Learning for Quantified Boolean Logic Satisfiability

Enrico Giunchiglia and **Massimo Narizzano** and **Armando Tacchella**
DIST - Università di Genova
Viale Causa 13, 16145 Genova, Italy
{enrico,mox,tac}@mrg.dist.unige.it

Abstract

Learning, i.e., the ability to record and exploit some information which is unveiled during the search, proved to be a very effective AI technique for problem solving and, in particular, for constraint satisfaction. We introduce learning as a general purpose technique to improve the performances of decision procedures for Quantified Boolean Formulas (QBFs). Since many of the recently proposed decision procedures for QBFs solve the formula using search methods, the addition of learning to such procedures has the potential of reducing useless explorations of the search space. To show the applicability of learning for QBF satisfiability we have implemented it in QUBE, a state-of-the-art QBF solver. While the backjumping engine embedded in QUBE provides a good starting point for our task, the addition of learning required us to devise new data structures and led to the definition and implementation of new pruning strategies. We report some experimental results that witness the effectiveness of learning. Noticeably, QUBE augmented with learning is able to solve instances that were previously out if its reach. To the extent of our knowledge, this is the first time that learning is proposed, implemented and tested for QBFs satisfiability.

Introduction

The goal of learning in problem solving is to record in a useful way some information which is unveiled during the search, so that it can be reused later to avoid useless explorations of the search space. Learning leads to substantial speed-ups when solving constraint satisfaction problems (see, e.g., (Dechter 1990)) and, more specifically, propositional satisfiability problems (see, e.g., (Bayardo, Jr. & Schrag 1997)).

We introduce learning to improve the performances of decision procedures for Quantified Boolean Formulas (QBFs). Due to the particular nature of the satisfiability problem for QBFs, there are both unsuccessful and successful terminal nodes. The former correspond to assignments violating some constraint, while the latter correspond to assignments satisfying all the constraints. With our procedure, we are able to learn from both types of terminal nodes. Learning from constraint violations —best known in the literature as *nogood* learning— has been implemented and tested in other contexts, but never for QBFs. Learning from the satisfaction of all constraints —called *good* learning in (Bayardo Jr. & Pehoushek 2000)— is formally introduced, implemented and tested here for the first time. Furthermore, learning goods enables to extend Boolean Constraint Propagation (BCP) (see, e.g., (Freeman 1995)) to universally quantified literals. Such an extension, which does not make sense in standard QBF reasoning, is obtained as a side effect of our learning schema.

To show the effectiveness of learning for the QBFs satisfiability problem, we have implemented it in QUBE, a state-of-the-art QBF solver. Since learning exploits information collected during backtracking, the presence of a backjumping engine in QUBE provides a good starting point for our task. Nevertheless, the dual nature of learned data, i.e., nogoods as well as goods, and the interactions with standard QBFs pruning techniques, require the development of ad hoc data structures and algorithms. Using QUBE, we have done some experimental tests on several real-world QBFs, corresponding to planning (Rintanen 1999a) and circuit verification (Abdelwaheb & Basin 2000) problems. The results witness the effectiveness of learning. Noticeably, QUBE augmented with learning is able to solve in a few seconds instances that previously were out of its reach.

Finally, we believe that the learning techniques that we propose enjoy a wide applicability. Indeed, in the last few years we have witnessed the presentation of several implemented decision procedures for QBFs, like QKN (Kleine-Büning, Karpinski, & Flögel 1995), EVALUATE (Cadoli, Giovanardi, & Schaerf 1998), DECIDE (Rintanen 1999b), QSOLVE (Feldmann, Monien, & Schamberger 2000), and QUBE (Giunchiglia, Narizzano, & Tacchella 2001b). Most of the above decision procedures are based on search methods and thus the exploitation of learning techniques has the potential of improving their performances as well. Furthermore, we believe that the ideas behind the good learning mechanism can be exploited also to effectively deal with other related problems, as the one considered in (Bayardo Jr. & Pehoushek 2000).

The paper is structured as follows. We first review the logic of QBFs, and the ideas behind DLL-based decision procedures for QBFs. Then, we introduce learning, presenting the formal background, the implementation and the experimental analysis. We end with some final remarks.

