Journal on Satisfiability, Boolean Modeling and Computation 1 (2007) 147–167

Recording and Minimizing Nogoods from Restarts

Christophe Lecoutre lecoutre@cril.univ-artois.fr
Lakhdar Saïs sais@cril.univ-artois.fr
Sébastien Tabary tabary@cril.univ-artois.fr
Vincent Vidal vidal@cril.univ-artois.fr
CRIL (Centre de Recherche en Informatique de Lens),
CNRS FRE 2499,
rue de l'université, SP 16
62307 Lens cedex, France

Abstract

In this paper[1], 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 ones. 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 exploited using the structure of watched literals originally proposed for SAT. We prove that the worst-case time complexity of extracting such nogoods at the end of each run is only $O(n^2d)$ where n is the number of variables of the constraint network and d the size of the greatest domain, whereas for any node of the search tree, the worst-case time complexity of exploiting these nogoods to enforce Generalized Arc Consistency (GAC) is $O(n|\mathscr{B}|)$ where $|\mathscr{B}|$ denotes the number of recorded nogoods. As the number of nogoods recorded before each new run is bounded by the length of the last branch, the total number of recorded nogoods is polynomial in the number of restarts. Interestingly, we show that when the minimization of the nogoods is envisioned with respect to an inference operator ϕ, it is possible to directly identify some nogoods that cannot be minimized. For $\phi = AC$ (i.e. for MAC), the worst-case time complexity of extracting minimal nogoods is slightly increased to $O(en^2d^3)$ where e is the number of constraints of the network. Experimentation over a wide range of CSP instances using a generic state-of-the-art CSP solver demonstrates the effectiveness of this approach. Recording nogoods (and in particular, minimal nogoods) from restarts significantly improves the robustness of the solver.

Keywords: *learning, watched literals, restarts, CSP*

Submitted November 2006; revised April 2007; published May 2007

1. Introduction

Nogood recording (or learning) has been suggested as a technique to enhance CSP (Constraint Satisfaction Problem) solving in [10]. The principle is to record a nogood whenever a conflict occurs during a backtracking search. Such nogoods can then be exploited later

1. This paper extends [27] by exploiting a nogood minimization technique and providing further experiments.

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 [10, 14, 37].

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 [32]. 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 [10] and conflict directed backjumping [34] have been introduced for CSP and later imported into SAT solvers [2, 28].

Recently, a generalization of nogoods, as well as an elegant learning method, have been proposed in [23, 24] 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 cannot lead to a complete elimination of redundancy in search trees. An original alternative [41] 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 for CSP within the randomization and restart framework. The principle of our approach is to learn nogoods from the last branch of the search tree before each restart, discarding already explored parts of the search tree in subsequent runs. Remark that a related approach has been proposed in [1, 15] for SAT in order to obtain a complete restart strategy while reducing the number of recorded nogoods. Roughly speaking, in our approach, we manage nogoods by introducing a global constraint with a dedicated filtering algorithm which exploits watched literals [32]. Interestingly, this algorithm that allows to enforce Generalized Arc Consistency (GAC) on the base of recorded nogoods can be easily integrated to any constraint propagation engine (e.g. see [38]) but also to any generic GAC algorithm. The simplicity and the good worst-case time complexity (only $O(n|\mathscr{B}|)$ where n denotes the number of variables of the constraint network and $|\mathscr{B}|$ the number of nogoods in the base $\mathscr{B}$) render this approach attractive. As the number of nogoods recorded before each new run is bounded by the length of the last branch of the search tree, the total number of recorded nogoods is polynomial in the number of restarts. Besides, we show that it is possible to directly identify some nogoods that cannot be subject to minimization with respect to an inference operator ϕ.

The paper is organized as follows. After some technical background, we introduce so-called reduced nld-nogoods and the principle of nogood recording from restarts. Then we present a detailed description of how such nogoods are extracted and exploited in the context of a backtrack search algorithm. Next, we address the issue of minimizing reduced nld-nogoods. Finally, we give the results of a vast experimentation that we have conducted before concluding.

2. Technical Background

A Constraint Network (CN) P is a pair $(\mathscr{X},\mathscr{C})$ where $\mathscr{X}$ is a set of variables and $\mathscr{C}$ a set of 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 $vars(C)$. The number of variables of a CN will be denoted by n, the number of constraints by e, the greatest domain size by d and the greatest constraint arity by r. Also, for any given set E, $|E|$ will denote the number of elements in E.

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 if and only if 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 [11].

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 [11]. Today, MAC [36] 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 [20]. With binary branching, at each step of search, a pair (X,a) is selected where X is an unassigned variable and a a value in dom(X), and two cases are considered: the assignment $X = a$ and the refutation $X \neq a$. 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) [34] and DBT (Dynamic Backtracking) [16] exist, it has been shown [3, 5, 25] that MGAC 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 (e.g. MGAC) that is based on the 2-way branching scheme, positive decisions being performed first, and that maintains a consistency (e.g. Generalized Arc Consistency) at each node.

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,a) be a pair such that $X \in \mathscr{X}$ and $a \in dom(X)$. The assignment $X = a$ is called a positive decision whereas the refutation $X \neq a$ is called a negative decision. $\neg(X = a)$ is equivalent to $X \neq a$ and $\neg(X \neq a)$ is equivalent to $X = a$.*

Definition 2. *Let $\Sigma = \langle \delta_1, \ldots, \delta_i, \ldots, \delta_m \rangle$ be a sequence of decisions. The sequence $\langle \delta_1, \ldots, \delta_i \rangle$, where δ_i is a negative decision, 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 such that, for any positive decision $(X = a) \in \Delta$, $dom(X)$ is restricted to $\{a\}$, and, for any negative decision $(X \neq a) \in \Delta$, a 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, from the root to a leaf, a nogood can be extracted from each negative decision (also mentioned in [35]). 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)*[2.]*.*

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 [13, 24]. In the following, we will show that nld-nogoods can be reduced in size by considering positive decisions only.

Propositional resolution can be extended to directly deal with CSP nogoods, e.g. in [31] where it is called Constraint Resolution (C-Res for short). Note that we can safely use C-Res since, in the search tree, two opposite decisions, e.g. both $X = a$ and $X \neq a$, cannot occur in the same branch and consequently, cannot occur in any nogood. It can be defined as follows:

Definition 5. *Let P be a CN, and $\Delta_1 = \Gamma \cup \{X = a\}$ and $\Delta_2 = \Lambda \cup \{X \neq a\}$ 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).*

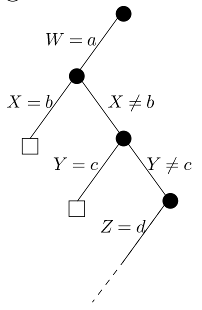
2. The notation $\{\delta_1, \ldots, \neg\delta_i\}$ corresponds to $\{\delta_j \in \Sigma \mid j < i\} \cup \{\neg\delta_i\}$ reduced to $\{\neg\delta_1\}$ when $i = 1$.

