The Effect of Nogood Recording in MAC-CBJ SAT Algorithms

Inês Lynce and João Marques-Silva

Technical University of Lisbon,
IST/INESC/CEL, Lisbon, Portugal
{ines,jpms}@sat.inesc.pt

Abstract. Propositional Satisfiability (SAT) solvers have been the subject of remarkable improvements in the last few years. Currently, the most successful SAT solvers share a number of similarities, being based on backtrack search, applying unit propagation, and incorporating a number of additional search reduction techniques. Most, if not all, of the search reduction techniques used by state-of-the-art SAT solvers have been imported from the Constraint Satisfaction Problem (CSP) domain and, most significantly, include forms of backjumping and of nogood recording. This paper proposes to investigate the actual usefulness of these CSP techniques in SAT solvers, with the objective of evaluating the actual role played by each individual technique.

1 Introduction

The areas of Constraint Satisfaction Problem (CSP) and Propositional Satisfiability (SAT) have been the subject of intensive research in recent years, with significant theoretical and practical contributions. In the area of SAT, several highly optimized solvers have been developed [12, 2, 21, 13, 10]. These state-of-the-art SAT solvers can now very easily solve very large, very hard real-world problem instances, which more traditional SAT solvers are totally incapable of. All of these highly effective SAT solvers are based on improvements made to the original Davis-Putnam-Logemann-Loveland (DPLL) backtrack search SAT algorithm [5]. Such improvements range from new search strategies, to new search pruning and reasoning techniques, and to new fast implementations.

Interestingly, the relationship between SAT and CSP has become closer due to an increasing number of mappings between SAT and CSP that have recently been proposed [19, 9]. These different encodings, jointly with the current very efficient SAT solvers, motivate a better understanding of the actual usefulness of the CSP techniques that have been utilized in successful SAT solvers.

Regarding different algorithmic solutions for SAT, and despite the potential theoretical and practical interest of all of them, we believe backtrack search to be the most robust approach for solving hard, structured, real-world instances of SAT. This belief has been amply supported by extensive experimental evidence obtained in recent years [12, 2, 21, 13, 10]. Moreover, the most effective algorithms are complete, and so able to prove what local search is not capable of, i.e. unsatisfiability. Indeed, this is often the objective in a large number of significant SAT-related real-world applications.

Most if not all backtrack search SAT algorithms also incorporate propagation techniques for consistency checking, by applying boolean constraint propagation [20] (which is conceptually similar to maintaining arc consistency (MAC) [17], and equivalent for suitable mappings [19, 9]). Another strategy for reducing the number of searched nodes consists of performing non-chronological jumps in the search tree, skipping portions of the search space that can be shown not to contain a solution. In this context, and whenever a consistency check fails, conflict-directed backjumping (CBJ) [15] enables the search process to safely jump directly to the cause of the conflict.

In addition, state-of-the-art SAT solvers [12, 2, 21, 13, 10] effectively use learning techniques. In these solvers, whenever a conflict (*dead-end*) is reached, a new clause (*nogood*) is recorded to prevent the occurrence of the same conflict again during the subsequent search. Moreover, and from the first SAT solvers that incorporated non-chronological backtracking [12, 2], learning has always been a key component of the search algorithm, where recorded nogoods are used to determine the search point to backtrack to.

Hence, it is certainly relevant to conduct an unbiased evaluation of the isolated usefulness of MAC-CBJ [16] and of learning on a successful SAT solver. Therefore, the objectives of this paper are two-fold. First, to describe the organization of a MAC-CBJ SAT algorithm. Second, to evaluate the effect of learning in this algorithm. For this purpose, we developed a general framework that implements a MAC-CBJ SAT algorithm *with* and *without* nogood recording. Moreover, we evaluate the performance on a representative set of instances, obtained from different real-world problems. This allows us to confirm and extend the preliminary experimental results presented in [1].

The remainder of the paper is organized as follows. Next, we introduce definitions used throughout the other sections. Afterwards, we briefly describe chronological backtrack algorithms for SAT. In Section 4 we overview non-chronological backtracking SAT algorithms, and further relate them with MAC-CBJ. Experimental results are given in Section 5, and finally Section 6 concludes the paper.

2 Definitions

This section introduces the notational framework used throughout the paper, for both CSP and SAT.

2.1 CSP

A CSP consists of a set of variables V and a set of constraints C. Each variable $v \in V$ has a domain of values M_v of size m_v. Each a-ary constraint $c \in C$ restricts a tuple of variables $\langle v_1, ..., v_a \rangle$ to an allowed combination of simultaneous values for the variables in the tuple. In a binary CSP, each constraint is a relation between two variables. Any binary CSP can be associated with a *constraint* *graph*, where the nodes represent variables and each edge links a pair of nodes if and only if there is a constraint on the corresponding variables. A CSP consists of deciding whether there exists an assignment to the variables such that all the constraints are satisfied, i.e. no $c \in C$ is violated.

