Nogood Recording from Restarts

Christophe Lecoutre, Lakhdar Sais, Sébastien Tabary, and Vincent Vidal

CRIL (Centre de Recherche en Informatique de Lens)
CNRS FRE 2499
rue de l'université, SP 16
62307 Lens cedex, France
{lecoutre,sais,tabary,vidal}@cril.univ-artois.fr

Abstract. In this paper, nogood recording is investigated for CSP within the randomization and restart framework. Our goal is to avoid the same situations to occur from one run to the next one. More precisely, nogoods are recorded when the current cutoff value is reached, i.e. before restarting the search algorithm. Such a set of nogoods is extracted from the last branch of the current search tree and managed using the structure of watched literals originally proposed for SAT. Interestingly, the number of nogoods recorded before each new run is bounded by the length of the last branch of the search tree. As a consequence, the total number of recorded nogoods is polynomial in the number of restarts. Experiments over a wide range of CSP instances demonstrate the effectiveness of this approach.

1 Introduction

Nogood recording (or learning) has been suggested as a technique to enhance CSP (Constraint Satisfaction Problem) solving in [9]. The principle is to record a nogood whenever a conflict occurs during a backtracking search. Such nogoods can then be exploited later to prevent the exploration of useless parts of the search tree. The first experimental results obtained with learning were given in the early 90's [9, 13, 27].

Contrary to CSP, the recent impressive progress in SAT (Boolean Satisfiability Problem) has been achieved using nogood recording (clause learning) under a randomization and restart policy enhanced with a very efficient lazy data structure [24]. Indeed, the interest of clause learning has arisen with the availability of large instances (encoding practical applications) which contain some structures and exhibit heavy-tailed phenomenon. Learning in SAT is a typical successful technique obtained from the cross fertilization between CSP and SAT: nogood recording [9] and conflict directed backjumping [25] have been introduced for CSP and later imported into SAT solvers [2, 21].

Recently, a generalization of nogoods, as well as an elegant learning method, have been proposed in [18, 19] for CSP. While standard nogoods correspond to variable assignments, generalized nogoods also involve value refutations. These generalized nogoods benefit from nice features. For example, they can compactly capture large sets of standard nogoods and are proved to be more powerful than standard ones to prune the search space.

As the set of nogoods that can be recorded might be of exponential size, one needs to achieve some restrictions. For example, in SAT, learned nogoods are not minimal

and are limited in number using the First Unique Implication Point (First UIP) concept. Different variants have been proposed (e.g. relevance bounded learning [2]), all of them attempt to find the best trade-off between the overhead of learning and performance improvements. Consequently, the recorded nogoods can not lead to a complete elimination of redundancy in search trees. An original alternative [29] to combine search scattering and redundancy avoidance involves performing random jumps in the search space. It is particularly relevant when an allotted time is given.

In this paper, nogood recording is investigated within the randomization and restart framework. The principle of our approach is to learn nogoods from the last branch of the search tree before a restart, discarding already explored parts of the search tree in subsequent runs. Roughly speaking, we manage nogoods by introducing a global constraint with a dedicated filtering algorithm which exploits watched literals [24]. The worst-case time complexity of this propagation algorithm is O($n^2\gamma$) where n is the number of variables and γ the number of recorded nogoods. Besides, we know that γ is at most $nd\rho$ where d is the greatest domain size and ρ is the number of restarts already performed.

This approach, so-called *nogood recording from restarts*, can be seen as the CSP adaptation of the *search signature* technique [1] introduced for SAT. Indeed, this technique involves recording the explanations (as clauses) of the search path before restarting, while discarding all clauses inferred (if any) during the last run. *Nogood recording from restarts* presents some interesting features. First, when search is stopped before finding a solution, one can run later the CSP solver with the guarantee of not exploring the same portion of the search space. Secondly, it can be used as a complementary approach of the classical learning schemes which extract and record nogoods each time a conflict occurs.

2 Technical Background

A Constraint Network (CN) P is a pair $(\mathscr{X}, \mathscr{C})$ where $\mathscr{X}$ is a set of n variables and $\mathscr{C}$ a set of e constraints. Each variable $X \in \mathscr{X}$ has an associated domain, denoted *dom*(X), which contains the set of values allowed for X. Each constraint $C \in \mathscr{C}$ involves a subset of variables of $\mathscr{X}$, denoted *vars*(C), and has an associated relation, denoted *rel*(C), which contains the set of tuples allowed for *vars*(C).

A solution to a CN is an assignment of values to all the variables such that all the constraints are satisfied. A CN is said to be satisfiable iff it admits at least one solution. The Constraint Satisfaction Problem (CSP) is the NP-complete task of determining whether a given CN is satisfiable. A CSP instance is then defined by a CN, and solving it involves either finding one (or more) solution or determining its unsatisfiability. To solve a CSP instance, one can modify the CN by using inference or search methods [10].

The backtracking algorithm (BT) is a central algorithm for solving CSP instances. It performs a depth-first search in order to instantiate variables and a backtrack mechanism when dead-ends occur. Many works have been devoted to improve its forward and backward phases by introducing look-ahead and look-back schemes [10]. Today, MAC [26] is the (look-ahead) algorithm considered as the most efficient generic approach

to solve CSP instances. It maintains a property called Arc Consistency (AC) during search. When mentioning MAC, it is important to indicate which branching scheme is employed. Indeed, it is possible to consider binary (2-way) branching or non binary (d-way) branching. These two schemes are not equivalent as it has been shown that binary branching is more powerful (to refute unsatisfiable instances) than non-binary branching [17]. With binary branching, at each step of search, a pair (X,v) is selected where X is an unassigned variable and v a value in dom(X), and two cases are considered: the assignment $X = v$ and the refutation $X \neq v$. The MAC algorithm (using binary branching) can then be seen as building a binary tree. Classically, MAC always starts by assigning variables before refuting values. Generalized Arc Consistency (GAC) (e.g. [4]) extends AC to non binary constraints, and MGAC is the search algorithm that maintains GAC.