Proof. $\Sigma' = \langle \delta_1, \ldots, \delta_i \rangle$ is a nld-subsequence, so it contains $k \geq 1$ negative decisions, denoted by $\delta_{g_1}, \ldots, \delta_{g_k}$, in the order that they appear in Σ. Remark that we have $\delta_{g_k} = \delta_i$. Let us show by recurrence that $\forall j \in [1, k]$, the following hypothesis $h(j)$ is true : $\Delta'_j = pos(\Sigma') \cup \{\delta_{g_1}, \ldots, \delta_{g_{j-1}}\} \cup \{\neg\delta_i\}$ is a nogood of P.
First, we show that $h(k)$ holds. From Proposition 1, we know that $\Delta'_k = pos(\Sigma') \cup \{\delta_{g_1}, \ldots, \delta_{g_{k-1}}\} \cup \{\neg\delta_i\}$ is a nogood since Δ'_k is the nogood corresponding to the nld-subsequence Σ'.
Next, we show that if $h(j+1)$ holds, then $h(j)$ holds too. By hypothesis, Δ'_{j+1} is a nogood of P. Let Σ_j be the nld-subsequence corresponding to the prefix of Σ' such that δ_{g_j} is its last (negative) decision (we have $pos(\Sigma_j) \subseteq pos(\Sigma')$ and $neg(\Sigma_j) = \{\delta_{g_1}, \ldots, \delta_{g_j}\}$), and Δ_j be its corresponding nogood according to Proposition 1. By applying Constraint Resolution, we have:

$$\begin{array}{rcl} \Delta'_{j+1} & = & pos(\Sigma') \cup \{\delta_{g_1}, \ldots, \delta_{g_{j-1}}, \delta_{g_j}\} \cup \{\neg\delta_i\} \\ \Delta_j & = & pos(\Sigma_j) \cup \{\delta_{g_1}, \ldots, \delta_{g_{j-1}}\} \cup \{\neg\delta_{g_j}\} \\ \text{C-Res}(\Delta'_{j+1}, \Delta_j) & = & pos(\Sigma') \cup pos(\Sigma_j) \cup \{\delta_{g_1}, \ldots, \delta_{g_{j-1}}\} \cup \{\neg\delta_i\} \\ & = & pos(\Sigma') \cup \{\delta_{g_1}, \ldots, \delta_{g_{j-1}}\} \cup \{\neg\delta_i\} \end{array}$$

since $pos(\Sigma_j) \subseteq pos(\Sigma')$, $\neg\delta_{g_j} \in \Delta_j$ and $\delta_{g_j} \in \Delta'_{j+1}$. $\Delta'_j = \text{C-Res}(\Delta'_{j+1}, \Delta_j)$ is then proved to be a nogood of P. As a consequence, we have just proved that $h(j)$ holds.
For $j = 1$, we obtain that $\Delta = pos(\Sigma') \cup \{\neg\delta_i\}$ is a nogood of P. □

As an illustration, let $\Sigma = \langle W = a, X \neq b, Y \neq c, Z = d \rangle$ be a sequence of decisions taken along a branch of the search tree. We have then:



- $\Sigma' = \langle W = a, X \neq b, Y \neq c \rangle$ is a nld-subsequence
- $\Delta_1 = \langle W = a, X \neq b, Y = c \rangle$ is a nld-nogood
- $\Delta_2 = \langle W = a, Y = c \rangle$ is a reduced nld-nogood

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 worst-case space complexity to record all nld-nogoods of Σ is $O(n^2d^2)$ while the worst-case 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. $O(n^2d)$). □

It is important to note that reduced nld-nogoods extracted from a branch admit a better pruning capability than nld-nogoods extracted from the same branch since for each nld-subsequence, the corresponding nld-nogood is subsumed by the reduced nld-nogood.

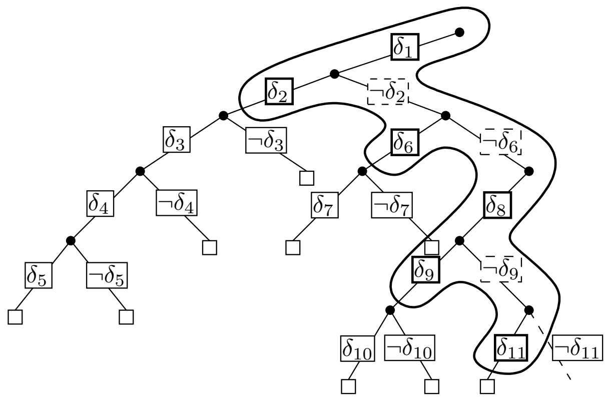


Figure 1. Partial search tree and nld-nogoods

4. Nogood Recording from Restarts

In [17], 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 randomization and restarts, i.e. by restarting search several times while randomizing the employed search heuristic. For constraint satisfaction, restarts have been shown productive on some problems. 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 7).

Nogood recording has not yet been shown to be quite convincing for CSP 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 extracted (to be recorded in a nogood base) from the current branch of the search tree at the end of each run. Our aim is to benefit from both restarts and learning capabilities without sacrificing solver performance and space complexity.

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. Here, search has been stopped after refuting δ_{11} and taking the decision $\neg\delta_{11}$. The nld-nogoods of P are the following: $\Delta_1 = \{\delta_1, \neg\delta_2, \neg\delta_6, \delta_8, \neg\delta_9, \delta_{11}\}$, $\Delta_2 = \{\delta_1, \neg\delta_2, \neg\delta_6, \delta_8, \delta_9\}$, $\Delta_3 = \{\delta_1, \neg\delta_2, \delta_6\}$ and $\Delta_4 = \{\delta_1, \delta_2\}$.