Arc-consistency checking [11] is commonly used to solve CSP. A state is arc-consistent if every variable has a value in its domain that is consistent with each of the constraints on that variable. Arc consistency can be achieved by successive deletion of values that are inconsistent with some constraint. As values are deleted, other values may become inconsistent because they relied on the deleted values. Arc consistency therefore exhibits a form of *constraint propagation*, as choices are gradually narrowed down. Furthermore, *maintaining arc consistency* (MAC) [17] extends arc consistency by fully maintaining arc consistency during backtrack search.

2.2 SAT

In a SAT problem, propositional variables are denoted $x_1, \ldots, x_n$, and can be assigned truth values 0 (or F) or 1 (or T). The truth value assigned to a variable x is denoted by $\nu(x)$. (When clear from context we use $x = \nu_x$, where $\nu_x \in \{0, 1\}$). A literal l is either a variable x_i or its negation $\neg x_i$. A clause ω is a disjunction of literals and a CNF formula φ is a conjunction of clauses. A clause is said to be *satisfied* if at least one of its literals assumes value 1, *unsatisfied* if all of its literals assume value 0, *unit* if all but one literal assume value 0, and *unresolved* otherwise. Literals with no assigned truth value are said to be *free literals*. A formula is said to be *satisfied* if all its clauses are satisfied, and is *unsatisfied* if at least one clause is unsatisfied. A *truth assignment* A for a formula is a set of assigned variables and their corresponding truth values. The SAT problem consists of deciding whether there exists a truth assignment to the variables such that the formula becomes satisfied.

It will often be simpler to refer to a clause as a set of literals, and to the CNF formula as a set of clauses. Hence, the notation $l \in \omega$ indicates that a literal l is one of the literals of clause ω, whereas the notation $\omega \in \varphi$ indicates that clause ω is one of the clauses of CNF formula φ.

3 Chronological Backtrack SAT Algorithms

Over the years a large number of algorithms have been proposed for SAT, from the original Davis-Putnam procedure [6], to recent backtrack search algorithms [12, 2, 21, 13, 10], among many others.

The vast majority of chronological backtrack (CB) search SAT algorithms build upon the original backtrack search algorithm of Davis, Logemann and Loveland [5]. The backtrack search algorithm is implemented by a *search process* that implicitly enumerates the space of 2^n possible binary assignments to the n problem variables. Each different truth assignment defines a *search path* within the search space. A *decision level* is associated with each variable selection and assignment. The first variable selection corresponds to decision level 1, and the decision level is incremented by 1 for each new decision assignment [1]. When relevant to the context, we use an assignment notation to indicate the decision level at which the variable assignment occurred. Thus, $x = \nu_x@d$ reads as "x is assigned ν_x at decision level d." In addition, and for each decision level, the *unit clause rule* [6] is applied. If a clause is unit, then the sole free literal must be assigned value 1 for the formula to be satisfied. In this case, the value of the literal and of the associated variable are said to be *implied*. The iterated application of the unit clause rule is often referred to as Boolean Constraint Propagation (BCP) [20]. Observe that BCP is conceptually similar to MAC, being equivalent for suitable mappings [19, 9].

In chronological backtracking, the search algorithm keeps track of which decision assignments have been toggled. Given an unsatisfied clause (i.e. a *conflict* or a *dead end*) at decision level d, the algorithm checks whether at the current decision level the corresponding decision variable x has already been toggled. If not, the algorithm erases the variable assignments which are implied by the assignment on x, including the assignment on x, assigns the opposite value to x, and marks decision variable x as toggled. In contrast, if the value of x has already been toggled, the search backtracks to decision level $d - 1$.

4 Non-Chronological Backtrack SAT Algorithms

Recent state-of-the-art SAT solvers [2, 12, 21, 13, 10] utilize different forms of non-chronological backtracking (NCB). Non-chronological backtracking backs up the search tree to one of the identified causes of failure, skipping over irrelevant variable assignments.

For example, let us consider Figure 1, that illustrates non-chronological backtracking for a given CNF formula. Once both x_1 and x_2 are assigned value 0, there are no possible assignments for the remaining variables x_3 and x_4 to satisfy the formula. In this example, chronological backtracking wastes a potentially significant amount of time exploring a useless region of the search space, only to discover, after potentially much effort, that the region does not contain any satisfying assignments.

The forms of non-chronological backtracking used in state-of-the-art SAT solvers can be related with *dependency-directed backtracking* [18], since they are always associated with learning from conflicts. The incorporation of learning consists of the following: for each identified conflict, its causes are identified, and a new clause (called *nogood*) is created to explain and prevent the identified conflicting conditions.

In the next section we address conflict-directed backjumping (CBJ) [15], another form of non-chronological backtracking that does not incorporate learning. Since our main goal is to evaluate the effect of learning in MAC-CBJ SAT algorithms, a brief description of CBJ is essential.

[1] Observe that all the assignments made before the first decision assignment correspond to decision level 0.

