The problem to evaluate formula (1) with CNF formulas R and S is

called *Q-ALL SAT*. The problem needs to be solved in many applications, for example, in conditional planning (Rintanen [9]), regulatory compliance (Straach and Truemper [13]), and diagnosis (Poole [8]). Further applications are discussed by Eiter, Gottlob, and Leone [3] and Truemper [14]. Hence, solution algorithms for Q-ALL SAT are very much needed.

Q-ALL SAT is at the second level of the polynomial hierarchy. Thus, it is unlikely that one can develop a polynomial solution algorithm. For the polynomial hierarchy, see Stockmeyer [12]. Recent work for solving quantified Boolean formulas at the second or even higher levels is as follows. Kleine Büning, Karpinski, and Flögel [6] propose a resolution-based algorithm. Due to the possibly exponential growth of the formula, the algorithm is not practical. Among the most promising algorithms for quantified Boolean formulas (QBF) are backtracking search algorithms, for example, Cadoli, Schaerf, Giovanardi, and Giovanardi [2], Giunchiglia, Narrizano, and Tacchella [4], Rintanen [10], and Zhang and Malik [17]. These algorithms treat quantified Boolean formulas where the quantifiers constitute a prefix of a CNF formula. In contrast, Q-ALL SAT is composed of two CNF formulas. In principle, the QBF solvers for formulas where all quantifiers are a prefix of the formula, can be used to solve Q-ALL SAT instances. However, the required transformation tends to destroy structural properties that can be exploited in a direct attack on Q-ALL SAT. For example, if R and S are Horn formulas, the transformation does not necessarily result in a quantified Horn formula.

In this paper, we present an algorithm for Q-ALL SAT that takes advantage of the structure of formula (1). The implementation relies on a backtracking scheme. Preliminary tests have shown that a new form of learning clauses for Q-ALL SAT reduces the search tree significantly.

2 The Solution Algorithm

We describe a solution algorithm for Q-ALL SAT called QRSsat. The basic backtracking scheme is outlined below. In Step 1, the algorithm simplifies the CNF formulas R and S by unit-resolution. That is, we set all literals l to *True* that occur in a clause with the single literal l . Such a clause is called a *unit clause*. Since we want to determine an S -unacceptable assignment for Q , unit clauses of S that contain a Q variable cannot be used in the resolution process.

QRSsat(R, S, Q)

1. Do unit-resolution in R on all variables and in S on all variables not in Q ;
2. If R contains an empty clause or all clauses of S are satisfied
return false;
3. If all Q variables are fixed or S contains an empty clause
Solve the SAT instances R and S ;
If R is satisfiable and S is unsatisfiable
return true;
Else
return false;
4. Select a Q variable q ;
5. If QRSsat($R \wedge q, S \wedge q, Q - \{q\}$) returns true
return true;
Else if QRSsat($R \wedge \neg q, S \wedge \neg q, Q - \{q\}$) returns true
return true;
Else
return false;

Two situations trigger direct backtracking in Step 2: R is unsatisfiable or S is satisfied under the current assignment. In both cases, it is possible to learn from the failure a new clause that can be added to R . If R is unsatisfiable, any learning scheme applied in a backtracking-based SAT solver can be used here. Since this form of learning has been well investigated, we refer to previous work, like Bayardo and Schrag [1], João, Marques-Silva, and Sakallah [5], Zhang [15], or Zhang, Madigan, Moskewicz, and Malik [16]. A learning scheme for the second case, that is for satisfiability of S , is described in Section 2.2.

If all Q variables are fixed, the problem is reduced to two SAT problems in R and S since R and S share the Q variables only. See Step 3. If S contains an empty clause, then S is unsatisfiable, and the SAT problem in S is trivial. In this case, it is sufficient to solve R as a SAT problem even if not all Q variables are fixed. We can choose any state-of-the-art SAT solver in Step 3. SAT solvers have become very effective, see SAT 2003 [11]. Observe that, if all clauses of R are satisfied but some Q variables are not yet fixed, the problem is not reduced to a SAT problem in S since we want to achieve unsatisfiability in S for an assignment for Q .