Copyright © 2002, American Association for Artificial Intelligence (www.aaai.org). All rights reserved.

Quantified Boolean Logic

Logic

Consider a set P of propositional letters. An *atom* is an element of P. A *literal* is an atom or the negation of an atom. In the following, for any literal l,

- $|l|$ is the atom occurring in l; and
- $\bar{l}$ is $\neg l$ if l is an atom, and is $|l|$ otherwise.

A *propositional formula* is a combination of atoms using the k-ary ($k \geq 0$) connectives $\wedge, \vee$ and the unary connective $\neg$. In the following, we use TRUE and FALSE as abbreviations for the empty conjunction and the empty disjunction respectively.

A *QBF* is an expression of the form

$$\varphi = Q_1 x_1 Q_2 x_2 \ldots Q_n x_n \Phi \qquad (n \geq 0) \tag{1}$$

where

- every Q_i ($1 \leq i \leq n$) is a quantifier, either existential $\exists$ or universal $\forall$,
- $x_1, \ldots, x_n$ are pairwise distinct atoms in P, and
- Φ is a propositional formula in the atoms $x_1, \ldots, x_n$.

$Q_1 x_1 \ldots Q_n x_n$ is the *prefix* and Φ is the *matrix* of (1). We also say that a literal l is *existential* if $\exists |l|$ belongs to the prefix of (1), and is *universal* otherwise. Finally, in (1), we define

- the *level of an atom* x_i, to be 1 + the number of expressions $Q_j x_j Q_{j+1} x_{j+1}$ in the prefix with $j \geq i$ and $Q_j \neq Q_{j+1}$;
- the *level of a literal* l, to be the level of $|l|$;
- the *level of the formula*, to be the level of x_1.

The semantics of a QBF φ can be defined recursively as follows. If the prefix is empty, then φ's satisfiability is defined according to the truth tables of propositional logic. If φ is $\exists x \psi$ (respectively $\forall x \psi$), φ is satisfiable if and only if φ_x or (respectively and) $\varphi_{\neg x}$ are satisfiable. If $\varphi = Qx\psi$ is a QBF and l is a literal, φ_l is the QBF obtained from ψ by substituting l with TRUE and $\bar{l}$ with FALSE. It is easy to see that if φ is a QBF without universal quantifiers, the problem of deciding the satisfiability of φ reduces to SAT.

Two QBFs are *equivalent* if they are either both satisfiable or both unsatisfiable.

DLL Based Decision Procedures

Consider a QBF (1). From here on, we assume that Φ is in conjunctive normal form (CNF), i.e., that it is a conjunction of disjunctions of literals. Using common clause form transformations, it is possible to convert any QBF into an equivalent one meeting the CNF requirement, see, e.g., (Plaisted & Greenbaum 1986). As standard in propositional satisfiability, we represent Φ as a set of clauses, and assume that for any two elements l, l' in a clause, it is not the case that $l = \overline{l'}$. With this notation,

- The empty clause $\{\}$ stands for FALSE,
- The empty set of clauses $\{\}$ stands for TRUE,
- The propositional formula $\{\{\}\}$ is equivalent to FALSE.

Thus, for example, a typical QBF is the following:

$$\begin{array}{r} \exists x \forall y \exists z \exists s \{\{\neg x, \neg y, z\}, \{x, \neg y, \neg s\}, \\ \{\neg y, \neg z\}, \{y, z, \neg s\}, \{z, s\}\}. \end{array} \tag{2}$$

In (Cadoli, Giovanardi, & Schaerf 1998), the authors showed how it is possible to extend DLL in order to decide satisfiability of φ. As in DLL, variables are selected and are assigned values. The difference wrt DLL is that variables need to be selected taking into account the order in which they occur in the prefix. More precisely, a variable can be selected only if all the variables at higher levels have been already selected and assigned. The other difference is that backtracking to the last universal variable whose values have not been both explored occurs when the formula simplifies to TRUE. In the same paper, the authors also showed how the familiar concepts of "unit" and "monotone" literal from the SAT literature can be extended to QBFs and used to prune the search. In details, a literal l is

- *unit* in (1) if l is existential, and, for some $m \geq 0$,
  - a clause $\{l, l_1, \ldots, l_m\}$ belongs to Φ, and
  - each expression $\forall |l_i|$ ($1 \leq i \leq m$) occurs at the right of $\exists |l|$ in the prefix of (1).
- *monotone* if either l is existential [resp. universal], l belongs to some [resp. does not belong to any] clause in Φ, and $\bar{l}$ does not belong to any [resp. belongs to some] clause in Φ.