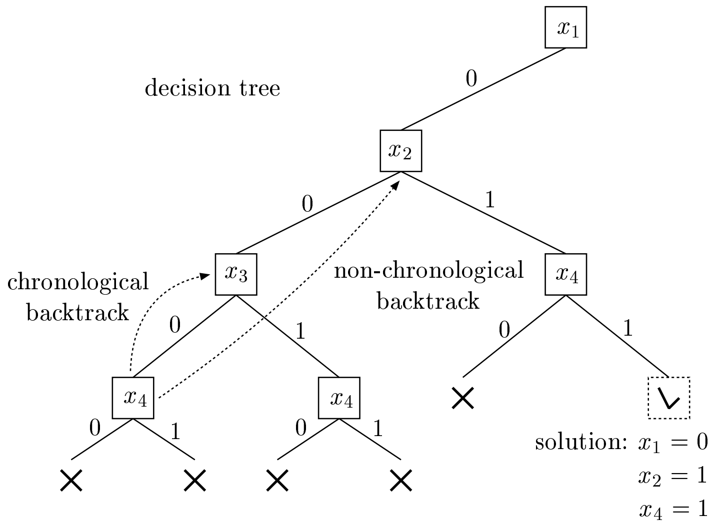


Fig. 1. Non-Chronological Backtracking

4.1 Conflict-Directed Backjumping

Conflict-directed backjumping (CBJ) [15] is the most accurate form of backjumping, and can be considered a combination of Gaschnig's backjumping (BJ) [8] and Dechter's graph-based backjumping (GBJ) [7]. BJ aims performing higher jumps in the search tree, rather than backtracking to the most recent yet untoggled decision variable. Gaschnig's algorithm uses a marking technique that maintains, for each variable v_j, a pointer to a variable v_i with the highest level with which any value of v_j was found to be inconsistent. After a backjump from v_j, and when the domain of variable v_i is wiped out, the search simply backtracks chronologically to v_{i-1}. As an improvement, Dechter's GBJ extracts knowledge about dependencies from the constraint graph. CBJ builds upon this idea and, based on dependencies from the constraints, records the *set* of past variables that failed consistency checks with each variable v. This set is called *conflict set* in [7].

BJ and CBJ behaviors are illustrated with Figure 2. Let us consider a CSP with variables A, B, C, D, E and F. Suppose that we instantiate these variables in alphabetically order, and also check backwards to ensure consistency. Assume that a value tried for D conflicts with A, and a value for F conflicts with D. In Gaschnig's BJ the search will jump back from F to D and then to C, whereas in CBJ will jump from E to C and then from C to A.

4.2 Learning and Conflict-Directed Backjumping

Learning can be combined with CBJ. When each identified conflict is analyzed, its causes identified, and a nogood is recorded to explain and prevent the iden-

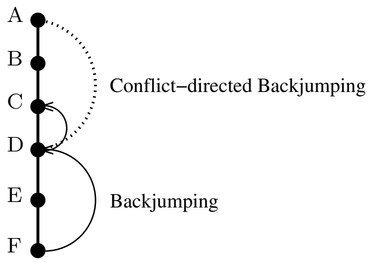


Fig. 2. Backjumping and Conflict-directed backjumping

tified conflicting conditions from occurring again during the subsequent search. Moreover, the newly recorded nogood is then used to compute the backtrack point as the *most recent* decision assignment from all the decision assignments represented in the recorded nogood.

For implementing learning techniques common to some of the most competitive backtrack search SAT algorithms, it is necessary to properly *explain* the truth assignments to the propositional variables that are implied by the clauses of the CNF formula. For example, let $x = v_x$ be a truth assignment implied by applying the unit clause rule to a unit clause clause ω. Then the explanation for this assignment is the set of assignments associated with the remaining literals of ω, which are assigned value 0. For example, let $\omega = (x_1 \vee \neg x_2 \vee x_3)$ be a clause of a CNF formula φ, and assume the truth assignments $\{x_1 = 0, x_3 = 0\}$. For clause ω to be satisfied we must necessarily have $x_2 = 0$. Hence, we say that the *antecedent assignment* of x_2, denoted as $A(x_2)$, is defined as $A(x_2) = \{x_1 = 0, x_3 = 0\}$.

In addition, in order to explain other NCB-related concepts, we shall often analyze the sequences of implied assignments generated by BCP. These sequences are captured by a directed acyclic *implication graph* I defined as follows:

1. Each vertex in I corresponds to a variable assignment $x = \nu(x)$.
2. The predecessors of vertex $x = \nu(x)$ in I are the antecedent assignments $A(x)$ corresponding to the unit clause ω that led to the implication of x. The directed edges from the vertices in $A(x)$ to vertex $x = \nu(x)$ are all labeled with ω. Vertices that have no predecessors correspond to decision assignments.
3. Special conflict vertices are added to I to indicate the occurrence of conflicts. The predecessors of a conflict vertex κ correspond to variable assignments that force a clause ω to become unsatisfied and are viewed as the antecedent assignment $A(\kappa)$. The directed edges from the vertices in $A(\kappa)$ to κ are all labeled with ω.