Although sophisticated look-back algorithms such as CBJ (Conflict Directed Backjumping) [25] and DBT (Dynamic Backtracking) [14] exist, it has been shown [3, 5, 20] that MAC combined with a good variable ordering heuristic often outperforms such techniques.

3 Reduced nld-Nogoods

From now on, we will consider a search tree built by a backtracking search algorithm based on the 2-way branching scheme (e.g. MAC), positive decisions being performed first. Each branch of the search tree can then be seen as a sequence of positive and negative decisions, defined as follows:

Definition 1. *Let $P = (\mathscr{X}, \mathscr{C})$ be a CN and (X,v) be a pair such that $X \in \mathscr{X}$ and $v \in dom(X)$. The assignment $X = v$ is called a positive decision whereas the refutation $X \neq v$ is called a negative decision. $\neg(X = v)$ denotes $X \neq v$ and $\neg(X \neq v)$ denotes $X = v$.*

Definition 2. *Let $\Sigma = \langle \delta_1, \ldots, \delta_i, \ldots, \delta_m \rangle$ be a sequence of decisions where δ_i is a negative decision. The sequence $\langle \delta_1, \ldots, \delta_i \rangle$ is called a nld-subsequence (negative last decision subsequence) of Σ. The set of positive and negative decisions of Σ are denoted by $pos(\Sigma)$ and $neg(\Sigma)$, respectively.*

Definition 3. *Let P be a CN and Δ be a set of decisions. $P|_\Delta$ is the CN obtained from P s.t., for any positive decision ($X = v$) $\in \Delta$, dom(X) is restricted to $\{v\}$, and, for any negative decision ($X \neq v$) $\in \Delta$, v is removed from dom(X).*

Definition 4. *Let P be a CN and Δ be a set of decisions. Δ is a nogood of P iff $P|_\Delta$ is unsatisfiable.*

From any branch of the search tree, a nogood can be extracted from each negative decision. This is stated by the following property:

Proposition 1. *Let P be a CN and Σ be the sequence of decisions taken along a branch of the search tree. For any nld-subsequence $\langle \delta_1, \ldots, \delta_i \rangle$ of Σ, the set $\Delta = \{\delta_1, \ldots, \neg\delta_i\}$ is a nogood of P (called nld-nogood)*[1].

[1] The notation $\{\delta_1, \ldots, \neg\delta_i\}$ corresponds to $\{\delta_j \in \Sigma \mid j < i\} \cup \{\neg\delta_i\}$ reduced to $\{\neg\delta_1\}$ when $i = 1$.

Proof. As positive decisions are taken first, when the negative decision δ_i is encountered, the subtree corresponding to the opposite decision $\neg\delta_i$ has been refuted. □

These nogoods contain both positive and negative decisions and then correspond to the definition of generalized nogoods [12, 19]. In the following, we show that nld-nogoods can be reduced in size by considering positive decisions only. Hence, we benefit from both an improvement in space complexity and a more powerful pruning capability.

By construction, CSP nogoods do not contain two opposite decisions i.e. both $x = v$ and $x \neq v$. Propositional resolution allows to deduce the clause $r = (\alpha \vee \beta)$ from the clauses $x \vee \alpha$ and $\neg x \vee \beta$. Nogoods can be represented as propositional clauses (disjunction of literals), where literals correspond to positive and negative decisions. For example, a nogood $\Delta = \{X_1 = v_1, X_2 \neq, v_2, X_3 = v_3, X_4 \neq v_4\}$ can be represented by the clause $c = (X_1 \neq v_1 \vee X_2 = v_2 \vee X_3 \neq v_3 \vee X_4 = v_4)$. Consequently, we can extend resolution to deal directly with CSP nogoods (e.g. [23]), called Constraint Resolution (C-Res for short). It can be defined as follows:

Definition 5. *Let P be a CN, and $\Delta_1 = \Gamma \cup \{x_i = v_i\}$ and $\Delta_2 = \Lambda \cup \{x_i \neq v_i\}$ be two nogoods of P. We define Constraint Resolution as C-Res$(\Delta_1, \Delta_2) = \Gamma \cup \Lambda$.*

It is immediate that C-Res(Δ_1, Δ_2) is a nogood of P.