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 principle to the other nld-nogoods, we obtain:

$$\begin{aligned}
\Delta'_2 &= \text{C-Res}(\text{C-Res}(\Delta_2, \Delta_3), \Delta_4) = \{\delta_1, \delta_8, \delta_9\} \\
\Delta'_3 &= \text{C-Res}(\Delta_3, \Delta_4) = \{\delta_1, \delta_6\} \\
\Delta'_4 &= \Delta_4 = \{\delta_1, \delta_2\}
\end{aligned}$$

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 set of decisions of the current branch does not contain all the decisions of any recorded nogood. Moreover, the negation of the last undetermined decision of any nogood can be inferred as described in the next section. For example, whenever the decision δ_1 becomes true, 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.* This follows from the fact that each subtree completely explored (and, thus, all nld-nogoods that could be built from all branches of this subtree) is prefixed by at least one nld-nogood of the last branch.

5. Managing Nogoods

In this section, we show how to efficiently exploit reduced nld-nogoods by using the SAT technique of watched literals [32, 42, 12]. Reduced nld-nogoods, which correspond to sets (conjunctions) of positive decisions, will be recorded as disjunctions of negative decisions which can be seen as new constraints to be satisfied. We then present an efficient propagation algorithm enforcing GAC on all learned reduced nld-nogoods that can be collectively considered as a global constraint.

5.1 Nogood Base and Watched Literals

Reduced nld-nogoods that are extracted from the last branch only contain positive decisions and can be recorded in a base of nogoods $\mathscr{B}$. To exploit them, it suffices, each time a decision is taken during search, to check if the set of current decisions is compatible with all nogoods of $\mathscr{B}$.

In order to provide an efficient access to these nogoods we use the lazy data structure of watched literals [32, 42, 12] depicted by Figure 2. The principle is to select two decisions per nogood in order to ensure that the nogood is not violated (one watched decision would be sufficient) and no inference can be performed (using the second watched decision). These decisions are called watched literals (referenced by w_1 and w_2 in Figure 2). As long as both watched literals are not falsified, inference is not possible. Note that, in our case, watched literals correspond to negative decisions since reduced nld-nogoods only contain positive decisions and we represent each nogood as a disjunction of negative decisions. When a decision $X = a$ is performed, we have then to check for each nogood which contains $X \neq a$ as watched literal, if another valid decision can be found. If this is the case, this decision becomes the new watched one (replacing $X \neq a$), and otherwise, we have to infer the second watched decision in order to satisfy the nogood.

In practice, when a decision $X = a$ is performed, only nogoods where $X \neq a$ appears as watched are checked. We maintain for each negative decision the list of nogoods which includes this decision as watched literal. The required data structures can be defined as follows. First, we need an array of nd entries. Each entry corresponds to a negative decision

δ and represents the head of a linked list allowing the access to the nogoods of $\mathscr{B}$ which contain δ as watched literal. We can access to such a list, denoted $\mathscr{B}_\delta$, in constant time. Each nogood which is an array of (at most n) negative decisions is associated with two watched literals. In Figure 2, one can observe a nogood base with two recorded nogoods. The first is watched by $X \neq a$ and $Z \neq c$ whereas the second one is watched by $X \neq a$ and $W \neq d$.

5.2 Recording Nogoods

Reduced nld-nogoods derived from the current branch of the search tree when the current run is stopped can be recorded by calling the *storeNogoods* function (see Algorithm 1). The parameters of this function are the sequence of decisions labelling the current branch taken from the root to the leaf and the current nogood base. 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 current branch, when a positive decision is encountered, its negation is recorded, and when a negative decision is encountered, we build a nogood Δ' from this decision and all previously recorded ones in the set Δ (line 9). As a nogood is stored as a disjunction of negative decisions, we record the negation of the (positive) decision (line 4). If the nogood is of size 1, it can be directly exploited by reducing the domain of the involved variable (line 7). Otherwise, it is recorded into the nogood base $\mathscr{B}$ (line 10).

Two decisions are watched each time a nogood is recorded. Note that any decision of Δ can be watched since the search algorithm is about to restart. It means that the two

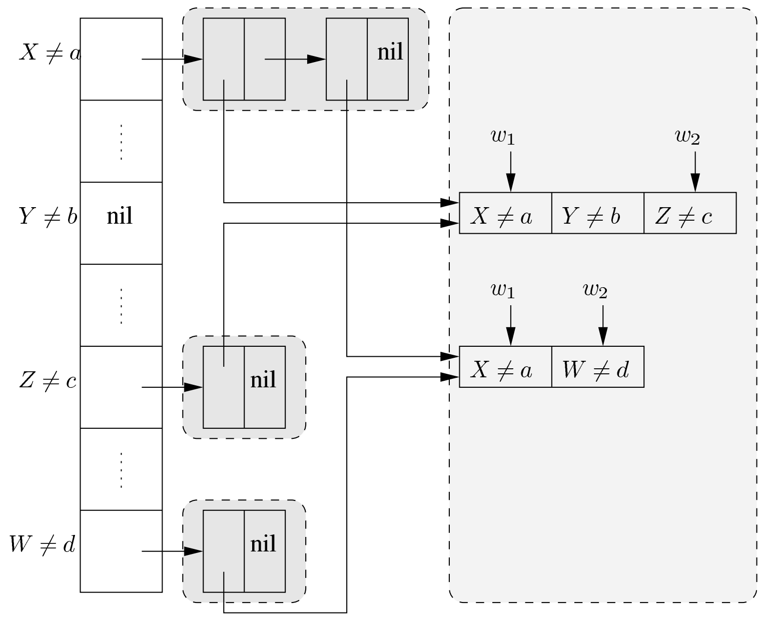


Figure 2. Partial view of a nogood base $\mathscr{B}$

Algorithm 1 storeNogoods(Σ : Sequence of Decisions, $\mathscr{B}$: Nogood base)

```
1: $\Delta \leftarrow \emptyset$
2: for each decision $\delta \in \Sigma$ ranging from the first decision of $\Sigma$ to the last do
3:    if $\delta$ is a positive decision then
4:       $\Delta \leftarrow \Delta \cup \{\neg\delta\}$
5:    else
6:       if $\Delta = \emptyset$ then
7:          remove $a$ from $dom(X)$ where $\delta = (X \neq a)$, for all subsequent runs
8:       else
9:          $\Delta' \leftarrow \Delta \cup \{\delta\}$
10:         $\mathscr{B} \leftarrow \mathscr{B} \cup \{\Delta'\}$
11:      end if
12:   end if
13: end for
```

selected decisions will be valid at the beginning of the next run. For each selected decision, a new entry (for the new nogood) is inserted in the list of nogoods watched by this decision.

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 [30, 8, 6]. Algorithm 2 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.