We illustrate nogood recording with the example of Figure 3. A subset of the CNF formula is shown, and we assume that the current decision level is 6, corresponding to the decision assignment $x_1 = 1$. This assignment yields a conflict κ involving clause ω_6. By inspection of the implication graph, we can

Current Truth Assignment: $\{x_9 = 0@1, x_{10} = 0@3, x_{11} = 0@3, x_{12} = 1@2, x_{13} = 1@2, ...\}$
Current Decision Assignment: $\{x_1 = 1@6\}$

$\omega_1 = (\neg x_1 \vee x_2)$
$\omega_2 = (\neg x_1 \vee x_3 \vee x_9)$
$\omega_3 = (\neg x_2 \vee \neg x_3 \vee x_4)$
$\omega_4 = (\neg x_4 \vee x_5 \vee x_{10})$
$\omega_5 = (\neg x_4 \vee x_6 \vee x_{11})$
$\omega_6 = (\neg x_5 \vee \neg x_6)$
$\omega_7 = (x_1 \vee x_7 \vee \neg x_{12})$
$\omega_8 = (x_1 \vee x_8)$
$\omega_9 = (\neg x_7 \vee \neg x_8 \vee \neg x_{13})$
...

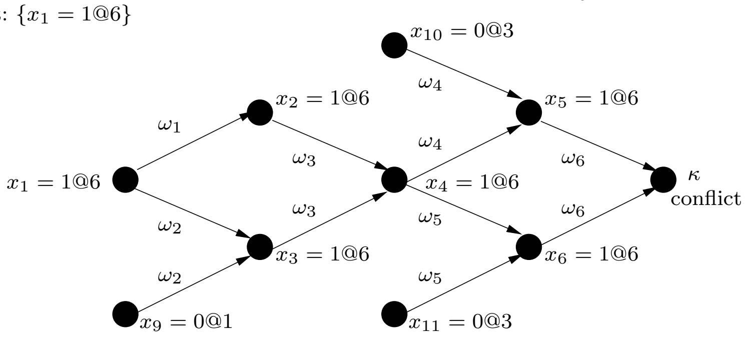


Fig. 3. Example of conflict diagnosis with nogood recording

readily conclude that a *sufficient condition* for this conflict to be identified is,

$$(x_{10} = 0) \wedge (x_{11} = 0) \wedge (x_9 = 0) \wedge (x_1 = 1) \tag{1}$$

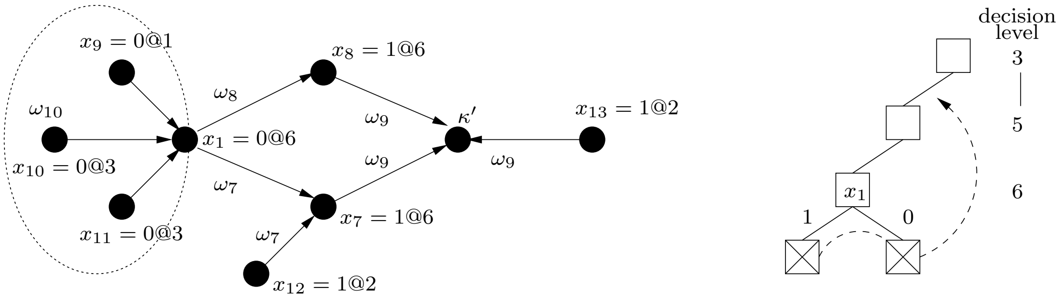
By creating clause $\omega_{10} = (x_{10} \vee x_{11} \vee x_9 \vee \neg x_1)$ we prevent the same set of assignments from occurring again during the subsequent search.

In order to illustrate non-chronological backtracking based on nogood recording, let us now consider the example of Figure 4, which continues the example in Figure 3, after recording clause $\omega_{10} = (x_{10} \vee x_{11} \vee x_9 \vee \neg x_1)$. At this stage BCP implies the assignment $x_1 = 0$ because clause ω_{10} becomes unit at decision level 6. By inspection of the CNF formula (see Figure 3), we can conclude that clauses ω_7 and ω_8 imply the assignments shown, and so we obtain a conflict κ' involving clause ω_9. By creating clause $\omega_{11} = (\neg x_{13} \vee \neg x_{12} \vee x_{11} \vee x_{10} \vee x_9)$ we prevent the same conflicting conditions from occurring again. It is straightforward to conclude that even though the current decision level is 6, all assignments directly involved in the conflict are associated with variables assigned at decision levels less than 6, the highest of which being 3. Hence we can backtrack immediately to decision level 3.

4.3 Nogoods Deletion Policy

Unrestricted nogood recording can in some cases be impractical. Recorded nogoods consume memory and repeated recording of nogoods can eventually lead to the exhaustion of the available memory. Observe that the number of recorded nogoods grows with the number of conflicts; in the worst case, such growth can be *exponential* in the number of variables. Furthermore, large recorded nogoods are known for not being particularly useful for search pruning purposes [12].

