Watched Data Structures for QBF Solvers

Ian Gent[1], Enrico Giunchiglia[2], Massimo Narizzano[2], Andrew Rowley[1], and Armando Tacchella[2]

[1] Dept. of Computer Science, University of St. Andrews
[2] DIST - Università di Genova

Abstract. In the last few years, we have seen a tremendous boost in the capacity of SAT solvers, such boost mostly due to Chaff. Chaff owes some of its efficiency to its "two-literal watching" data structure. In this paper we present watched data structures for Quantified Bolean Formula (QBF) satisfiability solvers. In particular, we propose (*i*) two Chaff-like literal watching schemes for unit clause detection; (*ii*) a "clause watching" schema suited for detecting pure literals; and (*iii*) a "quantifier watching" schema for the detection of void quantifiers. We have conducted an experimental evaluation of the proposed data structures, using both randomly generated and real-world benchmarks. Our preliminary results indicate that clause watching is very effective, while the other data structures do not have significant effects.

1 Introduction

In the last few years, we have seen a tremendous boost in the capacity of SAT solvers, such boost mostly due to Chaff. Chaff is based on DPLL procedure [1, 2], and owes part of its efficiency to its data structures designed for the specific look-ahead it implements, i.e., unit-propagation. The basic idea is to detect unit clauses by watching two unassigned literals per clause. As soon as one of the watched literals is assigned, another unassigned literal is looked for in the clause: failure to find one implies that the clause is unit. The main advantage of any such procedure is that, when a literal is given a truth value, only its watched occurrences are assigned. This is to be contrasted to traditional DPLL implementations where, when assigning a variable, all its occurrences are considered. This simple idea can be realized in various ways, differing for the specific operations done when assigning a watched literal or when backtracking (see, e.g., [3–5]). In Chaff, backtracking requires a constant number of operations. See [4] for more details.

In this paper we tackle the problem of designing, implementing and experimenting with watching data structures for DPLL-based QBF solvers. In particular, we propose (*i*) two Chaff-like literal watching schemes for unit clause detection; (*ii*) a "clause watching" schema suited for detecting pure literals; and (*iii*) a "quantifier watching" schema for the detection of void quantifiers. We have implemented such watching structures, and we conducted an experimental evaluation, using both randomly generated and real-world benchmarks. Our preliminary results indicate that clause watching is very effective, while the other data structures do not have significant effects. We are currently running a wider set of benchmarks, whose results will be presented in the full paper.

The paper is structured as follows. We first introduce some basic terminology and notation (§2). In §3, we briefly present the standard data structures, leaving their detailed presentation to the full paper. The watched data structures that we propose are comprehensively described in §4. We end the paper with the experimental analysis (§5).

2 Basic definitions

We take for granted the definitions of variable, literal, clause. Notationally, if l is a literal, we write $\bar{l}$ as an abbreviation for x if $l = \neg x$, and for $\neg l$ otherwise.

A *QBF* is an expression of the form

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

where every Q_i $(1 \leq i \leq n)$ is a quantifier (either existential $\exists$ or universal $\forall$); $\boldsymbol{x_1}, \ldots, \boldsymbol{x_n}$ are sets of variables; and Φ is a set of clauses in $\boldsymbol{x_1} \cup \ldots \cup \boldsymbol{x_n}$. We assume that no variable occurs twice in a clause; that $\boldsymbol{x_1}, \ldots, \boldsymbol{x_n}$ are pairwise disjoint; and that $Q_i \neq Q_{i+1}$ $(1 \leq i < n)$. In (1), $Q_1 \boldsymbol{x_1} \ldots Q_n \boldsymbol{x_n}$ is the *prefix*, Φ is the *matrix*, and Q_i is the *bounding quantifier* of each variable in $\boldsymbol{x_i}$.

The semantics of a QBF φ can be defined recursively as follows:

1. If the matrix of φ contains an empty clause then φ is False.
2. If the matrix of φ is the empty set of clauses then φ is True.
3. If φ is $\exists \boldsymbol{x} \psi$ and $x \in \boldsymbol{x}$, φ is True if and only if φ_x or $\varphi_{\neg x}$ are True.
4. If φ is $\forall \boldsymbol{x} \psi$ and $x \in \boldsymbol{x}$, φ is True if and only if φ_x and $\varphi_{\neg x}$ are True.