Proposition 2. *Let P be a CN and Σ be the sequence of decisions taken along a branch of the search tree. For any nld-subsequence $\Sigma' = \langle \delta_1, \ldots, \delta_i \rangle$ of Σ, the set $\Delta = pos(\Sigma') \cup \{\neg\delta_i\}$ is a nogood of P (called reduced nld-nogood).*

Proof. We suppose that Σ contains $k \geq 1$ negative decisions, denoted by $\delta_{g_1}, \ldots, \delta_{g_k}$, in the order that they appear in Σ. The nld-subsequence of Σ with k negative decisions is $\Sigma'_1 = \langle \delta_1, \ldots, \delta_{g_1}, \ldots, \delta_{g_k} \rangle$. Its corresponding nld-nogood is $\Delta_1 = \{\delta_1, \ldots, \delta_{g_1}, \ldots, \delta_{g_{k-1}}, \ldots, \neg\delta_{g_k}\}$, $\delta_{g_{k-1}}$ being now the last negative decision. The nld-subsequence of Σ with $k-1$ negative decisions is $\Sigma'_2 = \langle \delta_1, \ldots, \delta_{g_1}, \ldots, \delta_{g_{k-1}} \rangle$. Its corresponding nld-nogood is $\Delta_2 = \{\delta_1, \ldots, \delta_{g_1}, \ldots, \neg\delta_{g_{k-1}}\}$. We now apply C-Res between Δ_1 and Δ_2 and we obtain $\Delta'_1 = $ C-Res$(\Delta_1, \Delta_2) = \{\delta_1, \ldots, \delta_{g_1}, \ldots, \delta_{g_{k-2}}, \ldots, \delta_{g_{k-1}-1}, \delta_{g_{k-1}+1}, \ldots, \neg\delta_{g_k}\}$. The last negative decision is now $\delta_{g_{k-2}}$, which will be eliminated with the nld-nogood containing $k-2$ negative decisions. All the remaining negative decisions are then eliminated by applying the same process. □

One interesting aspect is that the space required to store all nogoods corresponding to any branch of the search tree is polynomial with respect to the number of variables and the greatest domain size.

Proposition 3. *Let P be a CN and Σ be the sequence of decisions taken along a branch of the search tree. The space complexity to record all nld-nogoods of Σ is O(n^2d^2) while the space complexity to record all reduced nld-nogoods of Σ is O(n^2d).*

Proof. First, the number of negative decisions in any branch is O(nd). For each negative decision, we can extract a (reduced) nld-nogood. As the size of any (resp. reduced) nld-nogood is O(nd) (resp. O(n) since it only contains positive decisions), we obtain an overall space complexity of O(n^2d^2) (resp. O(n^2d)). □

4 Nogood Recording from Restarts

In [15], it has been shown that the runtime distribution produced by a randomized search algorithm is sometimes characterized by an extremely long tail with some infinite moment. For some instances, this heavy-tailed phenomenon can be avoided by using random restarts, i.e. by restarting search several times while randomizing the employed search heuristic. For constraint satisfaction, restarts have been shown productive. However, when learning is not exploited (as it is currently the case for most of the academic and commercial solvers), the average performance of the solver is damaged (cf. Section 6).

Nogood recording has not yet been shown to be quite convincing for CSP (one noticeable exception is [19]) and, further, it is a technique that leads, when uncontrolled, to an exponential space complexity. We propose to address this issue by combining nogood recording and restarts in the following way: reduced nld-nogoods are recorded from the last (and current) branch of the search tree between each run. Our aim is to benefit from both restarts and learning capabilities without sacrificing solver performance and space complexity.

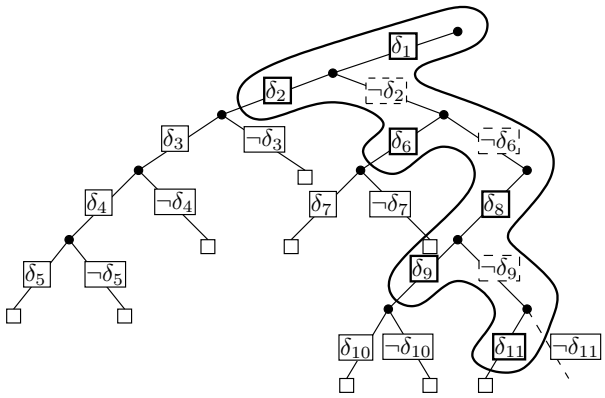

Fig. 1. Area of nld-nogoods in a partial search tree

Figure 1 depicts the partial search tree explored when the solver is about to restart. Positive decisions being taken first, a δ_i (resp. $\neg\delta_i$) corresponds to a positive (resp. negative) decision. Search has been stopped after refuting δ_{11} and taking the decision $\neg\delta_{11}$. The nld-nogoods of P are the following: $\Delta_1 = \{\delta_1, \neg\delta_2, \neg\delta_6, \delta_8, \neg\delta_9, \delta_{11}\}$, $\Delta_2 = \{\delta_1, \neg\delta_2, \neg\delta_6, \delta_8, \delta_9\}$, $\Delta_3 = \{\delta_1, \neg\delta_2, \delta_6\}$, $\Delta_4 = \{\delta_1, \delta_2\}$. The first reduced nld-nogood is obtained as follows:

$$\begin{aligned}\Delta'_1 &= \text{C-Res}(\text{C-Res}(\text{C-Res}(\Delta_1, \Delta_2), \Delta_3), \Delta_4)\\ &= \text{C-Res}(\text{C-Res}(\{\delta_1, \neg\delta_2, \neg\delta_6, \delta_8, \delta_{11}\}, \Delta_3), \Delta_4)\\ &= \text{C-Res}(\{\delta_1, \neg\delta_2, \delta_8, \delta_{11}\}, \Delta_4)\\ &= \{\delta_1, \delta_8, \delta_{11}\}\end{aligned}$$

Applying the same process to the other nld-nogoods, we obtain:
$\Delta'_2 =$ C-Res(C-Res$(\Delta_2, \Delta_3), \Delta_4) = \{\delta_1, \delta_8, \delta_9\}$.
$\Delta'_3 =$ C-Res$(\Delta_3, \Delta_4) = \{\delta_1, \delta_6\}$.
$\Delta'_4 = \Delta_4 = \{\delta_1, \delta_2\}$.