As a result, there are three main solutions for guaranteeing the worst case growth of the recorded nogoods to be *polynomial* in the number of variables:



(a) Conflicting implication sequence (a) Decision tree

Fig. 4. Computing the backtrack decision level

1. We may consider *n-order learning*, that records only nogoods with n or fewer literals [7].
2. Nogoods can be temporarily recorded while they either imply variable assignments or are unit clauses, being discarded as soon as the number of unassigned literals is greater than an integer m [2]. This technique is named *m-size relevance-based learning.*
3. Nogoods with a size less than a threshold k are kept during the subsequent search, whereas larger nogoods are discarded as soon as the number of unassigned literals is greater than one [12]. We refer to this technique as *k-bounded learning.*

Observe that we can use k-bounded learning together with m-size relevance-based learning. The search algorithm is organized so that all recorded clauses of size no greater than k are kept and larger clauses are deleted only after m literals have become unassigned.

5 Experimental Results

In this section we present the obtained experimental results. We start by describing the experimental setup that has been used to obtain the different results. Then we analyze the results for the MAC-CB SAT algorithm, the MAC-CBJ SAT algorithm and the MAC-CBJ SAT algorithm with nogood recording [2].

5.1 Experimental Setup

In order to experimentally evaluate the different algorithms, in a controlled experiment that ensures that only specific differences are evaluated, a dedicated SAT solving framework is needed. Consequently, we developed the JQUEST SAT framework, a Java implementation that can be used to conduct unbiased experimental evaluations of SAT algorithms and techniques.

[2] Observe that in this context MAC corresponds to BCP.

In order to perform this comparison using the JQUEST SAT solver, instances were selected from several classes of instances (see Table 1)[3]. In all cases, the problem instances chosen can be solved with several thousands of decisions by the most effective solvers, usually taking a few tens of seconds, thus being significantly hard [4]. For this reason, different algorithms can result in significant variations on the time required for solving a given instance. In addition, we should also observe that the problem instances selected are intended to be *representative*, since each resembles, in terms of hardness for SAT solvers, the typical instance in each class of problem instances.

Table 1. Example Instances

Application Domain	Selected Instance	# Variables	#Clauses	Satisfiable?
Circuit Testing (Dimacs)	bf0432-079	1044	3685	N
	ssa2670-141	4843	2315	N
Inductive Inference(Dimacs)	ii16b2	1076	16121	Y
	ii16e1	1245	14766	Y
Parity Learning(Dimacs)	par16-1-c	317	1264	Y
	par16-4	1015	3324	Y
Graph Colouring	flat200-39	600	2237	Y
	sw100-49	500	3100	Y
Quasigroup	qg3-08	512	10469	Y
	qg5-09	729	28540	N
Blocks World	2bitadd_12	708	1702	Y
	4blocksb	410	24758	Y
Planning-Sat	logistics.a	828	6718	Y
	bw_large.c	3016	50457	Y
Planning-Unsat	logistics.c	1027	9507	N
	bw_large.b	920	11491	N
Bounded Model Checking	barrel5	1407	5383	N
	queueinvar16	1168	6496	N
	longmult6	2848	8853	N
Superscalar Processor Verification	dlx2_aa	490	2804	N
	dlx2_cc_a_bug17	4847	39184	Y
	2dlx_cc_mc_ex_bp_f2_bug006	4824	48215	Y
	2dlx_cc_mc_ex_bp_f2_bug010	5754	60689	Y
Data Encryption Standard	cnf-r3-b2-k1.1	152608	17857	Y
	cnf-r3-b4-k1.2	939040	35963	Y

[3] All the instances are available from `http://www.lri.fr/~simon/satex/satex.php3` (Sat-Ex web site), with the exception of the *superscalar processor verification* instances.

[4] The *Data Encryption Standard* instances have significantly more variables and clauses then the other instances, even though some of them are known to be trivially removed.

For the results shown a P-IV@1.7 GHz Linux machine with 1 GByte of physical memory was used. The Java Virtual Machine used was SUN's HotSpot JVM for JDK1.4. The CPU time was limited to 1500 seconds.

Table 2. CPU Time (in seconds)