If a literal l is unit or monotone in Φ, then φ is satisfiable iff φ_l is satisfiable. Early detection and assignment of unit and monotone literals are fundamental for the effectiveness of QBF solvers. See (Cadoli, Giovanardi, & Schaerf 1998) for more details.

A typical computation tree for deciding the satisfiability of a QBF is represented in Figure 1.

Learning

CBJ and SBJ

The procedure presented in (Cadoli, Giovanardi, & Schaerf 1998) uses a standard backtracking schema. In (Giunchiglia, Narizzano, & Tacchella 2001b), the authors show how Conflict-directed Backjumping (CBJ) (Prosser 1993b) can be extended to QBFs, and introduce the concept of Solution-directed Backjumping (SBJ). The idea behind both forms of backjumping is to dynamically compute a subset ν of the current assignment which is "responsible" for the current result. Intuitively, "responsible" means that flipping the literals not in ν would not change the result. In the following, we formally introduce these notions to put the discussion on firm grounds.

Consider a QBF φ. A sequence $\mu = l_1; \ldots; l_m$ ($m \geq 0$) of literals is an *assignment for* φ if for each literal l_i in μ

- l_i is unit, or monotone in $\varphi_{l_1;\ldots;l_{i-1}}$; or
- for each atom x at a higher level of l_i, there is an l_j in μ with $j < i$ and $|l_j| = x$.

For any sequence of literals $\mu = l_1; \ldots; l_m$, φ_μ is the QBF obtained from φ by

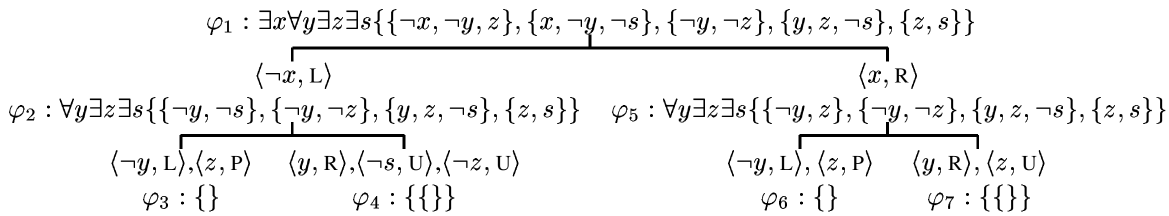

Figure 1: A typical computation tree for (2). U, P, L, R stand for UNIT, PURE, L-SPLIT, R-SPLIT respectively, and have the obvious meaning.

- deleting the clauses in which at least one of the literals in μ occurs;
- removing $\bar{l}_i$ ($1 \leq i \leq m$) from the clauses in the matrix; and
- deleting $|l_i|$ ($1 \leq i \leq m$) and its binding quantifier from the prefix.

For instance, if φ is (2) and μ is $\neg x$, then φ_μ is the formula φ_2 in Figure 1.

Consider an assignment $\mu = l_1; \ldots; l_m$ for φ.

A set of literals ν is a *reason for φ_μ result* if

- $\nu \subseteq \{l_1, \ldots, l_m\}$, and
- for any sequence μ' such that
  - $\nu \subseteq \{l : l \text{ is in } \mu'\}$,
  - $\{|l| : l \text{ is in } \mu'\} = \{|l| : l \text{ is in } \mu\}$,

  $\varphi_{\mu'}$ is satisfiable iff φ_μ is satisfiable.

According to previous definitions, e.g., if φ is (1) and μ is $\neg x; \neg y; z$ then $\varphi_\mu = \varphi_3 = \{\}$ and $\{\neg y, z\}$ is a reason for φ_μ result (see Figure 1).

Reasons can be dynamically computed while backtracking, and are used to skip the exploration of useless branches. In the previous example, the reason $\{\neg y, z\}$ is computed while backtracking from φ_3.