In order to avoid exploring the same parts of the search space during subsequent runs, recorded nogoods can be exploited. Indeed, it suffices to control that the decisions of the current branch do not contain all decisions of one nogood. Moreover, the negation of the last unperformed decision of any nogood can be inferred as described in the next section. For example, whenever the decision δ_1 is taken, we can infer $\neg\delta_2$ from nogood Δ'_4 and $\neg\delta_6$ from nogood Δ'_3.

Finally, we want to emphasize that *reduced nld-nogoods extracted from the last branch subsume all reduced nld-nogoods that could be extracted from any branch previously explored*.

5 Managing Nogoods

In this section, we show how to efficiently exploit reduced nld-nogoods by using the SAT technique of watched literals [24, 30, 11]. We present an efficient propagation algorithm enforcing GAC on all learned reduced nld-nogoods that can be collectively considered as a global constraint. It is important to note that, reduced nld-nogoods will be stored under the form of propositional clauses only involving negative literals.

5.1 Data structures

First, we introduce three basic types that will be useful for defining our data structures. The first one, denoted $Literal$, identifies any positive or negative decision (i.e. any variable assignment or value refutation). It then corresponds to a structure including three fields as follows:

- $variable$ is a reference to a variable
- $value$ is an integer that belongs to the initial domain of the variable
- $positive$ is a Boolean that indicates if the decision is positive ($true$) or not ($false$)

The second one, denoted $Element$, associates a nogood with two watched literals. It then corresponds to a structure including three fields as follows:

- $nogood$ is an array of $Literal$ references (whose size is at least 2)
- $watch1$ is an integer which gives the position of a first watched literal in $nogood$
- $watch2$ is an integer which gives the position of a second watched literal in $nogood$

The third one, denoted $Link$, allows to build linked lists of $Element$ references. These links are used to access nogoods recorded in the base. It corresponds to a structure including two fields as follows:

- $element$ is an $Element$ reference
- $next$ is a $Link$ reference (whose value is nil if it is not followed by another link)

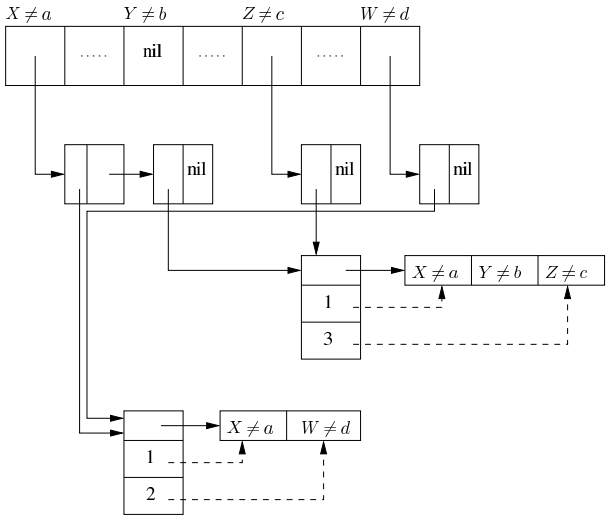

Fig. 2. Partial view of the nogood base

We can now introduce the following global structure:

– $watches$ is an array of $Link$ references, which gives, for each literal δ, the head of a list containing all nogoods Δ such that δ is watched in Δ.

We will consider that the indexing of any array t of size s ranges from 1 to s and that $size(t)$ denotes s. Also, remark that references must be considered as pointers (following the Java model), and that nil is used for empty references. Initially, $watches$ is an array such that, for each literal δ, we have $watches[\delta]$ initialized to nil. Here, to simplify the presentation and without any loss of generality, we consider $watches$ as a kind of associative array (map) which gives for each literal δ, the reference to the first nogood (via an $Element$ reference) currently involving δ as watched literal. In practice, to guarantee a constant time access to this first element, we can either use a three-dimensional array or some specific encoding of literals.

Figure 2 illustrates the data structures that we have introduced. In a partial view, one can observe two recorded nogoods. The first one contains the two watched literals $X \neq a$ and $Z \neq c$ whereas the second one contains $X \neq a$ and $W \neq d$.

5.2 Recording Nogoods

Nogoods derived from the current branch of the search tree (i.e. reduced nld-nogoods) when the current run is stopped can be recorded by calling the $storeNogoods$ function (see Algorithm 1). The parameter of this function is the sequence of literals labelling the current branch. As observed in Section 3, a reduced nld-nogood can be recorded from each negative decision occurring in this sequence. From the root to the leaf of the

Algorithm 1 storeNogoods(branch : array of Literal)

```
 1: positiveLiterals : array of size(branch) Literal
 2: nbPositiveLiterals ← 0
 3: for i ranging from 1 to size(branch) do
 4:    if branch[i].positive then
 5:       nbPositiveLiterals ← nbPositiveLiterals + 1
 6:       positiveLiterals[nbPositiveLiterals] ← branch[i]
 7:    else
 8:       if nbPositiveLiterals = 0 then
 9:          remove branch[i].value from branch[i].variable for all subsequent runs
10:       else
11:          nogood : array of nbPositiveLiterals + 1 Literal
12:          for j ranging from 1 to nbPositiveLiterals do
13:             nogood[j] ← positiveLiterals[j]
14:             nogood[j].positive ← false
15:          end for
16:          nogood[nbPositiveLiterals+1] ← branch[i]
17:          addNogood(nogood)
18:       end if
19:    end if
20: end for
```