If φ is a QBF and l is a literal, φ_l is the QBF

1. whose matrix Φ is obtained from the matrix of φ by deleting the clauses C such that $l \in C$, and removing $\bar{l}$ from the others, and
2. whose prefix is obtained from the prefix of φ by deleting the variables not occurring in Φ. Void quantifiers (i.e., quantifiers not binding any variable) are also eliminated.

As usual, we say that a QBF φ is *satisfiable* iff φ is True.

On the basis of the semantics, a simple recursive procedure for determining the satisfiability of a QBF φ, simplifies φ to φ_x and/or $\varphi_{\neg x}$ if x is in the leftmost set of variables in the prefix, until either an empty clause or the empty set of clauses is produced: On the basis of the satisfiability of φ_x and $\varphi_{\neg x}$, the satisfiability of φ can be determined according to the semantics of QBFs.

Most of the current QBF solvers are based on such simple procedure. However, in order to prune the search tree, they introduce some improvements.

The first improvement is that it is possible to directly conclude that a QBF is unsatisfiable if the matrix contains a *contradictory clause*, i.e., a clause with no existential literals. (Notice that the empty clause is also contradictory).

Then, if a literal l is unit or pure in a QBF φ, then φ can be simplified to φ_l. We say that a literal l is

– *Unit* if the matrix contains a *unit clause in* l, i.e., a clause of the form $\{l, l_1, \ldots, l_m\}$ $(m \geq 0)$ with (i) l existential; and (ii) each literal l_i $(1 \leq i \leq m)$ universally quantified inside the quantifier binding l. For example, both x_1 and x_2 are unit in any QBF of the form:

$$\ldots \exists x_1 \forall y_1 \exists x_2 \ldots \{\{x_1, y_1\}, \{x_2\}, \ldots\}.$$

– *Pure* if either l is existential and $\overline{l}$ does not belong to any clause in Φ; or l is universal and l does not belong to any clause in Φ. For example, in the following QBF, the pure literals are y_1 and x_1:

$$\forall y_1 \exists x_1 \forall y_2 \exists x_2 \{\{\neg y_1, y_2, x_2\}, \{x_1, \neg y_2 \neg x_2\}\}.$$

In the above example, notice that after y_1 and x_1 are assigned, $\neg y_2$ can be assigned because is pure, and then x_2 can be assigned because is unit. This simple example shows the importance of implementing pure literal fixing in QBFs: The assignment of a pure existential literal may cause the detection of a pure universal literal, and the assignment of a pure universal literal may cause the detection of unit literals.

Finally, all QBF solvers implement some heuristic in order to decide the best (among those admissible) literal to be used for branching.

3 Unwatched Data Structures

The main requirements of any data structure in a QBF solver is to detect key events. The key events that we want to detect are

1. The occurrence of unit or pure literals.
2. The presence of contradictory clauses in the matrix.
3. The presence of void quantifiers in the prefix: This allows the removal of the quantifier from the prefix.
4. The presence of the empty set of clauses: This allows to immediately backtrack to the last universal variable whose right branch has not been explored yet.

All such events are to be detected while descending the search tree assigning variables. When a variable is assigned, data structures get updated and each condition checked. Of course, changes are stored so that they can be undone while backtracking. Here we briefly describe how such events are detected in our standard procedure. All details will be given in the full paper.

Unit literals and contradictory clauses, assuming a literal l is assigned true, are detected while removing $\overline{l}$ from any clauses it occurs in. To perform this operation more efficiently, each clause is first sorted into existential and universal literals. These sets are then sorted into the order in which the variables occur in the prefix. Further, since a literal can be removed from any point in the clause, it is assumed that a linked list data structure is used to hold the literals.

