EPiC Series in Computing
Volume 73, 2020, Pages 249–259
LPAR23. LPAR-23: 23rd International Conference on Logic for Programming, Artificial Intelligence and Reasoning

NACRE - A Nogood And Clause Reasoning Engine *

Gael Glorian[1], Jean-Marie Lagniez[2], and Christophe Lecoutre[3]

[1] LaBRI – CNRS UMR 5800, Université de Bordeaux, Talence, Nouvelle-Aquitaine, France
gael.glorian@labri.fr
[2] Huawei Technologies Ltd, Boulogne-Billancourt, Île-de-France, France
jean.marie.lagniez@huawei.com
[3] CRIL – CNRS UMR 8188, Université d'Artois, Lens, Hauts-de-France, France
lecoutre@cril.fr

Abstract

NACRE, for Nogood And Clause Reasoning Engine, is a constraint solver written in C++. It is based on a modular architecture designed to work with generic constraints while implementing several state-of-the-art search methods and heuristics. Interestingly, its data structures have been carefully designed to play around nogoods and clauses, making it suitable for implementing learning strategies. NACRE was submitted to the `CSP MiniTrack` of the 2018 and 2019 XCSP3 [8] competitions where it took the first place. This paper gives a general description of NACRE as a framework. We present its kernel, the available search algorithms, and the default settings (notably, used for XCSP3 competitions), which makes NACRE efficient in practice when used as a black-box solver.

1 Introduction

Many research efforts have been devoted to the design of general tools and algorithms for solving CSP (Constraint Satisfaction Problem) instances [33]. The current mainstream approaches rely on complete methods that embed filtering procedures, search heuristics and learning mechanisms. Implementing a new technique into a CSP solver usually requires a strong knowledge of its whole structure. Moreover, to be fully accepted by the community, a comparison with the state-of-the-art is necessary, with careful re-implementation.

During the last two decades, the impressive progress in the related problem SAT (Satisfiability Testing), where variables are Boolean, has been achieved using nogood recording (called clause learning when nogoods are managed à la SAT) under a restart policy enhanced by a very efficient lazy data structure [27]. The value of clause learning has risen with the availability of large instances (encoding practical applications). Learning in SAT is an example of a successful technique derived from cross-fertilization between CSP and SAT: nogood recording [10,36] and conflict-directed back jumping [31] were originally introduced for CSP and were later imported into SAT solvers [4,37]. Progress in SAT has stimulated a renewed interest of the CSP community in nogood recording [9,19,22,32]. While some CSP solvers embed a clause

*This work has been supported by the project CPER Data from the region "Hauts-de-France".

learning component, some others require to use an external SAT solver or are built above a SAT solver. In some cases, the whole problem is encoded into CNF [38, 42] or incrementally translated [12, 29]: the SAT solver often takes priority over the CSP solver for guiding search.

NACRE has been designed to be a hybrid solver, solving problem instances with either dedicated methods or SAT-inspired ones. Utilizing dedicated data structures, it can work efficiently with clauses (or nogoods); actually, the core of NACRE was conceived with this general objective in mind. Furthermore, NACRE has been conceived as a framework: it is designed to be upgradable, letting the user try new ideas on top of filtering or solving methods in a straightforward way. Moreover, it is possible to easily implement new reasoning methods concerning clauses and to specify minimization techniques to shorten them. Although NACRE has been conceived as a framework, its performances have not been left aside, as shown by the first place obtained in the `CSP MiniTrack` of the 2018 and 2019 XCSP3 competitions [8]. This track is for open-source mini-solvers[1] and aims at discovering new ideas and solvers.