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 $\{a\}$ (lines 4 to 11), we can exploit recorded nogoods by checking the consistency of the nogood base. This is performed by the function $inferences$ (described below) which iterates all nogoods involving $X \neq a$ as watched literal and returns a set of inferences deduced from such nogoods. This set of inferences is then taken into account: for each identified inference $Y \neq b \in inferences(X \neq a)$, if b belongs to dom(Y), then we can remove it, which can yield an inconsistency or an update of the set Q.

The rest of the algorithm (lines 12 to 19) 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 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 optimizations, see [6]. The algorithm loops until a fixed point is reached.

The principle of Algorithm 3 is to iterate the list of nogoods involving as watched literal the decision given in parameter. For each such nogood, denoted by Δ at each turn of the

Algorithm 2 propagate(S : Set of Variables) : Boolean

```
$Q \leftarrow S$
while $Q \neq \emptyset$ do
   pick and delete $X$ from $Q$
   if $| dom(X) | = 1$ then
      Let dom($X$) = $\{a\}$
      for each $(Y \neq b) \in inferences(X \neq a)$ do
         $dom(Y) \leftarrow dom(Y) \backslash \{b\}$
         if $dom(Y) = \emptyset$ then return false
         else $Q \leftarrow Q \cup \{Y\}$
      end for
   end if
   for each $C \mid X \in vars(C)$ do
      for each $Y \in Vars(C) \mid X \neq Y$ do
         if revise($C$,$Y$) then
            if $dom(Y) = \emptyset$ then return false
            else $Q \leftarrow Q \cup \{Y\}$
         end if
      end for
   end for
end while
return true
```

Algorithm 3 inferences($X \neq a$: Decision) : Set of Decisions

```
$\Gamma \leftarrow \emptyset$
for each nogood $\Delta \in \mathscr{B}_{X \neq a}$ do
   Let $(Y \neq b)$ be the second decision watched in $\Delta$
   if b $\in dom(Y)$ then
      if $\neg$ $canFindAnotherWatch(\Delta, X \neq a)$ then
         $\Gamma \leftarrow \Gamma \cup \{Y \neq b\}$
      end if
   end if
end for
return $\Gamma$
```

Algorithm 4 canFindAnotherWatch(Δ : Nogood, $X \neq a$: Decision) : Boolean

```
for each decision $(Y \neq b) \in \Delta \mid Y \neq b$ is not watched in $\Delta$ do
   if $b \notin dom(Y)$ or $| dom(Y) | > 1$ then
      watch $Y \neq b$ instead of $X \neq a$ in $\Delta$
      return $true$
   end if
end for
return $false$
```

main loop, we have to look for another watched literal when the second decision watched in Δ is not true, i.e. b has been removed from $dom(Y)$ (line 4). This is done by calling the function *canFindAnotherWatch*. If we cannot find a new watched decision (see line 5), then the second watched decision ($Y \neq b$) is inferred.

Finally, the function *canFindAnotherWatch* (see Algorithm 4) examines all decisions (not currently watched) of the given nogood in order to find the next decision to watch (line 3). Such a decision is either already satisfied or unassigned (line 2).

Even if not described here, note that when a new watched literal has been found, we have to remove the entry (corresponding to the nogood Δ) from the list of nogoods involving $X \neq a$ as watched literal. Next we have to update (i.e. add an entry) the list of nogoods involving $Y \neq b$ as new watched literal.

5.4 Complexity Analysis

Now, we present the complexities of the different algorithms proposed to store and exploit nogoods. For what follows, remember that $\mathscr{B}$ denotes the nogood base and $|\mathscr{B}|$ the number of nogoods in $\mathscr{B}$.

Proposition 4. *The worst-case time complexity of recording reduced nld-nogoods from restarts (i.e. the worst-case time complexity of storeNogoods) is $O(n^2d)$.*

Proof. First, each nogood Δ added to $\mathscr{B}$ (line 10 of Algorithm 1) is composed of at most $|pos(\Sigma)|$ decisions, and at most $|neg(\Sigma)|$ nogoods can be extracted from Σ. Then, we can observe that the worst-case time complexity of *storeNogoods* is $O(|pos(\Sigma)|.|neg(\Sigma)|)$. As $|pos(\Sigma)|$ is $O(n)$ and $|neg(\Sigma)|$ is $O(nd)$, we obtain $O(n^2d)$. □

Proposition 5. *The worst-case time complexity of exploiting reduced nld-nogoods at each node of the search, i.e. the cumulated worst-case time complexity of $inferences$ for a single call to propagate is $O(n|\mathscr{B}|)$.*

Proof sketch. First, it is important to note that when a decision $X \neq a$ (potentially watched) is not valid anymore (i.e. a is the only value remaining in the domain of X), then it cannot be watched again (during a same call to *propagate*). Also, when looking for a decision to be watched in a reduced nld-nogood (i.e. a set of negative decisions), we can iterate its decisions in any order. To obtain the mentioned complexity, we need a refinement (not presented for the sake of simplicity) of the function *canFindAnotherWatch* described in this paper. Considering that the set of decisions of each nogood is represented using an array, we assume here that before calling the *propagate* algorithm, the two first decisions of each array are swapped with those currently watched. This operation can be performed in $O(|\mathscr{B}|)$. Then, whenever we need to find a new watched literal using the *canFindAnotherWatch* function, we just have to iterate the decisions of the nogood Δ (given in parameter) by starting from the index that follows the greatest position of both decisions currently watched up to the last decision in the array. As a consequence, for one call to the *propagate* algorithm, whatever the number of *canFindAnotherWatch* (and so $inferences$) calls is, we will check up to $|\Delta|$ decisions per nogood Δ. Consequently, the cumulated worst-case complexity of managing any nogood Δ is then $O(|\Delta|)$ which is $O(n)$. Overall, the cumulated worst-case time complexity of $inferences$ in *propagate* is then $O(n|\mathscr{B}|)$. □

Remark that the worst-case time complexity of exploiting reduced nld-nogoods for each branch of the search tree, from the root to a leaf, is $O(n^2|\mathscr{B}|)$ since the event "variable whose domain becomes singleton" can only happen once per variable and per branch.

