Assessing the impact of Learning Techniques in 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. In (Giunchiglia *et al.* 2002) we have introduced learning as a general purpose technique to improve the performances of decision procedures for Quantified Boolean Formulas (QBFs). We have added learning techniques to QUBE, a state-of-the-art QBF solver. Embedding learning techniques in QUBE proved to be a challenging task, which also led to the definition and the implementation of new pruning strategies. In this paper, we report some experimental results that witness the effectiveness of learning and the associated pruning strategies. Noticeably, QUBE augmented with learning is able to solve instances that were previously out if its reach. On the other hand, the additional complexity of learning does impose some overhead on QUBE. We present results that highlight the presence of such overhead, we discuss its causes, and suggest possible remedies in QUBE.

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. and Schrag 1997)).

We have introduced learning to improve the performances of decision procedures for Quantified Boolean Formulas (QBFs) in (Giunchiglia *et al.* 2002). Due to the particular nature of the satisfiability problem for QBFs, our procedure is able to learn from constraint violations, i.e., *nogood* learning, and to learn from the satisfaction of all constraints, i.e., *good* learning. As noticed in (Giunchiglia *et al.* 2002), learning goods enables to extend Boolean Constraint Propagation (BCP) 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.

We have implemented learning 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, required the development of ad hoc data structures and algorithms.

In this paper we present the results obtained running QUBE on several real-world QBFs, corresponding to planning (Rintanen 1999) and circuit verification (Abdelwaheb and Basin 2000) problems. Some results witness the effectiveness of learning to its fullest extent. The size of the search space explored by QUBE on these instances is orders of magnitude smaller than QUBE without learning, and the performances are also consistently improved. Noticeably, QUBE augmented with learning is able to solve in a few seconds instances that previously were out of its reach. On the other hand, the additional complexity of learning does impose some overhead on QUBE. We present also results that highlight the presence of such overhead, i.e., instances where the search space is not reduced, or where the reduction does not correspond to an improvement in performance. Considering the instances where learning does not seem to pay off, we investigate the causes and discuss possible remedies in QUBE.

The paper is structured as follows. We first review the logic of QBFs, and the ideas behind DLL-based decision procedures for QBFs, including backjumping and learning. Then we present and discuss the results of our experimental analysis. We conclude the paper with some final remarks.

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

¬. 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

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

where

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

$Q_1 x_1 \dots Q_n x_n$ 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 and 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:

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

In (Cadoli *et al.* 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 $\{l, l_1, \dots, l_m\}$ 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 *et al.* 1998) for more details.

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

Backjumping and Learning

CBJ and SBJ

The procedure presented in (Cadoli *et al.* 1998) uses a standard backtracking schema. In (Giunchiglia *et al.* 2001b), we showed how Conflict-directed Backjumping (CBJ) (see, e.g., (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 $\mu = l_1; \dots; l_m$ $(m \geq 0)$ of literals is an *assignment for* φ if for each literal l_i in μ

- l_i is unit, or monotone in $\varphi_{l_1;\dots;l_{i-1}}$; 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 $\mu = l_1; \dots; l_m$, φ_μ is the QBF obtained from φ by

- 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 $\mu = l_1; \dots; l_m$ for φ.

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

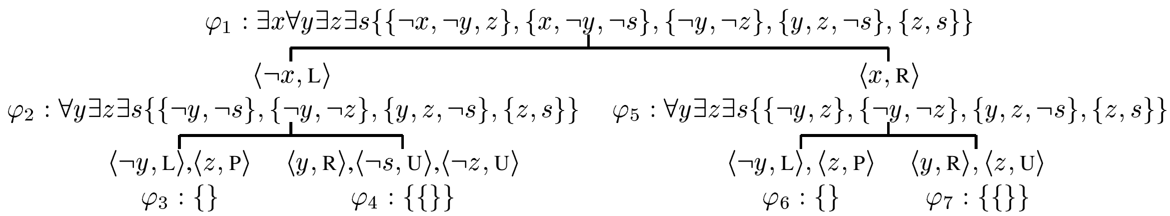

Figure 1: A typical computation tree for (2). U, P, L, R stand for UNIT, PURE, L-SPLIT, R-SPLIT respectively, and have the obvious meaning.

- $\nu \subseteq \{l_1, \ldots, l_m\}$, 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 *et al.* 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 they can be simply added to the set of input clauses. In the case of QBFs, the situation is different, since:

1. we have two types of reasons ("good" and "nogood"), and
2. a reason ν can be added to the set of input clauses only if it is extracted from a *prefix-closed* assignment μ, i.e., for each literal l_i in μ, all the variables at higher levels occur in μ.