Let ν be a reason for φ_μ result. We say that

- ν is a *reason for φ_μ satisfiability* if φ_μ is satisfiable, and
- ν is a *reason for φ_μ unsatisfiability*, otherwise.

See (Giunchiglia, Narizzano, & Tacchella 2001b) for more details.

Learning

The idea behind learning is simple: reasons computed during the search are stored in order to avoid discovering the same reasons in different branches. In SAT, the learnt reasons are "nogoods" and can be simply added to the set of input clauses. In the case of QBFs, the situation is different. Indeed, we have two types of reasons ("good" and "nogood"), and they need different treatments. Further, even restricting to nogoods, a reason can be added to the set of input clauses only under certain conditions, as sanctioned by the following proposition.

Consider a QBF φ. We say that an assignment $\mu = l_1; \ldots; l_m$ for φ is *prefix-closed* if for each literal l_i in μ, all the variables at higher levels occur in μ.

Proposition 1 *Let φ be a QBF. Let $\mu = l_1; \ldots; l_m$ ($m \geq 0$) be a prefix-closed assignment for φ. If ν is a reason for φ_μ unsatisfiability then φ is equivalent to*

$$Q_1 x_1 \ldots Q_n x_n (\Phi \wedge (\vee_{l \in \nu} \bar{l})).$$

As an application of the above proposition, consider the QBF φ

$$\exists x \forall y \exists z \exists s \{\{y, z\}, \{\neg y, \neg z, s\}, \{\neg x, z\}, \{\neg y, \neg s\}, \{y, \neg s\}\}$$

We have that φ is satisfiable, and

- the sequences $x; y; z$ and $x; z; y$ are prefix-closed assignments for φ,
- $\varphi_{x;y;z} = \varphi_{x;z;y} = \exists s \{\{s\}, \{\neg s\}\}$ is unsatisfiable, and $\{y, z\}$ is a reason for both $\varphi_{x;y;z}$ and $\varphi_{x;z;y}$ unsatisfiability,
- we can add $\{\neg y, \neg z\}$ to the matrix of φ and obtain an equivalent formula.

On the other hand, using the same example, we can show that the requirement that μ be prefix-closed in Proposition 1 cannot be relaxed. In fact, we have that

- the sequence $x; z$ is an assignment for φ, but it is not prefix-closed,
- $\varphi_{x;z} = \forall y \exists s \{\{\neg y, s\}, \{\neg y, \neg s\}, \{y, \neg s\}\}$ is unsatisfiable, and $\{z\}$ is a reason for $\varphi_{x;z}$ unsatisfiability,
- if we add $\{\neg z\}$ to the matrix of φ, we obtain an unsatisfiable formula.

Notice that the sequence $x; z$ is a prefix-closed assignment if we exchange the y with the z quantifiers, i.e.,

$$\exists x \exists z \forall y \exists s \{\{y, z\}, \{\neg y, \neg z, s\}, \{\neg x, z\}, \{\neg y, \neg s\}, \{y, \neg s\}\}.$$

In the above QBF φ, $\{z\}$ is a reason for $\varphi_{x;z}$ unsatisfiability; we can add $\{\neg z\}$ to the matrix of φ; and we can immediately conclude that φ is unsatisfiable.

In the hypotheses of Proposition 1, we say that the clause $(\vee_{l \in \nu} \bar{l})$ is a *nogood*. Notice, that if we have only existential quantifiers —as in SAT— any assignment is prefix-closed, each reason corresponds to a nogood ν, and ν can be safely added to (the matrix of) the input formula.

A proposition "symmetric" to Proposition 1 can be stated for good learning.

Proposition 2 *Let φ be a QBF. Let $\mu = l_1; \ldots; l_m$ ($m \geq 0$) be a prefix-closed assignment for φ. If ν is a reason for φ_μ satisfiability then φ is equivalent to*

$$Q_1 x_1 \ldots Q_n x_n (\neg(\vee_{l \in \nu} \bar{l}) \vee \Phi).$$

As before, in the hypotheses of Proposition 2, we say that the clause $(\vee_{l \in \nu} \bar{l})$ is a *good*. For example, with reference to the formula φ at the top of Figure 1,

- the assignment $\neg x; \neg y; z$ is prefix-closed,
- $\varphi_{\neg x;\neg y;z} = \varphi_3 = \{\}$ is satisfiable, and $\{\neg y; z\}$ is a reason for $\varphi_{\neg x;\neg y;z}$ satisfiability,
- the clause $(y \vee \neg z)$ is a good, and its negation can be put in disjunction with the matrix of φ.