Algorithm 2 addNogood(nogood : array of Literal)

```
1: element : Element
2: element.nogood ← nogood
3: element.watch1 ← 1
4: insertWatch(nogood[1],element)
5: element.watch2 ← size(nogood)
6: insertWatch(nogood[size(nogood)], element])
```

current branch, when a positive literal is encountered, it is recorded in an array (lines 5 and 6), and when a negative literal is encountered, we build a nogood from this literal and all recorded positive ones (lines 11 to 16). It is important to remark that, here, the nogood is considered as a clause (disjunction of literals), this is the reason why we modify the phase of the literals (see line 14). If the nogood is of size 1, it can be directly exploited by reducing the domain of the involved variable (line 9). Otherwise, it is recorded, by calling the $addNogood$ function, into the base (line 17).

To record a new nogood, the $addNogood$ function (see Algorithm 2) is called. We select as watched literals the first and last literal of the nogood. To do this, we have to call the function $insertWatch$ (see Algorithm 3). A new link is used to become the first link of the list of nogoods (via elements) involving the given literal as watched literal.

We can show that the worst-case time complexity of $storeNogoods$ is O($\lambda_p \lambda_n$) where λ_p and λ_n are the number of positive and negatives decisions on the current branch, respectively.

Algorithm 3 insertWatch(literal : Literal, element : Element)

```
1: link : Link
2: link.element ← element
3: link.next ← watches[literal]
4: watches[literal] ← link
```

5.3 Exploiting Nogoods

Inferences can be performed using reduced nld-nogoods while establishing (maintaining) Generalized Arc Consistency. We show it with a coarse-grained GAC algorithm based on a variable-oriented propagation scheme [22, 8, 6]. The Algorithm 4 can be applied to any CN (involving constraints of any arity) in order to establish GAC. At preprocessing, $propagate$ must be called with the set S of variables of the network whereas during search, S only contains the variable involved in the last positive or negative decision. At any time, the principle is to have in Q all variables whose domains have been reduced by propagation.

Algorithm 4 propagate(S : Set of variables) : Boolean

```
1: Q ← S
2: while Q ≠ ∅ do
3:    pick and delete X from Q
4:    if | dom(X) | = 1 then
5:       let a be the unique value in dom(X)
6:       if checkWatches(X ≠ a) = false then return false
7:    end if
8:    for each C | X ∈ vars(C) do
9:       for each Y ∈ Vars(C) | X ≠ Y do
10:         if revise(C,Y) then
11:            if dom(Y) = ∅ then return false
12:            else Q ← Q ∪ {Y}
13: end while
14: return true
```

Initially, Q contains all variables of the given set S (line 1). Then, iteratively, each variable X of Q is selected (line 3). If dom(X) corresponds to a singleton $\{v\}$ (lines 4 to 7), we can exploit recorded nogoods by checking the consistency of the nogood base. This is performed by the function $checkWatches$ (described below) by iterating all nogoods involving $X \neq v$ as watched literal. For each such nogood, either another literal not yet watched can be found, or an inference is performed (and the set Q is updated).

The rest of the algorithm (lines 8 to 12) corresponds to the body of a classical generic coarse-grained GAC algorithm. For each constraint C binding X, we perform the revision of all arcs (C, Y) with $Y \neq X$. A revision is performed by a call to the function $revise$, specific to the chosen coarse-grained arc consistency algorithm, and

Algorithm 5 checkWatches(literal : Literal) : Boolean

```
 1: previous ← nil
 2: current ← watches[literal]
 3: while current ≠ nil do
 4:    position ← canFindAnotherWatch(current.access)
 5:    if position ≠ -1 then
 6:       if previous = nil then watches[literal] ← watches[literal].next
 7:       else previous.next ← current.next
 8:       if literal = current.element.nogood[current.element.watch1] then
 9:         current.element.watch1 ← position
10:       else
11:         current.element.watch2 ← position
12:       let newWatchedLiteral be current.element.nogood[i]
13:       tmp ← current.next
14:       current.next ← watches[newWatchedLiteral]
15:       watches[newWatchedLiteral] ← current
16:       current ← tmp
17:    else
18:       if literal = current.element.nogood[current.element.watch1] then
19:         inferredLiteral ← current.element.nogood[current.element.watch2]
20:       else
21:         inferredLiteral ← current.element.nogood[current.element.watch1]
22:       let X be inferredLiteral.variable and v be inferredLiteral.value
23:       if v ∈ dom(X) then
24:          remove v from dom(X)
25:          if dom(X) = ∅ then return false
26:          else Q ← Q ∪ {X}
27:       end if
28:       previous ← current
29:       current ← current.next
30:    end if
31: end while
32: return true
```

entails removing values that became inconsistent with respect to C. When the revision of an arc (C, Y) involves the removal of some values in $dom(Y)$, $revise$ returns $true$ and the variable Y is added to Q. For more information about this algorithm and some of these optimizations, see [6] The algorithm loops until a fix-point is reached.