2.1 Selecting the Next Variable

In Step 4, the next variable is selected by a version of the MOMS (Maximum Occurrences in Minimum Size clauses) heuristic used in backtracking search for SAT. The rule selects the variable that occurs most often as a negated and nonnegated literal in short clauses. Thus, the rule aims at using a variable that leads fast to satisfiability or unsatisfiability if the variable is set to *True* as well if it is set to *False*. In the case of Q-ALL SAT, we want to select a variable that either leads fast to a solution or fast to a situation where backtracking is possible. In the first case, the selected variable and its truth value aim at inducing satisfiability in R and unsatisfiability in S . In the second case the variable and value aim at inducing unsatisfiability in R or satisfiability in S . Hence we have adapted the MOMS heuristic as follows. For each Q variable q , we count the nonnegated occurrences of q in 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 R of length i and nonnegated occurrences of q in S in clauses of length i . Define a vector h_q whose i th element is

$$h_q(i) = p_q(i) + n_q(i) - \frac{1}{3} \cdot |p_q(i) - n_q(i)| \quad (2)$$

We select the variable q with the lexicographically largest vector h_q . Thus, variables are first compared with respect to $h_q(1)$. 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*.

2.2 Learning S -conflict Clauses

Suppose an S -acceptable assignment for Q is reached in Step 2 or Step 3. In that case, an assignment for some variables in Q satisfies all clauses of S . Hence the assignment cannot lead to a solution of the Q-ALL SAT instance. We describe the learning process for that case. Using the truth assignment of the Q variables, we define a clause that contains each fixed Q variable and that cuts off that assignment in the search tree. For example, assume that (1) the given Q-ALL SAT instance contains the three Q variables q_1, q_2, q_3 , (2) the search tree sets q_1 to *True* and q_2 to *False*, and (3) all clauses of the formula S are satisfied under that assignment. Then the assignment for q_1 and q_2 cannot be extended to 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 $\neg q_1 \vee q_2$. Let us call such a clause an *S-conflict clause*. The clause could be added to R , but would be useless for the tree search since it

is always satisfied after the succeeding backtracking step and thus does not reduce the search space. Accordingly, we first sharpen each S -conflict clause by the following heuristic.

First, all free Q variables are removed from the clauses of S . In the above example, we have to remove the literals q_3 and $\neg q_3$ from all clauses of S . Then each fixed Q variable q_i is processed as follows. The variable q_i is deleted from the reduced formula S . If the resulting formula is unsatisfiable, the variable q_i is added back again. Otherwise, the literal of q_i in the S -conflict clause is removed from that clause. Assume, in the above example, that S is still satisfiable if q_1 is set to *True* and if the literals of q_2 and q_3 are removed from all clauses in S . Then the S -conflict clauses $\neg q_1 \wedge q_2$ can be sharpened to the unit clause $\neg q_1$. If the processing of the fixed Q -variables removes at least one literal from the original S -conflict clause, then the final, sharpened S -conflict clause is added to R . The heuristic for sharpening S -conflict clauses deletes variables first that have been fixed last in the search tree. Thus the variables fixed most recently are more likely to be removed than the variables fixed in the beginning of the search process. Accordingly, the new clauses will more likely lead to nonlinear backjumping, that is, the algorithm backtracks more than one node in the search tree.

The sharpening process requires the solution of several satisfiability problems arising from S , and thus may seem to be a considerable computational burden if the SAT instance S is hard. However, current SAT solvers have proved to be very effective in many practical applications (see SAT 2003 [11]). In addition, 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 S instance is hard.

3 Computational Results

We have implemented a first version of QRSsat. The implementation includes the learning of S -conflict clauses, but not learning R -conflict clauses. For initial tests, we generated a set of benchmark problems for the planning problem described in Section 1. The problem is structured and closer to real-world problems than randomly generated instances. All computational results were obtained on a Sun ULTRA 5 (400 MHz) workstation. In order to compare our computational results with other solvers, we converted our test instances into the standard format required by QBF solvers. For the comparison, we selected two solvers: QuBE by Giunchiglia, Narrizano, and Tacchella [4], and Semprop by Letz [7]. Both solvers performed very well at the SAT 2003 QBF Evaluation; see SAT 2003 [11].