Corollary 1. *The worst-case time complexity of propagate is $O(er^2d^r + n|\mathscr{B}|)$ where r is the greatest constraint arity.*

Proof. The cost of establishing GAC is $O(er^2d^r)$ when a generic algorithm such as GAC2001 [4] is used and the cost of exploiting nogoods has just been shown to be $O(n|\mathscr{B}|)$. □

Proposition 6. *The worst-case space complexity of storing reduced nld-nogoods is $O(n(d+|\mathscr{B}|))$.*

Proof. We know that $|\mathscr{B}|$ nogoods of size at most n are recorded. Further, the number of cells introduced to access nogoods is $O(|\mathscr{B}|)$ and the size of the array associated to each negative decision is $O(nd)$. We then obtain $O(n(d+|\mathscr{B}|))$. □

6. Minimal Reduced nld-Nogoods

In Section 3, we proved that nld-nogoods can be reduced in size by considering positive decisions only. Pursuing the same goal, we introduce in this section the concept of minimal reduced nld-nogood with respect to an inference operator ϕ. We can then obtain more powerful nogoods.

6.1 Minimal ϕ-Nogoods

In the context of a backtrack search algorithm, the ϕ operator can be employed at any step of a tree search, using a binary branching scheme. For example, MGAC corresponds to using an operator that enforces (Generalized) Arc Consistency at each node of the search tree.

Definition 6. *Let P be a CN, $\phi(P)$ is the CN obtained after applying the operator ϕ on P.*

If there exists a variable with an empty domain in $\phi(P)$ then P is clearly unsatisfiable, denoted $\phi(P) = \perp$.

Definition 7. *Let P be a CN and Δ be a set of decisions. Δ is a ϕ-nogood of P iff $\phi(P|_\Delta) = \perp$. Δ is a minimal ϕ-nogood of P iff $\nexists \Delta' \subset \Delta$ such that $\phi(P|_{\Delta'}) = \perp$.*

Obviously, ϕ-nogoods are nogoods, but the opposite is not necessarily true. It is easy to minimize a ϕ-nogood using a polynomial algorithm such as QuickXplain or one of its variants [21]. In our context, we know that at the end of each run, we can extract nld-nogoods from the current branch and reduce them. One interesting thing is that nld-nogoods which are not ϕ-nogoods can be directly identified and discarded. This is the case when the last decision δ_m of the nld-subsequence from which a nld-nogood Δ has been extracted, did not directly lead to a dead-end when applying ϕ. It means that we had to explore a non trivial subtree from that decision. On the other hand, when δ_m directly leads to a dead-end, we know that this decision necessarily belongs to any ϕ-nogood included in Δ. As a

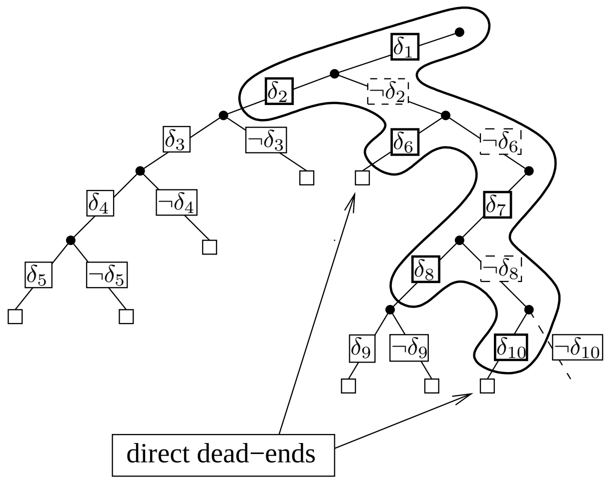


Figure 3. Identification of (reduced) nld-nogoods susceptible to be minimized

consequence, δ_m can be directly selected as the first transition decision of the minimization algorithm defined in the next subsection.

Figure 3 depicts a tree search for a run stopped after refuting δ_{10}, where an inference operator ϕ is maintained at each node. Among the four nld-nogoods that can be extracted, only two yield a direct dead-end: $\Delta_1 = \{\delta_1, \neg\delta_2, \delta_6\}$ and $\Delta_2 = \{\delta_1, \neg\delta_2, \neg\delta_6, \delta_7, \neg\delta_8, \delta_{10}\}$. Δ_1 and Δ_2 are clearly ϕ-nogoods, as the application of ϕ after each decision directly leads to an inconsistency. The reduced nld-nogoods that will be considered for minimization are then $\Delta'_1 = \{\delta_1, \delta_6\}$ and $\Delta'_2 = \{\delta_1, \delta_7, \delta_{10}\}$. Remark that a reduced nld-nogood obtained from a nld-nogood which is a ϕ-nogood is not necessarily a ϕ-nogood itself. Indeed, some negative decisions removed when reducing a nld-nogood may be involved in the conflict.

When reduced nld-nogoods can be highly minimized, the impact on subsequent runs can be quite important. In the best case, one can expect to isolate minimal ϕ-nogoods of size 1. They correspond to singleton ϕ-inconsistent values. For example, let us consider the queens-knights problem as proposed in [5] and the algorithm MAC to solve it. As any knight variable is singleton arc inconsistent, if such a variable is involved in the last decision of a reduced nld-nogood, then a singleton arc inconsistent value will be proved by the minimization algorithm. Some results that confirm this behaviour are given in Table 3 of Section 7.

6.2 Minimization Techniques

Some works concerning the identification of minimal ϕ-nogoods (also called conflict-sets) have already been proposed [21, 33]. As extracting a minimal ϕ-nogood is an activity limited to a branch of a search tree, the proposed algorithms involve (at least, partially) a constructive schema in order to keep some incrementality of the propagation process. On the other hand, the last version of QuickXplain [22] exploits a divide and conquer approach (as in [29]) but is defined in a more general context. For example, it can be used to extract

Minimal Unsatisfiable Cores (MUCs) of constraint networks which has been recently studied both theoretically and experimentally in [18].

To summarize, in order to find a minimal ϕ-nogood, it is necessary to iteratively identify the decisions that are involved in it. More precisely, we know that, given a ϕ-nogood $\Delta = \{\delta_1, \delta_2, \ldots, \delta_m\}$ of a CN P and a total ordering of the decisions (to simplify, we shall consider the natural order $\delta_1, \delta_2, \ldots, \delta_m$ of the decisions), there exists a decision δ_i such that $\phi(P|_{\{\delta_1,\ldots,\delta_{i-1}\}}) \neq \bot$ and $\phi(P|_{\{\delta_1,\ldots,\delta_i\}}) = \bot$. This decision which clearly belongs to a minimal ϕ-nogood will be called the *transition* decision of Δ (according to the given ordering). Note also that any decision δ_j with $j > i$ can be safely removed. This notion of transition decision is analogous to that of transition constraint defined in [18].