The principle of Algorithm 5 is to iterate the list of elements (nogoods) involving as watched literal the literal given in parameter. For each such element, denoted by $current$ at each turn of the main loop, we have to look for another watched literal. This is done by calling the function $canFindAnotherWatch$ (see Algorithm 6). If we can find a literal which is not currently watched (see line 2) and which can be watched then its position is returned. Otherwise, -1 is returned. When a new watched literal has been found, we have to update (i.e. remove an element) the list $watches[literal]$ (lines 6 and 7), update a watched literal position (lines 8 to 11) and update (i.e. add an element) the list $watches[newWatchedLiteral]$ (lines 13 to 15). When no other literal can be

Algorithm 6 canFindAnotherWatch(element : Element) : integer

```
1: for i ranging from 1 to size(element.nogood) do
2:    if element.watch1 = i or element.watch2 = i then continue
3:    let X be element.nogood[i].variable and v be element.nogood[i].value
4:    if v ∉ dom(X) or | dom(X) |> 1 then return i
5: end for
6: return −1
```

watched, we can then infer that the second watched literal must be verified. Remember that we only record reduced nld-nogoods. Hence, the inference is necessarily of the form $X \neq a$. Taking into account this inference when a still belongs to dom(X), we can remove a from dom(X), which can yield an inconsistency or an update of the set Q.

The worst-case time complexity of $checkWatches$ is O($n\gamma$) where γ is the number of reduced nld-nogoods stored in the base and n is the number of variables[2]. Indeed, in the worst case, each nogood is watched by the literal given in parameter, and the time complexity of dealing with a reduced nld-nogood in order to find another watched literal or make an inference is O(n). Then, the worst-case time complexity of $propagate$ is O($er^2d^r + n^2\gamma$) where r is the greatest constraint arity. More precisely, the cost of establishing GAC (using a generic approach) is O(er^2d^r) when an algorithm such as GAC2001 [4] is used and the cost of exploiting nogoods to enforce GAC is O($n^2\gamma$). Indeed, $checkWatches$ is O($n\gamma$) and it can be called only once per variable.

The space complexity of the structures introduced to manage reduced nld-nogoods in a backtracking search algorithm is O($n(d + \gamma)$). Indeed, we need to store γ nogoods of size at most n and we need to store watched literals which is O(nd).

6 Experiments

In order to show the practical interest of the approach described in this paper, we have conducted an extensive experimentation (on a PC Pentium IV 2.4GHz 512Mo under Linux). We have used the state-of-the-art algorithm MAC [26] and studied the impact of exploiting restarts (denoted by MAC+RST) and nogood recording from restarts (denoted by MAC+RST+NG). Concerning the restart policy, the initial number of allowed backtracks for the first run has been set to 10 and the increasing factor to 1.5 (i.e., at each new run, the number of allowed backtracks increases by a 1.5 factor). We used three different variable ordering heuristics: the classical $brelaz$ [7] and $dom/ddeg$ [3] as well as the adaptive $dom/wdeg$ that has been recently shown to be the most efficient generic heuristic [5, 20, 16, 28]. Importantly, when restarts are performed, randomization is introduced in $brelaz$ and $dom/ddeg$ to break ties. For $dom/wdeg$, the weight of constraints are preserved from each run to the next one, which makes randomization useless (weights are sufficiently discriminant).

In our first experimentation, we have tested the three algorithms on the full set of 1064 instances used as benchmarks for the first competition of CSP solvers [28]. The

[2] In practice, the size of reduced nld-nogoods can be far smaller than n (cf. Section 6).

time limit to solve an instance was fixed to 30 minutes. Table 1 provides an overview of the results in terms of the number of instances unsolved within the time limit ($\#timeouts$) and the average cpu time in seconds ($avg\ time$) computed from instances solved by all three methods. Figures 3 and 4 represent scatter plots displaying pairwise comparisons for $dom/ddeg$ and $dom/wdeg$. Finally, Table 2 focuses on the most difficult real-world instances of the Radio Link Frequency Assignment Problem (RLFAP). Performance is measured in terms of the cpu time in seconds (no timeout) and the number of visited nodes. An analysis of all these results reveals three main points.

Restarts (without learning) yields mitigated results. First, we observe an increased average cpu time for all heuristics and fewer solved instances for classical ones. However, a close look at the different series reveals that MAC+RST combined with $brelaz$ (resp. $dom/ddeg$) solved 27 (resp. 32) less instances than MAC on the series ehi. These instances correspond to random instances embedding a small unsatisfiable kernel. As classical heuristics do not guide search towards this kernel, restarting search tends to be nothing but an expense. Without these series, MAC+RST would have solved more instances than MAC (but, still, with worse performance). Also, remark that $dom/wdeg$ renders MAC+RST more robust than MAC (even on the ehi series).

Nogood recording from restarts improves MAC performance. Indeed, both the number of unsolved instances and the average cpu time are reduced. This is due to the fact that the solver never explores several times the same portion of the search space while benefiting from restarts.

Nogood recording from restarts applied to real-world instances pays off. When focusing to the hardest instances [28] built from the real-world RLFAP instance $scen$-11, we can observe in Table 2 that using a restart policy allows to be more efficient by almost one order of magnitude. When we further exploit nogood recording, the gain is about 10%.

		MAC		
			$+ RST$	$+ RST + NG$
$dom/ddeg$	$\#timeouts$	365	378	337
	$avg\ time$	125.0	159.0	109.1
$brelaz$	$\#timeouts$	277	298	261
	$avg\ time$	85.1	121.7	78.2
$dom/wdeg$	$\#timeouts$	140	123	121
	$avg\ time$	47.8	56.0	43.6

Table 1. Number of unsolved instances and average cpu time on the 2005 CSP competition benchmarks, given 30 minutes CPU.