An effective implementation of learning requires an efficient handling of goods and nogoods. The situation is more complicated than in SAT because goods have to be added as disjunctions to the matrix. However, once we group all the goods together, we obtain the negation of a formula in CNF, and ultimately the negation of a set of clauses. The treatment of nogoods is not different from SAT: we could simply add them to the matrix. However, since exponentially many reasons can be computed, some form of space bounding is required. In SAT, the two most popular space bounded learning schemes are

- *size learning of order* n (Dechter 1990): a nogood is stored only if its cardinality is less or equal to n,
- *relevance learning of order* n (Ginsberg 1993): given a current assignment μ, a nogood ν is stored as long as the number of literals in ν and not in μ is less or equal to n.

(See (Bayardo, Jr. & Miranker 1996) for their complexity analysis). Thus, in size learning, once a nogood is stored, it is never deleted. In relevance learning, stored nogoods are dynamically added and deleted depending on the current assignment.

The above notions of size and relevance learning trivially extend to QBFs for both goods and nogoods, and we can take advantage of the fast mechanisms developed in SAT to efficiently deal with learnt reasons. In practice, we handle three sets of clauses:

- the set of clauses Ψ corresponding to the goods learnt during the search;
- the set of clauses Φ corresponding to the input QBF; and
- the set of clauses Θ corresponding to the nogoods learnt during the search.

From a logical point of view, we consider Extended QBFs. An *Extended QBF* (EQBF) is an expression of the form

$$Q_1x_1 \ldots Q_nx_n\langle\Psi, \Phi, \Theta\rangle \qquad (n \geq 0) \tag{3}$$

where

- $Q_1x_1 \ldots Q_nx_n$ is the prefix and is defined as above, and
- Ψ, Φ and Θ are sets of clauses in the atoms $x_1, \ldots, x_n$ such that both

  $$Q_1x_1 \ldots Q_nx_n(\neg\Psi \vee \Phi) \tag{4}$$

  and

  $$Q_1x_1 \ldots Q_nx_n(\Phi \wedge \Theta) \tag{5}$$

  are logically equivalent to (1).

Initially Ψ and Θ are the empty set of clauses, and Φ is the input set of clauses. As the search proceeds,

- Nogoods are determined while performing CBJ and are added to Θ; and
- Goods are determined while performing SBJ and are added to Ψ.

In both cases, the added clauses will be eventually deleted when performing relevance learning (this is why the clauses in Θ are kept separated from the clauses in Φ). Notice that in a nogood C (and in the input clauses as well) the universal literals that occur to the right of all the existential literals in C can be removed (Kleine-Büning, Karpinski, & Flögel 1995). Analogously, in a good C, the existential literals that occur to the right of the universal literals in C can be removed.

Notice that the clauses in Ψ have to be interpreted — because of the negation symbol in front of Ψ in (4)— in the dual way wrt the clauses in Φ or Θ. For example, an empty clause in Ψ allows us to conclude that Ψ is equivalent to FALSE and thus that (1) is equivalent to TRUE. On the other hand, if Ψ is the empty set of clauses, this does not give us any information about the satisfiability of Φ.

Because of the clauses in Ψ and Θ the search can be pruned considerably. Indeed, a terminal node can be anticipated because of the clauses in Ψ or Θ. Further, we can extend the notion of unit to take into account the clauses in Ψ and/or Θ. Consider an EQBF (3). A literal l is

- *unit* in (3) if
  - either l is existential, and, for some $m \geq 0$,
    - a clause $\{l, l_1, \ldots, l_m\}$ belongs to Φ or Θ, and
    - each expression $\forall|l_i|$ $(1 \leq i \leq m)$ occurs at the right of $\exists|l|$ in the prefix of (3).
  - or l is universal, and, for some $m \geq 0$,
    - a clause $\{l, l_1, \ldots, l_m\}$ belongs to Ψ, and
    - each expression $\exists|l_i|$ $(1 \leq i \leq m)$ occurs at the right of $\forall|l|$ in the prefix of (3).