For pure literals, we store which clauses a variable's literals are contained in. In the same way that a clause contains literals, a variable can be thought to contain c-literals. Each of these c-literals consists of a clause and a sign. The sign of the c-literal is the same as the sign of the literal of the variable in the clause. The c-literals are then stored in the variable, split into negative and positive c-literals. Again, a linked list data structure allows removal of any c-literal efficiently. When a clause is removed, its c-literals of the clause can be removed from the variables left in the clause. Pure literals have no positive or no negative c-literals.

For void quantifiers, the procedure is the same since we can think of a quantifier in a similar way to a clause: A quantifier contains q-variables, which consist of a variable and a quantification. As with literals in clauses, a linked list data structure is required here to allow removal from any part of the quantifier. When a q-variable is assigned, it is removed from the quantifier in which it occurs.

For detecting the empty matrix, we keep a count of the number of clauses. When a clause is marked as removed, this count is decremented and when a clause is restored, the count is incremented: Clauses are never actually removed.

4 Watched Data Structures

As has been demonstrated in SAT solvers such as Chaff, lazy data structures can be more efficient. This is attributed also to the fact that cache memory is used more efficiently. One of the requirements of these data structures that make this true is that no work should be done on the data structure during backtracking. To allow this to happen, no literals are ever removed from clauses, and similarly for q-variables in quantifiers and c-literals in clauses. This allows all the data structures to use arrays in place of linked lists. Here we outline two data structures for watching literals, and one each for clauses and quantifiers.

4.1 Two Literal Watching

In SAT solvers, two literal watching is used for the removal of clauses in addition to the removal of literals from clauses. In a SAT solver, we are only interested in finding a solution; once this has been done, no backtracking is required. This means that we do not care how many variable assignments it takes to get to the solution, or if these variable assignments are superfluous. In QBF solvers this is no longer the case. We are likely to need to backtrack upon finding a solution and so it is important that the empty set of clauses is detected as soon as possible, and that no variable assignments are made that are not absolutely necessary. To facilitate this, when assigning a literal, l, true, we only deal with watched literals from clauses containing $\bar{l}$, but remove all clauses containing l.

The invariants that we wish to uphold in a clause are as follows:

1. The clause contains a true literal and is therefore removed.
2. The clause contains no true existential literals and is therefore false.

3. The clause contains one unassigned existential literal and all unassigned universals are quantified inside the existential and is therefore unit.
4. The clause contains two unassigned watched existential literals.
5. The clause contains one unassigned watched existential literal and one unassigned watched universal literal quantified outside the existential.

These should hold in such a way that nothing has to be done upon backtracking. As before, we assume the literals of the clause are sorted. When removing a literal from a clause, if ever we find a literal that satisfies the clause, the operation is stopped.

If the initial literal is an existential, e_{old}, the rules are as follows:

1. If we find an unassigned, unwatched existential, e_{new} to the right of the current one, watch e_{new}. Due to sorting, e_{new} must be inside e_{old}, and so invariant 5 can still hold.
2. Scan left to find an unassigned, unwatched existential, e_{new}.
3. If we found the other pointer, and e_{new}, watch e_{new}. There must still be two existentials watched.
4. If we didn't find a new pointer or the other pointer, the clause is now contradictory.
5. If we found the other pointer e_{other}, but not e_{new}, we must scan the universals from the left to find an unassigned, unmarked universal, u_{new}, quantified outside e_{other}.
   (a) If we find u_{new}, watch it.
   (b) If we don't, we have a unit clause in e_{other}.
6. If we didn't find the other pointer, but found e_{new}, we must carry on scanning to the left to find the other pointer. If we encounter another unassigned unwatched existential, call it e_{new2}.
   (a) If we find the other pointer, watch the new existential. There must still be two existentials watched.
   (b) If we don't, we must scan the universals to find the watched universal, u_{other}.
      i. If we found e_{new} and e_{new2}, watch e_{new} in place of e_{old} and e_{new2} in place of u_{other}.
      ii. If u_{other} is quantified outside e_{new}, watch e_{new}.
      iii. If u_{other} is quantified inside e_{new}, we must scan to the left to find a new universal, u_{new}, that is quantified outside the existential.
         A. If this is not possible, the clause is unit in e_{new}.
         B. If it is found, watch e_{new} and move the u_{other} pointer to u_{new}.