3.1 The Test Instances

The test instances are divided into 3 problem sets, each of which contains 48 instances. Table 1 displays the problem sets. The second column of Table 1 shows how many instances of each problem

Table 1. Sets of game instances

problem set	solutions	$ Q $	$ X $	$ Y $	no. clauses in R	no. clauses in S
game_15	19	15	0	275–405	1	781–811
game_20	34	20	0	275–405	1	786–826
game_25	34	25	0	275–405	1	791–841

set have a solution, that is, how many instances evaluate to *True*. In case of game_15, 19 of the 48 instances evaluate to *True*. In case of game_20 and game_25, 34 of the 48 instances evaluate to

True. The next three columns display the number of variables. All instances in problem set game_15 contain 15 Q variables, the instances in game_20 contain 20, and the instances in game_25 contain 25 Q variables. In each instance, CNF formula S has 275 to 405 additional Y variables. Since we wanted to concentrate on the effect of learning of S -conflict clauses, we decided to generate formulas R with no X variables and with a single clause only. The number of clauses in S ranges from 781 to 841.

In order to test the instances with the QBF solvers QuBE and Semprop, we converted all instances into the required standard format. In general, the transformation can increase the number of variables and clauses substantially. Since in our test cases, CNF formula R consists of one clause only, the increase in size is not significant. For example one of the instances in game_15 has a 15 Q variables and 355 Y variables. The number of clauses in S is 781. The corresponding instance in standard format has a total number of 372 variables and 796 clauses. In general, the total number of variables increases by 2, and the total number of clauses increases by 14 to 16.

3.2 Evaluation

Table 2 displays the computational results. Due to the large number of test instances, we list the average run time for each of the 3 problem sets. The second, fourth, and sixth column show the average run time for each problem set, measured in seconds. The solution process was aborted if the run time exceeded 20 min. Each column with the average run times is followed by a column that displays how many instances could not be solved within 20 min. The calculation of the average run times considers only the instances that are solved within 20 min by the corresponding algorithm. QRSsat solved each instance within 612sec except for one case, where the time limit of 20 min was

Table 2. Computational results

problem set	QRSsat no. timed		QuBE no. timed		Semprop no. timed	
	(sec)	out	(sec)	out*	(sec)	out
game_15	3.0	0	59.8	4	28.4	0
game_20	22.5	0	93.3	11	204.7	5
game_25	32.7	1	86.6	14	44.5	16

* The number of instances that were not solved with in 20 min includes 9 cases in which QuBE terminated with an error.

exceeded. That limit was exceeded by the solver QuBE in 20 cases (9 cases resulted in an error) and by Semprop in 21 cases. Most of the unsolved instances had no solution. In general, all QBF solvers needed significantly more run time on instances without a solution. Even if we exclude the unsolved instances, QRSsat performs still better on average than QuBE and Semprop. When compared with QuBE, QRSsat is faster by a factor of 6 to 21. Compared with Semprop, QRSsat is faster by a factor of 3 to 9.

Since the transformation of the instances to the standard format does not increase the size of the instances significantly, we attribute the comparatively fast run times of QRSsat mainly to the learning scheme. Especially, if an instance evaluates to *False*, learning of S -conflict clauses speeds up the solution process substantially. In 8 test instances, the number of learned S -conflict clauses exceeded 1000. These instances are typically the ones that could not be solved by the QBF solvers within 20 min. In one case, about 10000 S -conflict clauses were added to R . QRSsat needed 612sec

to solve this instance. In this case, the learned clauses seem to be of little use. In addition, the large amount of clauses added to R increases the run time. We are considering to improve the learning scheme by removing R -conflict clauses whenever they become useless. For example, a clause becomes useless when the assignment that sets all its literals to *False* has been cut off in the search tree.