2 Preliminaries

A Constraint Network (CN) [20] $\mathcal{P}$ is composed of a finite set of variables $\mathcal{X}$ and a finite set of constraints $\mathcal{C}$. Each *variable* x has a (current) domain $\texttt{dom}(x)$, which is the finite set of values that can be assigned to x; the initial domain of a variable x is denoted by $\texttt{dom}^{init}(x)$. Each *constraint* c involves an ordered set of variables $\texttt{scp}(c)$, called the *scope* of c. A constraint c is semantically defined by a relation $\texttt{rel}(c)$, which is the set of tuples allowed by (variables of) c. A tuple τ can be perceived as the instantiation of a subset of variables of $\mathcal{X}$; we note $\tau[x]$ the value of variable x in τ. A solution of a CN $\mathcal{P}$ is an instantiation of $\mathcal{X}$ such that all constraints $c \in \mathcal{C}$ are satisfied. A constraint $c \in \mathcal{C}$ is satisfied by an instantiation τ if and only if τ covers (i.e., instantiates) all the variables in the scope of c and τ corresponds to a tuple accepted by c. A CN is also usually referred to as a CSP instance.

To find solutions efficiently, it is important to filter the search space, typically by enforcing some local properties (called consistencies). Generalized Arc Consistency (GAC) [26] is a very classical one. For a constraint c, it guarantees that each pair (x, a) with $x \in \texttt{scp}(c)$ and $a \in \texttt{dom}(x)$, admits a support on c, i.e., a tuple $\tau \in \Pi_{x \in \texttt{scp}(c)} \texttt{dom}(x)$ such that $\tau[x] = a$ and $\tau \in \texttt{rel}(c)$. Constraint propagation is the process of enforcing a local consistency such as GAC, step by step, by soliciting all constraints to filter variable domains in turn until a fixed point is reached. A *literal on* a variable x corresponds to either an assignment of the form $x \leftarrow a$ or a refutation of the form $x \nleftarrow a$, with $a \in \texttt{dom}^{init}(x)$. The literals $x \leftarrow a$ and $x \nleftarrow a$ are said to be *positive* and *negative*, respectively. In the following, we also denote literals with Greek letters $(\lambda, \delta, \ldots)$. A decision δ is checked (in our algorithms) to be positive or negative by simply writing $pos(\delta)$ and $neg(\delta)$, respectively.

In this paper, we focus on backtrack search where decisions are taken in sequence to find solutions, and we assume that decisions always correspond to literals, which is a very classical approach in constraint programming. For example, MAC (Maintaining Arc Consistency) [34] builds a search tree by systematically maintaining GAC after such (positive and negative) decisions. For the sake of precision, *decisions* refer to literals (unary constraints) added to the CN by the search algorithm when extending the current branch of the search tree, whereas *deductions* refer to inferences performed by the filtering process (e.g., the algorithm that enforces GAC). Deductions are also assumed to be literals (which is always the case with GAC).

[1] NACRE source is available at `https://github.com/crillab/nacre_mini`.

3 Inside NACRE

We first succinctly introduce the data structures employed in NACRE before presenting the main components of the solver and the way nogood recording is conducted.

3.1 Data Structures

In NACRE, variables and values are represented through propositional variables: every pair (x, a) with $x \in \mathcal{X}$ and $a \in dom(x)$ is paired to an integer representing the propositional variable x_a (direct encoding [39], where each CSP variable is represented by a sequence of propositional variables). Therefore, the variables are stored as virtual objects, meaning that each variable does not directly have its own domain but only knows an index where its domain starts (in a big contiguous table) as well as its initial domain size and its current domain bounds.

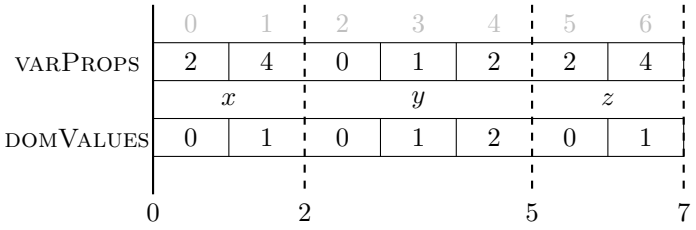

Figure 1: Data Structures for Variables/Values in NACRE.