Thus, existential and universal literals can be assigned by "unit propagation" because of the nogoods and goods stored. The assignment of universal literals by unit propagation is a side effect of the good leaning mechanism, and there is no counterpart in standard QBF reasoning.

To understand the benefits of learning, consider the QBF (2). The corresponding EQBF is

$$\begin{aligned}\exists x\forall y\exists z\exists s\langle\{\}, \{\{\neg x, \neg y, z\}, \{x, \neg y, \neg s\},\\ \{\neg y, \neg z\}, \{y, z, \neg s\}, \{z, s\}\}, \{\}\rangle.\end{aligned} \tag{6}$$

With reference to Figure 1, the search proceeds as in the Figure, with the first branch leading to φ_3. Then, given what we said in the paragraph below Proposition 2, the good $\{y, \neg z\}$ (or $\{y\}$) is added to the initial EQBF while backtracking from φ_3. As soon as x is flipped,

- y is detected to be unit and consequently assigned; and
- the branch leading to φ_6 is not explored.

As this example shows, (good) learning can avoid the useless exploration of some branches.

Test File	T/F	LV.	$\#\exists x$	$\#\forall x$	# CL.	TIME-BT	NODES-BT	TIME-BJ	NODES-BJ	TIME-LN	NODES-LN
Adder.2-S	T	4	35	16	109	0.03	1707	0.04	1643	0.10	1715
Adder.2-U	F	3	44	9	110	2.28	150969	0.49	20256	0.20	1236
B*3i.4.4	F	3	284	4	2928	> 1200	–	> 1200	–	0.07	903
B*3ii.4.3	F	3	243	4	2533	> 1200	–	2.28	59390	0.04	868
B*3ii.5.2	F	3	278	4	2707	> 1200	–	52.47	525490	1.59	3462
B*3ii.5.3	T	3	300	4	3402	> 1200	–	> 1200	–	203.25	39920
B*3iii.4	F	3	198	4	1433	> 1200	–	0.56	18952	0.04	807
B*3iii.5	T	3	252	4	1835	> 1200	–	> 1200	–	1.83	11743
B*4ii.6.3	F	3	831	7	15061	> 1200	–	> 1200	–	176.21	53319
B*4iii.6	F	3	720	7	9661	> 1200	–	> 1200	–	65.63	156896
Impl08	T	17	26	8	66	0.08	12509	0.03	3998	0.00	65
Impl10	T	21	32	10	82	0.60	93854	0.20	22626	0.01	96
Impl12	T	25	38	12	98	4.49	698411	1.15	127770	0.00	133
Impl14	T	29	44	14	114	33.22	5174804	6.72	721338	0.00	176
Impl16	T	33	50	16	130	245.52	38253737	39.03	4072402	0.00	225
Impl18	T	37	56	18	146	> 1200	–	227.45	22991578	0.01	280
Impl20	T	41	62	20	162	> 1200	–	> 1200	–	0.01	341
L*BWL*A1	F	3	1098	1	62820	> 1200	–	224.57	265610	21.63	20000
L*BWL*B1	F	3	1870	1	178750	> 1200	–	> 1200	–	57.44	34355
C*12v.13	T	3	913	12	4582	0.20	4119	0.23	4119	8.11	4383

Table 1: QUBE results on 20 structured problems. Names have been abbreviated to fit into the table.

Implementation and Experimental Analysis

In implementing the above ideas, the main difficulty that has to be faced is caused by the unit and monotone pruning rules. Because of them, many assignments are not prefix-closed and thus the corresponding reasons, according to Propositions 1 and 2, cannot be stored in form of goods or nogoods. One obvious solution is to enable the storing only when the current assignment is prefix-closed. Unfortunately, it may be the case that the assignment is not prefix-closed because of a literal at its beginning. In these cases, learning would give no or little benefits. The other obvious solution is to allow the generation of prefix-closed assignments only. This implies that unit and monotone literals get no longer propagated as soon as they are detected, but only when the atoms at higher levels are assigned. This solution may cause a dramatic worsening of the performances of the QBF solver.