If the initial literal is a universal, u_{old}, the rules are as follows:

1. Scan to the left and try to find an unwatched existential, e_{new} or the existential watched literal, e_{other}.
2. If we find e_{new}, watch it. It makes more sense to be watching two existentials if possible.
3. If we find e_{other} but not e_{new}, we must scan left and right over the universals to find one that is quantified outside e_{other}.
   (a) If we find it, we watch it.
   (b) If we don't, the clause must be unit in e_{other}.

4.2 Three Literal Watching

In the above, we can be watching an existential and a universal as in invariant 5 but there might be two unassigned existentials in the clause. To reference this problem, we suggest a method where by three literals are watched in a clause: two existentials, and one universal. The invariants for this are as follows (invariants 1-3 are as above):

4. watched existentials are both unassigned.
5. One of the two watched existentials is assigned, and the watched universal literal is unassigned and is quantified outside the watched unassigned existential literal.

In order to determine the other watched literals in the clause as quickly as possible, each clause contains a set of watched literals. These point to the actual watched literals in the clause. It is now less important that the existential literals in the clause are sorted, but universal sorting is still important, since we still need to scan for universals with a proper position in the prefix. As before, search is stopped if a literal that satisfies the clause is found.

If the initial literal is an existential, e_{old}, the rules are as follows:

1. Determine the other existential watched literal, e_{other}, and the universal watched literal u.
2. If e_{other} is assigned false, find a universal literal, u_{sat} that satisfies the clause.
   (a) If u_{sat} exists, stop.
   (b) If u_{sat} does not exist, the clause is contradictory.
3. If e_{other} is unassigned find another unwatched existential literal, e_{new}.
   (a) If e_{new} exists, watch it.
   (b) If e_{new} does not exist, scan the universals to the right until an unassigned universal u_{new} is found that is quantified outside e_{other}.
      i. If u_{new} exists, watch it.
      ii. If u_{new} does not exist, the clause is unit in e_{other}.

If the initial literal is universal, the rules are as follows:

1. Determine the existential watched literals, e_1 and e_2.
2. If e_1 and e_2 are both unassigned, stop.
3. If only one of e_1 and e_2 are assigned, scan the universals until an unassigned universal, u_{new}, is found that is quantified outside the unassigned existential watched literal.
   (a) If u_{new} exists, watch it.
   (b) If u_{new} does not exist, the clause is unit.

4.3 Clause Watching

In clause watching, we need to detect if either or both of the signs of the c-literals become empty. For this, we require two watched c-literals per variable, one of positive sign, and the other of negative sign.

The invariants for c-literal watching are:

1. The variable is pure in one or other of the signs.
2. The variable is removed.
3. There are two watched c-literals in the variable, one of each sign.

When a c-literal is removed, the rules are as follows:

1. Search for a new c-literal of the same sign, c_{new}.
   (a) If c_{new} exists, watch it.
   (b) If c_{new} does not exist, search for an unassigned c-literal of the opposite sign, c_o.
      i. If c_o exists, the variable is pure in the sign of c_o.
      ii. If c_o does not exist, the variable is removed.

4.4 Quantifier Watching

In two literal watching in SAT solvers, the two literals allow us to detect when a clause only contains one item, as well as when it is empty. In quantifier watching, we only need to know when the quantifier is empty, and for this, only one watched q-variable is needed per quantifier.

The invariants for q-variable watching are:

1. The quantifier is empty and so removed.
2. There is one watched unassigned q-variable in the quantifier.

When we remove the watched q-variable, q_{old}, the rules are as follows:

1. Search left and right for an unassigned q-variable, q_{new}.
   (a) If q_{new} exists, watch it.
   (b) If q_{new} does not exist, remove the quantifier.