Finally, we noticed that the number and the size of the reduced nld-nogoods recorded during search were always very limited. As an illustration, let us consider the hardest RLFAP instance $scen11 - f1$ which involves 680 variables and a greatest domain size of 43 values. MAC+RST+NG solved this instance in 36 runs while only 712 nogoods of average size 8.5 and maximum size 33 were recorded.

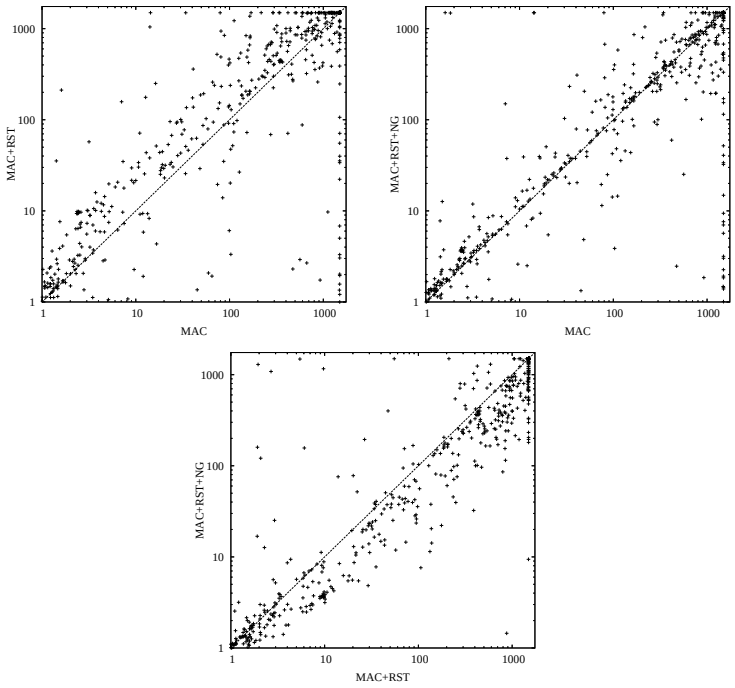

Fig. 3. Pairwise comparison (cpu time) on the 2005 CSP competition benchmarks using the dom/ddeg heuristic

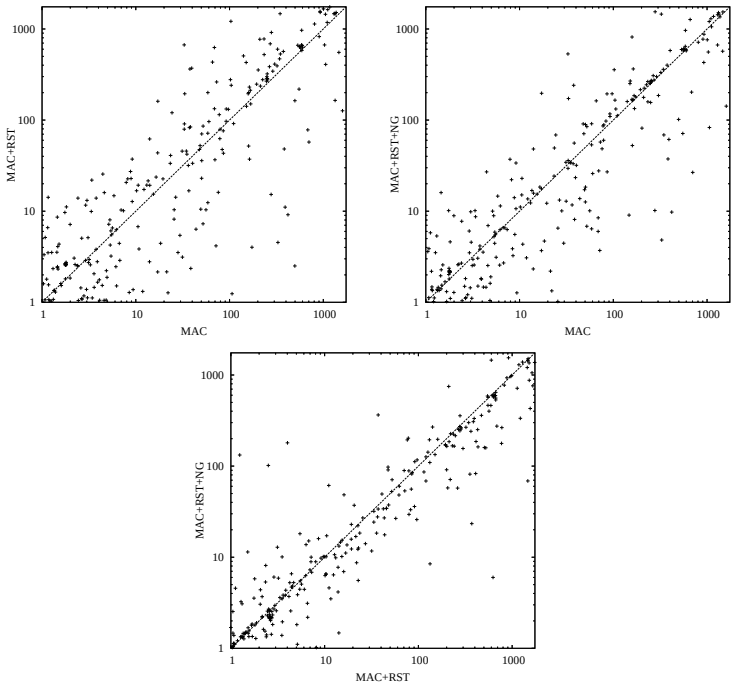

Fig. 4. Pairwise comparison (cpu time) on the 2005 CSP competition benchmarks using the dom/wdeg heuristic

		MAC		
			$+RST$	$+RST+NG$
scen11-f12	cpu	0.85	0.84	0.84
	$nodes$	695	477	445
scen11-f10	cpu	0.95	0.82	1.03
	$nodes$	862	452	636
scen11-f8	cpu	14.6	1.8	1.9
	$nodes$	14068	1359	1401
scen11-f7	cpu	185	9.4	8.4
	$nodes$	$207K$	9530	8096
scen11-f6	cpu	260	21.8	16.9
	$nodes$	$302K$	22002	16423
scen11-f5	cpu	1067	105	82.3
	$nodes$	$1327K$	$117K$	90491
scen11-f4	cpu	2494	367	339
	$nodes$	$2826K$	$419K$	$415K$
scen11-f3	cpu	9498	1207	1035
	$nodes$	$12M$	$1517K$	$1286K$
scen11-f2	cpu	$29K$	3964	3378
	$nodes$	$37M$	$5011K$	$4087K$
scen11-f1	cpu	$69K$	9212	8475
	$nodes$	$93M$	$12M$	$10M$

Table 2. Performance on hard RLFAP Instances using the $dom/wdeg$ heuristic (no timeout)

7 Conclusion

In this paper, we have studied the interest of recording nogoods in conjunction with a restart strategy. The benefit of restarting search is that the heavy-tailed phenomenon observed on some instances can be avoided. The drawback is that we can explore several times the same parts of the search tree. We have shown that it is quite easy to eliminate this drawback by recording a set of nogoods at the end of each run (similarly to the *search signature* technique proposed [1] for SAT). For efficiency reasons, nogoods are recorded in a base (and so do not correspond to new constraints) and propagation is performed using the 2-literal watching technique introduced for SAT. One can consider the base of nogoods as a unique global constraint with an efficient associated propagation algorithm.