To identify a transition decision, it is possible to use a constructive approach, a destructive approach or a dichotomic one. The principle of the constructive approach is to successively add the decisions of Δ (according to the given ordering) to the CN until an inconsistency is detected when applying ϕ. The principle of the destructive approach is to initally add all decisions of Δ to the CN and successively remove them one by one until no more inconsistency is detected when applying ϕ. As a third alternative, the transition decision can be identified by using a dichotomic search.

To extract a minimal ϕ-nogood, it suffices to adopt one of the approaches described above. After finding a first[3.] transition decision δ_i in Δ, one can search a second one after having removed all decisions δ_j with $j > i$ of Δ (since unsatisfiability is preserved) and considering a new order of the decisions such that all found transition decisions are the smallest ones (the background of [22]). This process can be repeated until all decisions of the current nogood correspond to transition decisions that have been successively found. The principle of this iterative process has been described in [9, 21, 33, 18].

In the context of identifying a minimal nogood, we can relate the constructive, destructive and dichotomic approaches succinctly described above to the algorithms called RobustXplain, ReplayXplain and QuickXplain [21]. However, here, we will assume the incrementality of the inference operator ϕ. It simply means that the worst-case time complexities of applying ϕ on a given CN from two respective sets of decisions Δ and Δ' such that $\Delta \subset \Delta'$ are equivalent. For example, all (known) generic algorithms that enforce $\phi = (G)AC$ are incremental. As a consequence, using a constructive approach to identify a transition decision is well-adapted to our purpose. This algorithm will be used for our experimentation and its complexity is discussed in the next subsection.

6.3 Complexity Analysis

Proposition 7. *The worst-case time complexity of extracting a minimal GAC-nogood from a reduced nld-nogood is $O(enr^2d^r)$.*

Proof. We know that a generic algorithm such as GAC2001 to enforce GAC on a CN is incremental. As a consequence, using a constructive approach to identify a transition decision is $O(er^2d^r)$, that is to say the worst-case time complexity of establishing GAC only once. If the extracted nogood is composed of k decisions, then we obtain an overall complexity in $O(ker^2d^r)$. As k is $O(n)$, we obtain $O(enr^2d^r)$ $\square$

3. If Δ is a ϕ-nogood, then we can directly consider δ_m as the first transition decision.

Corollary 2. *In the binary case (i.e. for $r = 2$), the worst-case time complexity of extracting a minimal AC-nogood from a reduced nld-nogood is $O(end^2)$.*

Proposition 8. *The worst-case time complexity of extracting minimal reduced GAC-nld-nogoods (at the end of each run) is $O(en^2r^2d^{r+1})$.*

Proof. Extracting a GAC-nogood from a reduced nld-nogood is $O(enr^2d^r)$ and we know that there is at most $O(nd)$ reduced nld-nogoods to be minimized. □

Corollary 3. *In the binary case (i.e. for $r = 2$), the worst-case time complexity of recording minimal reduced GAC-nld-nogoods (at the end of each run) is $O(en^2d^3)$.*

7. Experiments

In order to show the practical interest of the approach described in this paper, we have conducted an extensive experimentation (on a Xeon processor cadenced at 3 GHz and 1GiB RAM). We have used the CSP solver Abscon whose kernel is a MGAC algorithm (embedding GAC3^{rm} [26]). Abscon can be considered as a state-of-the-art generic CSP solver, considering the results it obtained at the 2005 [39] and 2006 CSP solver competition[4]. We have studied the impact of exploiting restarts (denoted by MGAC+RST), nogood recording from restarts (denoted by MGAC+RST+NG) and the same technique with minimization (denoted by MGAC+RST+NGm). 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). This is a geometric restart policy as introduced in [40]. Note that we have tested other restart policies, but the relative behaviour of the algorithms described in this paper, remained relatively the same. In any case, an in-depth study of the impact of different restart policies on these algorithms is beyond the scope of this paper.

For search, 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, 25, 19, 39]. 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 four algorithms on the full set of 3,621 instances used as benchmarks for the first round of the 2006 CSP solver competition. The time limit to solve an instance was fixed to 20 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 four methods.

First, on random instances, for all the heuristics, it appears that restarting search is penalysing. This is not surprising since there is no structure to exploit from one run to the next one. This result confirms those observed on random SAT instances which do not exhibit any heavy-tailed phenomenon. However, by recording nogoods, we approximately obtain the same results than MGAC without restarts. Second, on structured instances, as

4. http://www.cril.univ-artois.fr/CPAI06

Table 1. Number of unsolved instances and average cpu time on the benchmarks of the 2006 CSP Solver Competition (first round), given 20 minutes.

		MGAC			
			+RST	*+RST+NG*	*+RST+NGm*
Random instances (1,390 instances)					
dom/ddeg	*#timeouts*	270	301	276	273
	avg time	40.4	57.6	41.9	42.0
brelaz	*#timeouts*	305	330	311	311
	avg time	73.2	103.2	70.5	71.1
dom/wdeg	*#timeouts*	266	278	274	268
	avg time	36.4	45.8	38.1	41.2
Structured instances (2,231 instances)					
dom/ddeg	*#timeouts*	873	863	825	772
	avg time	87.5	97.8	79.7	72.4
brelaz	*#timeouts*	789	788	757	738
	avg time	79.0	92.4	74.8	71.5
dom/wdeg	*#timeouts*	623	554	551	551
	avg time	50.5	51.3	51.4	50.8
All (3,621 instances)					
dom/ddeg	*#timeouts*	1,143	1,164	1,101	1,045
	avg time	66.1	79.6	62.6	58.6
brelaz	*#timeouts*	1,094	1,118	1,068	1,049
	avg time	76.4	97.3	72.8	71.3
dom/wdeg	*#timeouts*	889	832	825	819
	avg time	44.1	48.8	45.4	46.5

expected, recording nogoods from restarts is benefiting. Also, minimizing nogoods has a significant impact, in particular when classical heuristics are used. In an overall analysis, while restarting without learning yields mitigated results, nogood recording from restarts significantly improves the robustness of the solver. 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. Another view on results in given by Figures 4, 5 and 6 which represent scatter plots displaying pairwise comparisons for *dom/ddeg*, *brelaz* and *dom/wdeg*. Note the presence of many dots on the right-hand side of these figures which represent instances unsolved by the methods whose name label the x-axis.