Point (2) is rather technical and we do not discuss it in depth; more details can be found in (Giunchiglia *et al.* 2002). Here it suffices to say that the condition on prefix closure is symmetric, i.e. it holds both for nogoods and for goods; 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.

Point (1) implies that an effective implementation of learning requires an efficient handling of both 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. 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

$$Q_1 x_1 \ldots Q_n x_n \langle \Psi, \Phi, \Theta \rangle \qquad (n \geq 0) \tag{3}$$

where

- $Q_1 x_1 \ldots Q_n x_n$ 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

  $$Q_1 x_1 \ldots Q_n x_n (\neg \Psi \vee \Phi) \tag{4}$$

  and

  $$Q_1 x_1 \ldots Q_n x_n (\Phi \wedge \Theta) \tag{5}$$

  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.

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 $m \geq 0$,

Test File	T/F	LV.	$\#\exists x$	$\#\forall x$	# CL.	TIME-BT	NODES-BT	TIME-BJ	NODES-BJ	TIME-LN	NODES-LN
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
T*a.08.01.12	F	3	312	8	3325	1.53	9539	1.54	9539	1.34	3134
T*a.08.01.13	F	3	338	8	3437	4.35	27295	4.37	27295	4.25	8847
T*a.08.05.3	F	3	198	8	3345	4.51	30203	3.90	28619	2.53	9350
T*a.10.01.12	F	3	384	10	11945	6.84	9539	6.89	9539	1.51	3134
T*a.10.01.13	F	3	416	10	12094	19.83	27295	19.34	27295	4.74	8848
T*a.10.01.14	F	3	448	10	12243	59.92	81939	58.01	81939	16.13	25784
T*a.10.01.15	F	3	480	10	12392	180.38	249647	176.61	249647	52.28	77800
T*a.10.01.16	F	3	512	10	12541	579.28	791395	565.30	791395	194.08	242157
T*a.10.01.17	F	3	544	10	12690	> 1200	–	> 1200	–	682.65	768355
T*a.10.05.3	F	3	240	10	12000	36.01	44961	28.67	41117	5.58	14888

Table 1: QuBE results on 28 out of 49 structured problems. On these instances learning helps in reducing the search space as well as the total execution time. Instance names are abbreviated to fit in the table.

  * 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

$$\exists x \forall y \exists z \exists s \langle \{\}, \{\{\neg x, \neg y, z\}, \{x, \neg y, \neg s\}, \{\neg y, \neg z\}, \{y, z, \neg s\}, \{z, s\}\}, \{\}\rangle. \quad (6)$$

With reference to Figure 1, the search proceeds as in the Figure, with the first branch leading to φ_3. Then, 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.

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 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

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
T*6.1.iv.11	F	3	291	3	1046	31.47	567923	7.05	139505	873.73	97900
T*6.1.iv.12	T	3	318	3	1144	13.99	318634	2.74	56387	120.57	42465
T*a.08.01.14	F	3	364	8	3549	12.97	81939	12.90	81939	13.41	25461
T*a.08.01.15	F	3	390	8	3661	38.43	249647	39.38	249647	42.56	76391
T*a.08.05.4	T	3	364	8	3857	0.13	382	0.12	382	0.72	382
T*a.10.01.20	T	3	640	10	13137	8.33	1132	8.47	1132	25.31	1134
T*c.08.01.13	F	3	333	8	1461	1.55	27295	1.62	27295	4.00	8847
T*c.08.01.14	F	3	359	8	1573	4.41	81939	4.65	81939	12.59	25461
T*c.08.01.15	F	3	385	8	1685	12.86	249647	13.57	249647	40.00	76391
T*c.08.05.3	F	3	193	8	1369	1.33	30203	1.30	28619	2.33	9350
T*c.10.01.13	F	3	410	10	1964	2.09	27295	2.14	27295	4.42	8848
T*c.10.01.14	F	3	442	10	2113	6.03	81939	6.23	81939	15.05	25784
T*c.10.01.15	F	3	474	10	2262	17.81	249647	18.47	249647	48.60	77800
T*c.10.01.16	F	3	506	10	2411	54.18	791395	56.34	791395	181.74	242157
T*c.10.01.17	F	3	538	10	2560	168.32	2548959	174.97	2548959	638.56	768355
T*c.10.05.3	F	3	234	10	1870	2.77	44961	2.47	41117	3.93	14888
T*a.10.05.4	T	3	320	10	12685	7.18	1217	6.86	1195	20.98	1169
T*a.10.10.2	T	3	280	10	12840	6.35	1232	6.10	1136	18.15	1123
C*16v.17	T	3	1601	16	8638	3.72	65567	4.28	65567	300.83	66047
C*17v.18	T	3	1803	17	9892	7.72	131105	8.94	131105	716.30	131649

Table 2: QUBE results on 21 out of 49 structured problems. On most of these instances, learning still reduces the search space. However the total execution time of QUBE is always higher than QUBE-BJ.

one is left. Intuitively, in each branch we are computing the reasons corresponding to the assignment in which all the "problematic" literals are moved to the end of the assignment.