5 Experimental Analysis

We implemented the above ideas in a QBF solvers featuring both conflict and solution directed backjumping [6]. In order to test the effectiveness of the watched data structures, we run the 5 different versions of the solver:

1. CSBJ represents the basic solver with the standard data structures,
2. CSBJ+2WL is CSBJ plus two literal watching, as described in § 4.1,
3. CSBJ+3WL is CSBJ plus three literal watching, as described in § 4.2,
4. CSBJ+WCL is CSBJ plus watching clauses, as described in § 4.3, and
5. CSBJ+WQT is CSBJ plus watching quantifiers, as described in § 4.4.

We considered all the 322 real world problems available at `www.qbflib.org`, excluding the robot problems. Of these 322 problems, all the solvers (*i*) timed out on 93, and (*ii*) were able to solve 28 in 0 seconds. The results on the remaining 201 problems are shown in Figure 1. In the figure, on the y-axis there is the time taken by each procedure to solve the number of instances specified on the x-axis. The time out is 7200s.

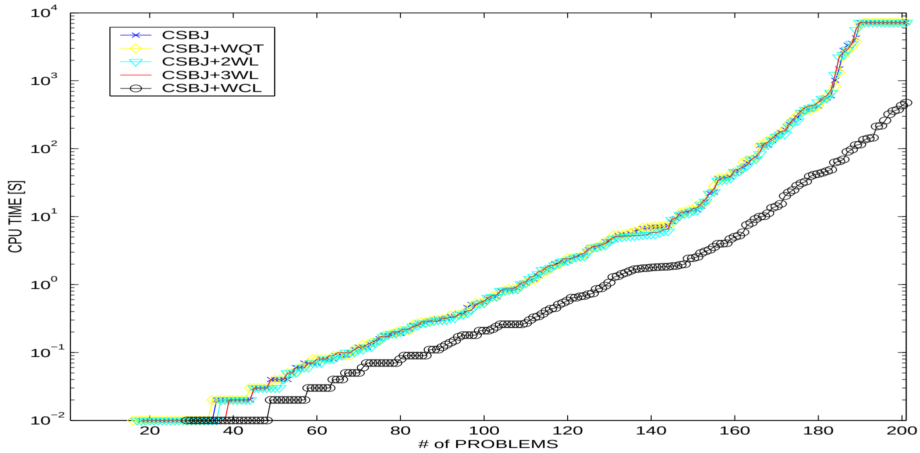

Fig. 1. Performances of CSBJ augmented with watched data structures on real-world instances.

As it can be seen, neither the two nor the three literal watching structures cause a speed-up in the performances. One reason could be that the average length of clauses is 3.8. On the positive side, we see that watching clauses provides a significant boost: For example, to solve 167 instances out of the 201 considered, CSBJ+WCL takes 10s, while the other solvers take 100s.

We have considered also instances randomly generated according to model A of [7]. The results that we got are similar: watching literal and quantifiers do not produce benefits, while watching clauses can give orders of magnitude speed-ups. These and other experimental results will be presented in the full paper.

References

1. Davis, M., Putnam, H.: A computing procedure for quantification theory. Journal of the ACM **7** (1960) 201–215
2. Davis, M., Logemann, G., Loveland, D.: A machine program for theorem proving. Journal of the ACM **5** (1962)
3. Zhang, H., Stickel, M.E.: An efficient algorithm for unit propagation. In: Proceedings of the Fourth International Symposium on Artificial Intelligence and Mathematics (AI-MATH'96), Fort Lauderdale (Florida USA) (1996)
4. Moskewicz, M.W., Madigan, C.F., Zhao, Y., Zhang, L., Malik, S.: Chaff: Engineering an Efficient SAT Solver. In: Proc. DAC. (2001)
5. Lynce, I., Marques-Silva, J.: Efficient data structures for fast sat solvers. In: Proceedings of the 5th International Symposium on the Theory and Applications of Satisfiability Testing (SAT'02). (2002)
6. Giunchiglia, E., Narizzano, M., Tacchella, A.: Backjumping for Quantified Boolean Logic Satisfiability. Artificial Intelligence **144** (2003)
7. Gent, I., Walsh, T.: Beyond NP: the QSAT phase transition. In: Proc. AAAI. (1999) 648–653