A closer analysis to the problem, reveals that troubles arise when backtrack occurs on a literal l, and in the reason for the current result there is a "problematic" literal l' whose level is minor than the level of l. Indeed, if this is the case, the assignment resulting after the backtracking will not be prefix-closed. To get rid of each of these literals l', we substitute l' with the already computed reason for rejecting $\overline{l'}$. The result may still contain "problematic" literals (these literals may have been introduced in the reason during the substitution process), and we go on substituting them till no one is left. Intuitively, in each branch we are computing the reasons corresponding to the assignment in which the "problematic" literals are moved to the end of the assignment.

We have implemented the above ideas in QUBE, together with a relevance learning scheme. QUBE is a state-of-the-art QBF solver (Giunchiglia, Narizzano, & Tacchella 2001b; 2001a). To test the effectiveness of learning, we considered the structured examples available at `www.mrg.dist.unige.it/qbflib`. This test-set consists of structured QBFs corresponding to planning problems (Rintanen 1999a) and formal verification problems (Abdelwaheb & Basin 2000). The results on 20 instances are summarized in Table 1. In the Table:

- T/F indicates whether the QBF is satisfiable (T) or not (F),
- LV. indicates the level of the formula,
- $\#\exists x$ (resp. $\#\forall x$) shows the number of existentially (resp. universally) bounded variables,
- #CL. is the number of clauses in the matrix,
- TIME-BT/NODES-BT (resp. TIME-BJ/NODES-BJ, resp. TIME-LN/NODES-LN) is the CPU time in seconds and number of branches of QUBE-BT (resp. QUBE-BJ, resp. QUBE-LN). QUBE-BT, QUBE-BJ, QUBE-LN are QUBE with standard backtracking; with SBJ and CBJ enabled; with SBJ, CBJ and learning of order 4 enabled, respectively.

The tests have been run on a network of identical Pentium III, 600MHz, 128MB RAM, running SUSE Linux ver. 6.2. The time-limit is fixed to 1200s. The heuristic used for literal selection is the one described in (Feldmann, Monien, & Schamberger 2000).

The first observation is that QUBE-LN, on average, produces a dramatic reduction in the number of branching nodes. In some cases, QUBE-LN does less than 0.1% of the branching nodes of QUBE-BJ and QUBE-BT. However, in general, QUBE-LN is not ensured to perform less branching nodes than QUBE-BJ. Early termination or pruning may prevent the computation along a branch whose reason can enable a "long backjump" in the search stack, see (Prosser 1993a). For the same reason, there is no guarantee that by increasing the learning order, we get a reduction in the number of branching nodes. For example, on L*BWL*B1, QUBE-LN branching nodes are 45595, 34355 and 55509 if run with learning order 2, 4 and 6 respectively. In terms of speed, learning may lead to very substantial improvements. The first observation is that QUBE-LN is able to solve 7 more examples than QUBE-BJ, and 12 more than QUBE-BT. Even more, some examples which are not solved in the time-limit by QUBE-BJ and QUBE-BT, are solved in less than 1s by QUBE-LN. However, learning does not always pay-off. To have an idea of the learning overhead, consider the results on the C*12v.13 example, shown last in the table. This benchmark is peculiar because backjumping plays no role, i.e., QUBE-BT and QUBE-BJ perform the same number of branching nodes.[1] On this example, QUBE-LN performs more branching node than QUBE-BJ. These extra branching nodes of QUBE-LN are not due to a missed "long backjump". Instead, they are due to the fact that QUBE-BT and QUBE-BJ skip the branch on a variable when it does not occur in the matrix of the current QBF, while QUBE-LN takes into account also the databases of the learnt clauses.

The table above shows QUBE-LN performances with relevance learning of order 4. Indeed, by changing the learning order we may get completely different figures. For example, on L*BWL*B1, QUBE-LN running times are 95.54, 57.44 and 112.73s with learning order 2, 4 and 6 respectively. We do not have a general rule for choosing the best learning order. According to the experiments we have done, 4 seems to be a good choice.