When focusing to the hardest instances (which involve 680 variables and a greatest domain size of about 50 values) built from the real-world Radio Link Frequency Assignment Problem (RLFAP), we have observed (see Table 2) that using a restart policy allows to be more efficient by almost one order of magnitude. Here, performance is measured in terms of cpu time (in seconds), amount of used memory (in bytes) and number of visited nodes. When we further exploit nogood recording, the gain is about 10%. What is interesting to note here is that (w.r.t. our restart policy) recording nogoods from restarts does not require a lot of memory. Also, we noticed that the number and the size of the reduced nld-nogoods recorded during search were always very limited. As an illustration, MAC+RST+NG solved the instance $scen11\text{-}f1$ in 36 runs (and $3,750$ seconds) while only 712 nogoods of average size 8.5 and maximum size 33 were recorded.

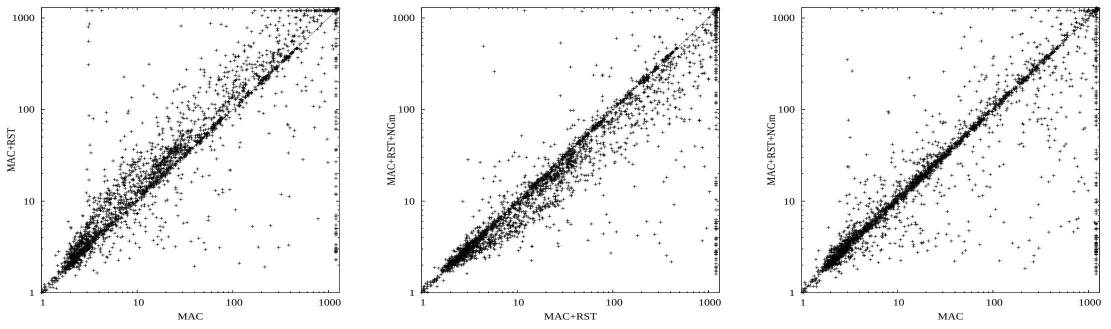


Figure 4. Pairwise comparison (cpu time) on the 3, 621 instances used as benchmarks of the 2006 CSP Solver Competition (first round). The variable ordering heuristic is dom/ddeg and the timeout to solve an instance is set to 20 minutes.

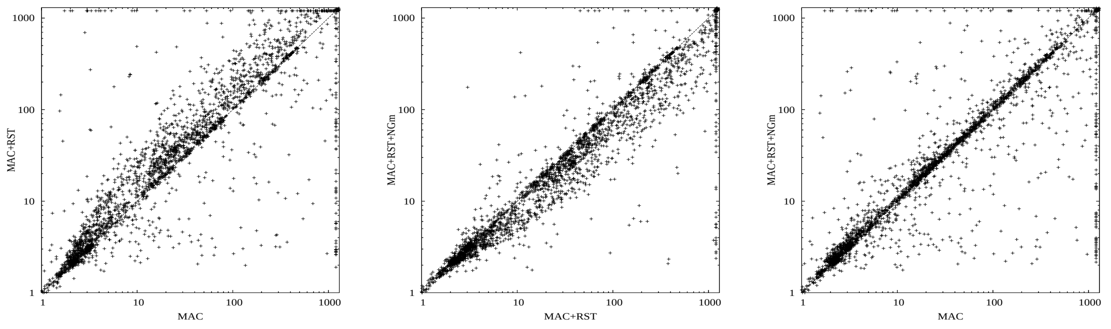


Figure 5. Pairwise comparison (cpu time) on the 3, 621 instances used as benchmarks of the 2006 CSP Solver Competition (first round). The variable ordering heuristic is brelaz and the timeout to solve an instance is set to 20 minutes.

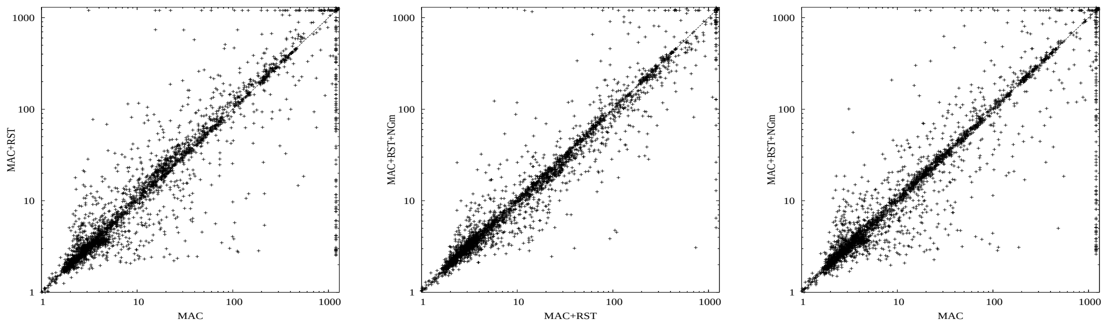


Figure 6. Pairwise comparison (cpu time) on the 3, 621 instances used as benchmarks of the 2006 CSP Solver Competition (first round). The variable ordering heuristic is dom/wdeg and the timeout to solve an instance is set to 20 minutes.

Table 2. Performance on hard RLFAP Instances using the *dom/wdeg* heuristic (timeout set to 20 minutes)

		MGAC			
			+RST	*+RST+NG*	*+RST+NGm*
scen11-f10	*cpu*	5.8	4.9	5.0	5.0
	mem	29M	31M	32M	32M
	nodes	891	403	405	556
scen11-f8	*cpu*	10.2	5.5	5.8	6.1
	mem	29M	31M	32M	32M
	nodes	15,045	1,149	1,098	1,287
scen11-f6	*cpu*	59.8	14.8	13.5	10.9
	mem	29M	31M	32M	32M
	nodes	217K	35,030	25,851	19,798
scen11-f4	*cpu*	924.1	141.1	116.6	125.9
	mem	30M	31M	32M	32M
	nodes	3,458K	494K	450K	476K
scen11-f3	*cpu*	*time-out*	370.6	361.5	342.5
	mem		31M	32M	32M
	nodes		1,506K	1,331K	1,314K
scen11-f2	*cpu*	*time-out*	*time-out*	1,135.3	1,137.7
	mem			32M	32M
	nodes			4,406K	4,370K

Finally, we present in Table 3 the results obtained for some instances of the queens-knights problem using *dom/wdeg*. As indicated in Section 6, minimizing nogoods is quite relevant on this kind of instances since singleton arc inconsistent values can be detected. Note that with the *dom/wdeg* heuristic, results are less impressive. Indeed, it is explained by the fact that this heuristic has a good capability of preventing thrashing.

Table 3. Cpu time to solve some instances of the queens-knights problem, given 20 minutes.