Instance	**MAC CB**	**MAC CBJ**	**MAC-CBJ+ng**						
			`+ng0`	`+ng5`	`+ng10`	`+ng20`	`+ng50`	`+ng100`	`+ngAll`
bf0432-079	—	41.74	5.18	2.78	2.97	2.70	1.53	1.44	1.45
ssa2670-141	—	—	1.21	0.87	0.81	0.52	0.55	0.54	0.56
ii16b2	—	—	—	—	857.61	302.63	158.63	141.41	141.05
ii16e1	—	—	20.58	26.75	20.40	12.65	12.89	15.96	11.86
par16-1-c	65.92	77.06	19.91	14.75	16.07	16.78	18.19	18.08	18.13
par16-4	14.88	20.59	11.51	8.49	8.77	9.34	7.16	7.20	7.14
flat200-39	8.44	8.75	97.35	255.04	85.19	114.05	67.55	67.00	67.19
sw100-49	—	—	1.94	13.48	1.26	2.18	0.73	0.74	0.71
qg3-08	2.29	2.65	0.86	0.88	0.91	1.00	1.07	1.30	1.32
qg5-09	13.28	8.61	1.35	1.29	1.06	1.17	1.16	1.21	1.15
2bitadd_12	—	—	—	—	—	—	87.68	50.74	50.93
4blocksb	—	—	31.23	30.25	39.62	29.66	16.34	20.33	31.75
logistics.a	—	—	2.98	1.87	1.62	1.65	1.61	1.63	1.68
bw_large.c	—	—	76.03	55.13	36.41	38.37	43.71	38.03	38.06
logistics.c	—	—	26.88	7.24	4.60	15.40	10.22	10.23	10.25
bw_large.b	8.27	4.78	1.60	0.59	0.61	0.64	0.61	0.62	0.62
barrel5	99.49	132.37	171.94	279.31	36.35	19.80	23.43	24.00	21.99
queueinvar16	—	—	23.36	21.39	15.74	15.48	8.05	8.08	8.12
longmult6	—	—	27.66	23.63	26.20	29.16	32.32	31.74	32.02
dlx2_aa	—	—	54.74	36.21	37.96	9.80	6.43	6.62	6.60
dlx2_cc_a_bug17	—	—	430.31	—	—	500.25	220.06	6.48	6.54
2dlx_..._bug006	—	—	17.29	14.32	3.87	2.27	2.27	2.23	2.22
2dlx_..._bug010	—	—	3.32	4.13	3.83	5.31	4.55	2.03	1.93
cnf-r3-b2-k1.1	802.90	19.37	3.05	3.13	2.39	2.58	2.40	2.16	3.90
cnf-r3-b4-k1.2	405.98	12.62	5.42	5.39	5.48	5.12	4.89	4.45	3.91

5.2 MAC-CBJ and Nogood Recording

The first table of results (Table 2) shows the CPU time required to solve each problem instance [5]. For the algorithms considered: **MAC-CB** denotes the chronological backtracking search SAT algorithm that maintains arc consistency, **MAC-CBJ** denotes the MAC-CBJ SAT algorithm and **MAC-CBJ+ng** denotes the

[5] Instances that were not solved in the allowed CPU time are marked with —.

MAC-CBJ SAT algorithm with nogood recording. Moreover, a variety of nogood deletion policies were considered, depending on the value of k, where k defines the *k-bounded learning* procedure used (see Section 4.3). For instance, `+ng10` means that recorded nogoods with size greater than `10` are deleted as soon as they become unresolved, whereas `+ngAll` means that all the recorded nogoods are kept.

Table 2 reveals interesting trends, and several conclusions can be drawn:

- Clearly, MAC-CB and MAC-CBJ have in general similar behavior (except for *bf0432-079* and *data encryption standard* instances, that only MAC-CBJ is able to solve).
- The MAC-CBJ+ng algorithms are in general significantly more efficient than the other algorithms. Indeed, for almost all the instances MAC-CBJ+ng achieves remarkable improvements, when compared with MAC-CB or with MAC-CBJ. Instances *flat200-39* and *barrel5* are the only exceptions. (For instance *barrel5*, this is only true for MAC-CBJ+ng with small values of k.)
- Some of the instances that are not solved by MAC-CBJ in the allowed CPU time (e.g *ii16b2* and *dlx2_cc_a_bug17*), also need a significant amount of time to be solved by *k-bounded learning* with a small value of k.
- For instance *flat200-39*, recorded nogoods result in an additional search effort to find a solution.
- From a practical perspective, unrestricted nogood recording is *not* necessarily a bad approach.

It is interesting to observe that the *uselessness* of CBJ-related algorithms w.r.t. CB-related algorithms has been experienced in the past [14] [3]. CBJ, when applied jointly with a domain filtering procedure (e.g. MAC) and an accurate variable ordering heuristic, has been considered an expensive approach that almost always slows down the search, even if it saves a few constraint checks [6].

Table 3 gives the results for the number of searched nodes, for each instance and for the different configurations. It is plain from the results that MAC-CB and MAC-CBJ in general need to search more nodes to find a solution. This can be explained by the effect of the recorded nogoods. Besides explaining an identified conflict, nogoods are often re-used, either for yielding conflicts or for implying variable assignments, introducing significant pruning in the search tree. Moreover, other conclusions can be established from the results on the searched nodes:

- For the *par* instances, there are none or just a few backjumps during the search. This fact explains why MAC-CB and MAC-CBJ have the same or an approximate number of search nodes for these instances.
- Only small-size nogoods are recorded for instances *logistics.a*, *bw_large.b* and *qg5-09*. This is the reason why the search has the same number of nodes for increasing values of k.

[6] Even though different conclusions have been obtained for instances of specific classes [4].

– Usually more recorded nogoods indicate less searched nodes and less time needed to find a solution (even though the reduction in the number of nodes is more significant than the reduction in the amount of time, due to the overhead introduced by the management of nogoods).