Conclusions

In the paper we have shown that it is possible to generalize nogood learning to deal with QBFs. We have introduced and implemented good learning. As we have seen, the interactions between nogood/good learning and the unit/monotone pruning heuristic is not trivial. We have implemented both nogood and good learning in QUBE, together with a relevance learning scheme for limiting the size of the learned clauses database. The experimental evaluation shows that very substantial speed-ups can be obtained. It also shows that learning has some overhead. However, we believe that such overhead can be greatly reduced extending ideas presented in (Moskewicz *et al.* 2001) for SAT. In particular, the extension of the "two literal watching" idea, originally due to (Zhang & Stickel 1996), seems a promising direction. This will be the subject of future research. Finally, we believe that the learning techniques that we propose can be successfully integrated in other QBFs solvers, and that similar ideas can be exploited to effectively deal with other related problems, such as the one considered in (Bayardo Jr. & Pehoushek 2000).

We have also tested QUBE-LN on more than 120000 randomly generated problems for checking the correctness and robustness of the implementation. These tests have been generated according to the "model A" described in (Gent & Walsh 1999). The results will be the subject of a future paper. QUBE-LN is publicly available on the web.

[1] In SAT, this happens for randomly generated problems.

References

Abdelwaheb, A., and Basin, D. 2000. Bounded model construction for monadic second-order logics. In *Proc. CAV*, 99–113.

Bayardo, Jr., R. J., and Miranker, D. P. 1996. A complexity analysis of space-bounded learning algorithms for the constraint satisfaction problem. In *Proc. AAAI*, 298–304.

Bayardo Jr., R. J., and Pehoushek, J. D. 2000. Counting models using connected components. In *Proc. AAAI*.

Bayardo, Jr., R. J., and Schrag, R. C. 1997. Using CSP look-back techniques to solve real-world SAT instances. In *Proc. AAAI*, 203–208.

Cadoli, M.; Giovanardi, A.; and Schaerf, M. 1998. An algorithm to evaluate QBFs. In *Proc. AAAI*.

Dechter, R. 1990. Enhancement schemes for constraint processing: Backjumping, learning, and cutset decomposition. *Artificial Intelligence* 41(3):273–312.

Feldmann, R.; Monien, B.; and Schamberger, S. 2000. A distributed algorithm to evaluate quantified boolean formulae. In *Proc. AAAI*.

Freeman, J. W. 1995. *Improvements to propositional satisfiability search algorithms*. Ph.D. Dissertation, University of Pennsylvania.

Gent, I., and Walsh, T. 1999. Beyond NP: the QSAT phase transition. In *Proc. AAAI*, 648–653.

Ginsberg, M. L. 1993. Dynamic backtracking. *Journal of Artificial Intelligence Research* 1:25–46.

Giunchiglia, E.; Narizzano, M.; and Tacchella, A. 2001a. An analysis of backjumping and trivial truth in quantified boolean formulas satisfiability. In *Proc. AI*IA, LNAI 2175*.

Giunchiglia, E.; Narizzano, M.; and Tacchella, A. 2001b. Backjumping for quantified boolean logic satisfiability. In *Proc. IJCAI*, 275–281.

Kleine-Büning, H.; Karpinski, M.; and Flögel, A. 1995. Resolution for quantified boolean formulas. *Information and computation* 117(1):12–18.

Moskewicz, M. W.; Madigan, C. F.; Zhao, Y.; Zhang, L.; and Malik, S. 2001. Chaff: Engineering an Efficient SAT Solver. In *Proc. DAC*.

Plaisted, D., and Greenbaum, S. 1986. A Structure-preserving Clause Form Translation. *Journal of Symbolic Computation* 2:293–304.

Prosser, P. 1993a. Domain filtering can degrade intelligent backjumping search. In *Proc. IJCAI*, 262–267.

Prosser, P. 1993b. Hybrid algorithms for the constraint satisfaction problem. *Computational Intelligence* 9(3):268–299.

Rintanen, J. 1999a. Constructing conditional plans by a theorem prover. *Journal of Artificial Intelligence Research* 10:323–352.

Rintanen, J. 1999b. Improvements to the evaluation of quantified boolean formulae. In *Proc. IJCAI*, 1192–1197.

Zhang, H., and Stickel, M. E. 1996. An efficient algorithm for unit propagation. In *Proc. of the 4th International Symposium on Artificial Intelligence and Mathematics*.