		MGAC			
			+RST	*+RST+NG*	*+RST+NGm*
qk-12-5-mul	*dom/ddeg*	265.1	408.9	256.2	2.1
	brelaz	255.7	377.9	250.8	2.1
	dom/wdeg	3.1	1.8	2.6	1.6
qk-25-5-mul	*dom/ddeg*	*time-out*	*time-out*	*time-out*	4.9
	brelaz	*time-out*	*time-out*	*time-out*	5.1
	dom/wdeg	*time-out*	4.2	4.8	4.3
qk-50-5-mul	*dom/ddeg*	*time-out*	*time-out*	*time-out*	67.3
	brelaz	*time-out*	*time-out*	*time-out*	65.3
	dom/wdeg	*time-out*	59.5	44.6	43.9

8. 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 structured 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 (that can be related 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.

Interestingly, this filtering algorithm can be embedded as a new propagator to any constraint propagation engine but it can also be easily integrated to any generic GAC algorithm as we have shown in this paper. In our approach, reduced nld-nogoods correspond to positive decisions only, and so, can be classified as standard nogoods. One advantage is that the complexity of managing the base of nogoods is small since the only event we need to intercept is when a variable becomes fixed (its domain becoming a singleton). In [24], it has been shown that generalized nogoods are more powerful than standard nogoods. However, it is not immediate if this remains true when a binary branching scheme is used [20]. In any case, it appears that one perspective to the approach proposed in this paper is to compute generalized nogoods, that is to say to extract minimal (not reduced) nld-nogoods from the last branch. The theoretical and practical aspects of this alternative deserve to be studied.

Acknowledgments

The authors would like to thank the anonymous reviewers for their useful comments which helped to improve the paper. This paper has been supported by the CNRS, the ANR "Planevo" project n^{o}JC05_41940 and the "IUT de Lens".

References

[1] L. Baptista, I. Lynce, and J. Marques-Silva. Complete search restart strategies for satisfiability. 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. *Artificial 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] J.L. de Siqueira and J.F. Puget. Explanation-based generalisation of failures. In *Proceedings of ECAI'88*, pages 339–344, 1988.

[10] R. Dechter. Enhancement schemes for constraint processing: backjumping, learning and cutset decomposition. *Artificial Intelligence*, **41**:273–312, 1990.

[11] R. Dechter. *Constraint processing.* Morgan Kaufmann, 2003.

[12] N. Eén and N. Sorensson. An extensible sat-solver. In *Proceedings of SAT'03*, 2003.

[13] F. Focacci and M. Milano. Global cut framework for removing symmetries. In *Proceedings of CP'01*, pages 77–92, 2001.

[14] D. Frost and R. Dechter. Dead-end driven learning. In *Proceedings of AAAI'94*, pages 294–300, 1994.

[15] A.S. Fukunaga. Complete restart strategies using a compact representation of the explored search space. In *Proceedings of SSA'03 workshop held with IJCAI'03*, 2003.

[16] M. Ginsberg. Dynamic backtracking. *Artificial Intelligence*, **1**:25–46, 1993.

[17] 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.

[18] F. Hemery, C. Lecoutre, L. Sais, and F. Boussemart. Extracting MUCs from constraint networks. In *Proceedings of ECAI'06*, pages 113–117, 2006.

[19] T. Hulubei and B. O'Sullivan. Search heuristics and heavy-tailed behaviour. In *Proceedings of CP'05*, pages 328–342, 2005.

[20] J. Hwang and D.G. Mitchell. 2-way vs d-way branching for CSP. In *Proceedings of CP'05*, pages 343–357, 2005.

[21] U. Junker. QuickXplain: conflict detection for abitrary constraint propagation algorithms. In *Proceedings of IJCAI'01 Workshop on modelling and solving problems with constraints*, pages 75–82, 2001.

[22] U. Junker. QuickXplain: preferred explanations and relaxations for over-constrained problems. In *Proceedings of AAAI'04*, pages 167–172, 2004.

[23] G. Katsirelos and F. Bacchus. Unrestricted nogood recording in CSP search. In *Proceedings of CP'03*, pages 873–877, 2003.

[24] G. Katsirelos and F. Bacchus. Generalized nogoods in CSPs. In *Proceedings of AAAI'05*, pages 390–396, 2005.

[25] C. Lecoutre, F. Boussemart, and F. Hemery. Backjump-based techniques versus conflict-directed heuristics. In *Proceedings of ICTAI'04*, pages 549–557, 2004.

[26] C. Lecoutre and F. Hemery. A study of residual supports in arc consistency. In *Proceedings of IJCAI'07*, pages 125–130, 2007.

[27] C. Lecoutre, L. Sais, S. Tabary and V. Vidal. Nogood Recording from Restarts. In *Proceedings of IJCAI'07*, pages 131–136, 2007.

[28] 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.

[29] J. Mauss and M. Tatar. Computing minimal conflicts for rich constraint languages. In *Proceedings of ECAI'02*, pages 151–155, 2002.

[30] J.J. McGregor. Relational consistency algorithms and their application in finding subgraph and graph isomorphisms. *Information Sciences*, **19**:229–250, 1979.

[31] D.G. Mitchell. Resolution and constraint satisfaction. In *Proceedings of CP'03*, pages 555–569, 2003.

[32] 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.

[33] T. Petit, C. Bessiere, and J.C. Régin. A general conflict-set based framework for partial constraint satisfaction. In *Proceedings of SOFT'03 workshop held with CP'03*, 2003.

[34] P. Prosser. Hybrid algorithms for the constraint satisfaction problems. *Computational Intelligence*, **9**(3):268–299, 1993.

[35] J.F. Puget. Symmetry breaking revisited. *Constraints*, **10**(1):23–46, 2005.

[36] D. Sabin and E. Freuder. Contradicting conventional wisdom in constraint satisfaction. In *Proceedings of CP'94*, pages 10–20, 1994.

[37] 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.

[38] C. Schulte and M. Carlsson. Finite domain constraint programming systems. In *Handbook of Constraint Programming*, chapter 14, pages 495–526. 2006.

[39] M.R.C. van Dongen, editor. *Proceedings of CPAI'05 workshop held with CP'05*, volume **II**, 2005.

[40] T. Walsh. Search in a small world. In *Proceedings of IJCAI'99*, pages 1172–1177, 1999.

[41] H. Zhang. A random jump strategy for combinatorial search. In *Proceedings of AI&M'02*, 2002.

[42] L. Zhang and S. Malik. The quest for efficient boolean satisfiability solvers. In *Proceedings of CADE'02*, pages 295–313, 2002.