4 Summary and Future Work

This paper defines the problem Q-ALL SAT arising from a planning problem. Q-ALL SAT needs to be solved in many other applications, like diagnosis. The paper describes the solution algorithm QRSsat. The method includes a scheme for learning clauses. We have reported first results for the instances of a nontrivial problem class. The computational results show that QRSsat is substantially faster than two state-of-the-art QBF solvers.

Q-ALL SAT ignores any costs associated with the values selected for the Q variables. Applications typically give rise to such costs, and thus lead to variations of Q-ALL SAT. Some applications even give rise to quantified logic problems at the third level of the polynomial hierarchy. Here, too, some versions involve costs. For details about these problems and heuristic solution algorithms, see Truemper [14]. In ongoing research, we are aiming for exact solution algorithms which handle large application classes effectively.

References

- [1] Bayardo Jr., R. J., Schrag, R. C.: Using CSP look-back techniques to solve real world SAT instances. Proceedings of the 14th National Conference on Artificial Intelligence (1997) 203–208
- [2] Cadoli, M., Schaerf, M., Giovanardi, A., Giovanardi, M.: An Algorithm to Evaluate Quantified Boolean Formulae and its Experimental Evaluation. Journal of Automated Reasoning (2003) to appear
- [3] Eiter, T., Gottlob, G., Leone, N.: Abduction from Logic Programs: Semantics and Complexity. Theoretical Computer Science 189(1-2) (1997) 129–177.
- [4] Giunchiglia, E., Narrizano, M., Tacchella, A.: Backjumping for Quantified Boolean Logic satisfiability. Artificial Intelligence 145(1) (2003) 99.
- [5] João, P., Marques-Silva, J. P., Sakallah, K. A.: GRASP – A Search Algorithm for Propositional Satisfiability. IEEE Transactions on Computers 48(5) (1999) 506–521.
- [6] Kleine Büning, H., Karpinski, M., Flögel, A.: Resolution for quantified Boolean formulas. Information and Computation 117(1) (1995) 12–18.
- [7] Letz, R.: Lemma and Model Caching in Decision Procedures for Quantified Boolean Formulas. U. Egly and C.G. Fermüller (Eds.): TABLEAUX 2002, LNAI 2381, (2002) 160–175.
- [8] Poole, D.: Normality and Faults in Logic-Based Diagnosis. Proceedings of the 11th International Joint Conference on Artificial Intelligence (1989) 1304–1310.
- [9] Rintanen, J.: Constructing Conditional Plans by a Theorem-Prover. Journal of Artificial Intelligence Research (1999) 323–352.
- [10] Rintanen, J.: Partial implicit unfolding in the Davis-Putnam procedure for quantified Boolean formula. International Conference on Logic Programming, Artificial Intelligence and Reasoning (2001) 362–376
- [11] SAT 2003, Sixth International Conference on Theory and Applications of Satisfiability Testing, QBF Evaluation, <http://www.satlive.org/> (2003)
- [12] Stockmeyer, L. J.: The polynomial hierarchy. Theoretical Computer Science 3 (1976) 1–22
- [13] Straach, J., Truemper, K.: Learning to Ask Relevant Questions. Artificial Intelligence 111 (1999) 301–327
- [14] Truemper, K.: Design of Logic-based Intelligent Systems. John Wiley & Sons, Inc. (2004)
- [15] Zhang, H.: SATO: An Efficient Propositional Prover. Proceedings of the International Conference on Automated Deduction (1997)
- [16] Zhang, L., Madigan, C. F., Moskewicz M. H., Malik, S.: Efficient Conflict Driven Learning in a Boolean Satisfiability Solver. Proceedings of the International Conference on Computer Aided Design (2001)
- [17] Zhang, L., Malik, S.: Towards a Symmetric Treatment of Satisfaction and Conflicts in Quantified Boolean Formula Evaluation. Proceedings of 8th International Conference on Principles and Practice of Constraint Programming (2002)