We have implemented the above ideas in QUBE, together with *relevance learning of order* n, i.e., given a current assignment μ a reason ν is discarded as soon as more than n literals are in ν and not in μ. QUBE is a state-of-the-art QBF solver (Giunchiglia *et al.* 2001b, Giunchiglia *et al.* 2001a). To assess the impact of learning on QUBE, 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 1999) and formal verification problems (Abdelwaheb and Basin 2000). Our experimental analysis partitions the above instances in three groups:

- instances in which learning reduces the size of the search space and consistently improves the solution time (28 instances in Table 1),
- instances in which learning may still reduce the size of the search space, but never enough to amortize its overhead (21 instances in Table 2), and
- the remaining instances where learning does not bring any advantage, either because they are already easy to solve or because introducing learning does not make them manageable.

In Tables 1 and 2:

- 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 *et al.* 2000).

Looking at Table 1, 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, as it can be seen on some examples in Table 2, 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.

Looking at Table 2 we can see that learning does not always pay-off. For example, on the T*6.1.iv.12 QBF, QuBE-LN performs (much) less nodes than QuBE-BJ (42465 vs 56387), but its performances are dramatically worse (122.99s vs 2.74s). In this regard, it is important to notice that none of these instances is particularly hard even for QuBE-BT. Therefore, the addition of backjunping and learning is probably not bound to bring dramatic benefits on instances of this kind. To have an idea of the learning overhead, consider the results on the C*16v.17 and C*16c.18 examples, shown last in Table 2. 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.

Some profiling done on QuBE-LN using the instances of Table 2 also revealed that:

- in our implementation, the cost of simplifying the set of clauses after assigning a truth value grows proportionally with the size of the clauses database; even with relevance learning enabled, it is not uncommon to maintain a database which is orders of magnitude bigger than the input formula;
- "undoing" the simplifications occurred after a truth assignment costs as much as "doing" the simplifications; therefore, each single truth assignment costs twice as much when it is on a failed existential branch or on a successfull universal branch;
- the cost of forgetting a clause is proportional to its size; although the maximun size of a learnt clause is polinomially bounded by the number of input variables, the number of deletions can be exponential in the size of the input formula, so the total overhead of deletions can be substantial.

We did not assess the impact of memory fragmentation or locality of reference, although they probably play an important role in determining the overall performance of learning. To tackle the above problems we are considering alternative data structures and/or learning schemas, such as those presented for SAT in (Moskewicz *et al.* 2001a).

One final remark concerns the relevance order. Tables 1 and 2 show 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.

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

Conclusions

Experimental evaluation shows that very substantial speed-ups can be obtained using learning in a QBF solver. It also shows that learning has some overhead, and such overhead can overcome its benefits. However, we believe that further improvements can be obtained by working on our implementation, for instance by extending the ideas presented in (Moskewicz *et al.* 2001b) for SAT. In particular, the extension of the "two literal watching" idea, originally due to (Zhang and Stickel 1996), seems a promising direction. This will be the subject of future research.

References

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

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

M. Cadoli, A. Giovanardi, and M. Schaerf. An algorithm to evaluate quantified boolean formulae. In *Proc. AAAI*, 1998.

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

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

Enrico Giunchiglia, Massimo Narizzano, and Armando Tacchella. An analysis of backjumping and trivial truth in quantified boolean formulas satisfiability. In *Proc. AI*IA, LNAI 2175*, 2001.

Enrico Giunchiglia, Massimo Narizzano, and Armando Tacchella. Backjumping for quantified boolean logic satisfiability. In *Proc. IJCAI*, pages 275–281, 2001.

Enrico Giunchiglia, Massimo Narizzano, and Armando Tacchella. Learning for quantified boolean logic satisfiability. In *Proc. of the Eighteenth National Conference on Artificial Intelligence (AAAI'2002)*, 2002. Accepted for publication. To appear.

Matthew W. Moskewicz, Conor F. Madigan, Ying Zhao, Lintao Zhang, and Sharad Malik. Chaff: Engineering an Efficient SAT Solver. In *Proceedings of the 38th Design Automation Conference (DAC'01)*, June 2001.

Matthew W. Moskewicz, Conor F. Madigan, Ying Zhao, Lintao Zhang, and Sharad Malik. Chaff: Engineering an Efficient SAT Solver. In *Proc. DAC*, June 2001.

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

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

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

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

H. Zhang and M. E. Stickel. 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.