Table 3. Searched nodes

Instance	**MAC CB**	**MAC CBJ**	**MAC-CBJ+ng** +ng0	+ng5	+ng10	+ng20	+ng50	+ng100	+ngAll
bf0432-079	—	98824	3939	1950	2010	1600	1168	1188	1188
ssa2670-141	—	—	2882	1988	1480	806	736	698	698
ii16b2	—	—	—	—	101451	32923	12188	10573	10573
ii16e1	—	—	20872	24714	17271	13869	8117	8442	7869
par16-1-c	52800	52800	11307	7249	7524	5255	5364	5364	5364
par16-4	10673	10246	4157	2676	2745	2467	1919	1919	1919
flat200-39	6286	5427	139264	287983	51308	40888	26428	25738	25738
sw100-49	—	—	4596	44247	2370	3748	1450	1450	1450
qg3-08	617	703	220	220	242	214	222	282	282
qg5-09	1578	1547	373	329	318	337	337	337	337
2bitadd_12	—	—	—	—	—	—	21244	11238	11238
4blocksb	—	—	5559	5007	6205	5009	2618	2491	3363
logistics.a	—	—	32872	16999	14899	15185	15185	15185	15185
bw_large.c	—	—	6137	5132	2763	2878	3000	2783	2783
logistics.c	—	—	55721	18839	14520	16444	15441	15441	15441
bw_large.b	1431	1112	293	128	195	195	195	195	195
barrel5	24727	24664	90115	141953	14684	8731	10396	12315	5985
queueinvar16	—	—	45053	41510	24842	19557	8460	8506	8083
longmult6	—	—	7407	5482	5666	5507	5019	4729	4725
dlx2_aa	—	—	319120	204995	208725	21036	10062	10035	10035
dlx2_cc_a_bug17	—	—	446626	—	—	212816	85713	3383	3383
2dlx_..._bug006	—	—	32297	25259	7775	3288	3227	3123	3123
2dlx_..._bug010	—	—	10086	19002	14358	13229	8533	3547	3522
cnf-r3-b2-k1.1	219037	6273	1168	1138	872	942	825	667	1221
cnf-r3-b4-k1.2	57843	2216	1011	1012	1010	943	910	776	729

Overall, the effect of nogood recording is clear, and in general dramatic. The results clearly indicate that nogood recording is an essential component of current state-of-the-art SAT solvers. Nevertheless, the actual best value for k is not clear, and subject of additional research.

Finally, we present another table of results (Table 4), but only for the MAC-CB and MAC-CBJ algorithms. The only difference regarding the previous results is the variable ordering heuristic used. For the first results (Tables 2 and 3), we have applied the variable selection heuristic VSIDS (Variable State Independent

Decaying Sum) [13]. It selects the literal that appears most frequently over all clauses, which means that the metrics only have to be updated when a new recorded clause is created.

Table 4. Time and nodes for DLIS

Instance	Time		Nodes	
	MAC-CB	**MAC-CBJ**	**MAC-CB**	**MAC-CBJ**
bf0432-079	—	437.12	—	275336
ssa2670-141	—	363.53	—	244848
ii16b2	1220.50	630.05	53667	37141
ii16e1	—	30.32	—	3277
par16-1-c	41.15	39.56	8212	7906
par16-4	130.45	132.49	13786	13556
flat200-39	45.26	8.91	9437	2756
sw100-49	—	—	—	—
qg3-08	323.92	184.29	35120	30943
qg5-09	—	—	—	—
2bitadd_12	—	49.79	—	51900
4blocksb	—	—	—	—
logistics.a	—	—	—	—
bw_large.c	—	—	—	—
logistics.c	—	—	—	—
bw_large.b	49.31	17.46	1794	986
barrel5	379.46	411.72	25624	25731
queueinvar16	—	—	—	—
longmult6	—	—	—	—
dlx2_aa	—	—	—	—
dlx2_cc_a_bug17	—	—	—	—
2dlx_..._bug006	—	—	—	—
2dlx_..._bug010	0.63	0.61	718	711
cnf-r3-b2-k1.1	—	—	—	—
cnf-r3-b4-k1.2	—	—	—	—

Given that MAC-CB and MAC-CBJ do not record nogoods, for these algorithms VSIDS corresponds to SLIS (Static Largest Individual Sum). SLIS is a heuristic that selects the literal that appears most frequently in the original clauses; in this case, the metrics are not dynamically changed during the search.

For the last table of results we used the DLIS (Dynamic Largest Individual Sum of literals) heuristic. The intuition is that MAC-CB(J) cooperates poorly with a simple heuristic such as SLIS. On the contrary, VSIDS cooperates effectively with MAC-CBJ+ng, because learning allows correcting early wrong variable orderings. For this reason, we decided to experiment DLIS, a more elaborated heuristic. The obtained results are significantly better than those obtained with SLIS (namely for MAC-CBJ), but are still far from being competitive with MAC-CBJ+ng results.