Our experimental results show the effectiveness of our approach since the state-of-the-art generic algorithm MAC-dom/wdeg is improved. Our approach not only allows to solve more instances than the classical approach within a given timeout, but also is, on the average, faster on instances solved by both approaches.

References

1. L. Baptista, I. Lynce, and J. Marques-Silva. Complete search restart strategies for satisfi ability. In *Proceedings of SSA'01 workshop held with IJCAI'01*, 2001.
2. R.J. Bayardo and R.C. Shrag. Using CSP look-back techniques to solve real-world SAT instances. In *Proceedings of AAAI'97*, pages 203–208, 1997.
3. C. Bessiere and J. Régin. MAC and combined heuristics: two reasons to forsake FC (and CBJ?) on hard problems. In *Proceedings of CP'96*, pages 61–75, 1996.
4. C. Bessiere, J.C. Régin, R.H.C. Yap, and Y. Zhang. An optimal coarse-grained arc consistency algorithm. *Artifi cial Intelligence*, 165(2):165–185, 2005.

5. F. Boussemart, F. Hemery, C. Lecoutre, and L. Sais. Boosting systematic search by weighting constraints. In *Proceedings of ECAI'04*, pages 146–150, 2004.
6. F. Boussemart, F. Hemery, and C. Lecoutre. Revision ordering heuristics for the Constraint Satisfaction Problem. In *Proceedings of CPAI'04 workshop held with CP'04*, pages 29–43, 2004.
7. D. Brelaz. New methods to color the vertices of a graph. *Communications of the ACM*, 22:251–256, 1979.
8. A. Chmeiss and P. Jégou. Efficient path-consistency propagation. *International Journal on Artificial Intelligence Tools*, 7(2):121–142, 1998.
9. R. Dechter. Enhancement schemes for constraint processing: backjumping, learning and cutset decomposition. *Artificial Intelligence*, 41:273–312, 1990.
10. R. Dechter. *Constraint processing*. Morgan Kaufmann, 2003.
11. N. Eén and N. Sorensson. An extensible sat-solver. In *Proceedings of SAT'03*, 2003.
12. F. Focacci and M. Milano. Global cut framework for removing symmetries. In *Proceedings of CP'01*, pages 77–92, 2001.
13. D. Frost and R. Dechter. Dead-end driven learning. In *Proceedings of AAAI'94*, pages 294–300, 1994.
14. M. Ginsberg. Dynamic backtracking. *Artificial Intelligence*, 1:25–46, 1993.
15. C.P. Gomes, B. Selman, N. Crato, and H. Kautz. Heavy-tailed phenomena in satisfiability and constraint satisfaction problems. *Journal of Automated Reasoning*, 24:67–100, 2000.
16. T. Hulubei and B. O'Sullivan. Search heuristics and heavy-tailed behaviour. In *Proceedings of CP'05*, pages 328–342, 2005.
17. J. Hwang and D.G. Mitchell. 2-way vs d-way branching for CSP. In *Proceedings of CP'05*, pages 343–357, 2005.
18. G. Katsirelos and F. Bacchus. Unrestricted nogood recording in CSP search. In *Proceedings of CP'03*, pages 873–877, 2003.
19. G. Katsirelos and F. Bacchus. Generalized nogoods in CSPs. In *Proceedings of AAAI'05*, pages 390–396, 2005.
20. C. Lecoutre, F. Boussemart, and F. Hemery. Backjump-based techniques versus conflict-directed heuristics. In *Proceedings of ICTAI'04*, pages 549–557, 2004.
21. J.P. Marques-Silva and K.A. Sakallah. Conflict analysis in search algorithms for propositional satisfiability. Technical Report RT/4/96, INESC, Lisboa, Portugal, 1996.
22. J.J. McGregor. Relational consistency algorithms and their application in finding subgraph and graph isomorphisms. *Information Sciences*, 19:229–250, 1979.
23. D.G. Mitchell. Resolution and constraint satisfaction. In *Proceedings of CP'03*, pages 555–569, 2003.
24. M. W. Moskewicz, C. F. Madigan, Y. Zhao, L. Zhang, and S. Malik. Chaff: Engineering an Efficient SAT Solver. In *Proceedings of DAC'01*, pages 530–535, 2001.
25. P. Prosser. Hybrid algorithms for the constraint satisfaction problems. *Computational Intelligence*, 9(3):268–299, 1993.
26. D. Sabin and E. Freuder. Contradicting conventional wisdom in constraint satisfaction. In *Proceedings of CP'94*, pages 10–20, 1994.
27. T. Schiex and G. Verfaillie. Nogood recording for static and dynamic constraint satisfaction problems. *International Journal of Artificial Intelligence Tools*, 3(2):187–207, 1994.
28. M.R.C. van Dongen, editor. *Proceedings of CPAI'05 workshop held with CP'05*, volume II, 2005.
29. H. Zhang. A random jump strategy for combinatorial search. In *Proceedings of AI&M'02*, 2002.
30. L. Zhang and S. Malik. The quest for efficient boolean satisfiability solvers. In *Proceedings of CADE'02*, pages 295–313, 2002.