6 Conclusions and Future Directions

In this paper we address the use of MAC-CBJ in SAT algorithms. In addition, we evaluate the effect of nogood recording in MAC-CBJ SAT algorithms, and further analyze the effect of different nogood deletion policies. Given the experimental results, obtained for representative instances from several classes of problem instances, we conclude that nogood recording is crucial for competitive SAT algorithms. In addition, the results strongly suggest that backjumping techniques are not enough *per se*.

Moreover, we believe that CSP algorithms may also improve their performance by applying both jumping and learning. Interestingly, backjumping techniques and learning have their roots in Truth Maintenance Systems [18] but have been extensively in CSP [8, 7, 15]; nevertheless, constraint programming technology appears not to exploit it.

Future research work will extend the results of this paper by considering alternative approaches with the goal of optimizing SAT solvers. It is well-known that state-of-the-art SAT solvers use nogood recording. On the other hand, MAC-CBJ does not incorporate learning, but does consider conflict sets. Hence, we can envision an algorithm that explores the advantages of CBJ to compensate the disadvantages of nogood recording. This algorithm can apply nogood recording, but use conflict sets to avoid recording large nogoods that must be eventually deleted.

Acknowledgments

We would like to thank Patrick Prosser for the insightful discussions we had on CBJ. This work is partially supported by the European research project IST-2001-34607 and by Fundação para a Ciência e Tecnologia under research projects PRAXIS/C/EEI/11249/98 and POSI/34504/CHS/2000.

References

1. R. Bayardo Jr. and R. Schrag. Using CSP look-back techniques to solve exceptionally hard SAT instances. In *Proceedings of the International Conference on Principles and Practice of Constraint Programming*, pages 46–60, 1996.
2. R. Bayardo Jr. and R. Schrag. Using CSP look-back techniques to solve real-world SAT instances. In *Proceedings of the National Conference on Artificial Intelligence*, pages 203–208, July 1997.
3. C. Bessière and J. C. Régin. MAC and combined heuristics: two reasons to forsake FC (and CBJ?) on hard problems. In *Proceedings of the International Conference on Principles and Practice of Constraint Programming*, pages 61–75, 1996.

4. X. Chen and P. van Beek. Conflict-directed backjumping revisited. *Journal of Artificial Intelligence Research*, 14:53–81, 2001.
5. M. Davis, G. Logemann, and D. Loveland. A machine program for theorem-proving. *Communications of the Association for Computing Machinery*, 5:394–397, July 1962.
6. M. Davis and H. Putnam. A computing procedure for quantification theory. *Journal of the Association for Computing Machinery*, 7:201–215, July 1960.
7. R. Dechter. Enhancement schemes for constraint processing: backjumping, learning, and cutset decomposition. *Artificial Intelligence*, 41(3):273–312, January 1990.
8. J. Gaschnig. *Performance Measurement and Analysis of Certain Search Algorithms*. PhD thesis, Carnegie-Mellon University, Pittsburgh, PA, May 1979.
9. I. Gent. Arc consistency in SAT. Technical Report APES-39-2002, APES Report, January 2002.
10. E. Goldberg and Y. Novikov. BerkMin: a fast and robust sat-solver. In *Proceedings of the Design and Test in Europe Conference*, March 2002.
11. A. K. Mackworth. Consistency in networks of relations. *Artificial Intelligence*, 8(1):99–118, 1977.
12. J. P. Marques-Silva and K. A. Sakallah. GRASP: A new search algorithm for satisfiability. In *Proceedings of the ACM/IEEE International Conference on Computer-Aided Design*, pages 220–227, November 1996.
13. M. Moskewicz, C. Madigan, Y. Zhao, L. Zhang, and S. Malik. Engineering an efficient SAT solver. In *Proceedings of the Design Automation Conference*, pages 530–535, June 2001.
14. P. Prosser. Domain filtering can degrade intelligent backjumping search. In *Proceedings of the International Joint Conference on Artificial Intelligence*, pages 262–267, 1993.
15. P. Prosser. Hybrid algorithms for the constraint satisfaction problems. *Computational Intelligence*, 9(3):268–299, August 1993.
16. P. Prosser. MAC-CBJ: maintaining arc consistency with conflict-directed backjumping. Technical Report 177, University of Strathclyde, Glasgow, Scotland, May 1995.
17. D. Sabin and E. Freuder. Contradicting conventional wisdom in constraint satisfaction. In *Proceedings of the European Conference on Artificial Intelligence*, pages 125–129, 1994.
18. R. M. Stallman and G. J. Sussman. Forward reasoning and dependency-directed backtracking in a system for computer-aided circuit analysis. *Artificial Intelligence*, 9:135–196, October 1977.
19. T. Walsh. SAT *v* CSP. In *Proceedings of the International Conference on Principles and Practice of Constraint Programming*, pages 441–456, September 2000.
20. R. Zabih and D. A. McAllester. A rearrangement search strategy for determining propositional satisfiability. In *Proceedings of the National Conference on Artificial Intelligence*, pages 155–160, 1988.
21. H. Zhang. SATO: An efficient propositional prover. In *Proceedings of the International Conference on Automated Deduction*, pages 272–275, July 1997.