There are two tables of this form (three in the case of clause learning). The first one, called varProps, stores the propositional variables and can be seen as the dense part of a full sparse set [1]. The propositional variables (referred to as PV) are objects that contain useful information such as the value of the PV into the original CSP variable, and also: (i) its current state (0 meaning that the value has been assigned to the CSP variable, 1 meaning that the value has been removed, and 2 when neither assigned nor removed), (ii) its *local*[2] position in the table domValues described below, and (iii) a few Boolean flags to record if this is the last assigned value (by the solver), if it was used during the process of conflict analysis, etc. The second table, called domValues, is similar to a full reversible sparse set, i.e., it stores for each variable a two-part table separated by a limit which is the current size of the variable domain. The values on the left of this limit are still active into the variable domain, whereas the ones on the right have been removed. It allows us to remove and restore values easily and efficiently since it only requires two basic operations (swapping two integers and decrementing a limit) to either remove a value or assign a variable. This lightweight structure is almost backtrack-free as we just need to store the local position of the limit at each decision level.

Figure 1 shows an example on three variables x, y and z with $\texttt{dom}(y) = \{0, 1, 2\}$ and $\texttt{dom}(x) = \texttt{dom}(z) = \{2, 4\}$ as defined initially (initial state). The upper table is varProps, in which we can clearly see the values of the domain (we do not show the other PV information in the figure, for clarity). The lower table is domValues containing the local positions. We can observe that they are initially sorted in an increasing manner. Above these two tables, in light gray, we can see the integers associated to the propositional variables (which are essentially the indices of the table varProps).

[2]We call *local* a position that has been normalized according to the current variable domain starting index.

3.2 Solver Components

Search Tree Exploration. NACRE provides various methods to solve CSP instances, given under XCSP3 format. Being a framework designed to handle clauses and nogoods, several state-of-the-art procedures are implemented. The first one is a standard MAC [34,35] approach, used with the method option `-complete`. It maintains arc-consistency while running a backtrack search based on binary branching. Binary branching means that at each search node, a left branch labeled with a positive decision $x = a$ is developed first, and a right branch labeled with a negative decision $x \neq a$ is developed next. To do this efficiently, NACRE uses time-stamps for both the constraints and the variables. Recall that a time-stamp is a value denoting the time at which a certain event occurred; time-stamps allow the progress of algorithms to be tracked over time. Technically, with time-stamps, a set is not required to represent the propagation queue, which renders propagation less expensive to deal with.

Algorithm 1: propagate(x: Variable): Boolean

```
1 Q ← {x};
2 while Q ≠ ∅ do
3     y ← pickVariable(Q);
4     forall c ∈ C | y ∈ scp(c) ∧ stamp[c] < stamp[y] do
5         touched ← filter(c) ;
6         if touched = WIPE_OUT then return true;        // Detected conflict
7         Q ← Q ∪ touched;
8         updateStamp(c);
9 return false                                           // No detected conflict
```

Algorithm 1 shows how propagation is conducted after the solver has decided on a variable x. Until a fixed point is reached, it picks (line 3) a variable y from Q (the propagation queue only contains x initially) by using Function pickVariable() that selects (and removes) a variable from Q. Then, we iterate over the constraints involving this variable (line 4), and the (time-)stamp of the current variable is compared with the (time-)stamp of the constraint. The rationale (discussed below) is that it allows us to determine whether it is useful to call the filtering algorithm (propagator), Function filter() at line 5, attached to the constraint c. This function returns the set of variables, `touched`, whose domains have been reduced by the propagator in order to add them to the propagation queue (line 7) or the special value $WIPE\_OUT$ when a conflict occurs. Concerning stamping, note that the stamp of a constraint is updated at line 8, whereas the stamp of a variable is updated when it is added to Q (lines 1 and 7), meaning that its domain has just been modified (or *touched*). This stamping system avoids unnecessary calls to constraint propagators, which could be very expensive. Indeed, when a constraint c is asked to filter, all variables in $scp(c)$ are GAC regarding c (that is ensured by Function filter() at line 5) and $\texttt{stamp}[c]$ is updated in such a way that $\texttt{stamp}[c] \geq \texttt{stamp}[x]$ for all $x \in scp(c)$. It is clear that it is unnecessary to consider c again before any variable of its scope is touched. When a variable x is touched, then $\texttt{stamp}[x]$ is updated, and x is added to the propagation queue. Therefore, if a variable x of $scp(c)$ is touched after c is revised then $\texttt{stamp}[x] > \texttt{stamp}[c]$ and consequently the condition (line 4) is satisfied when x is picked from the queue and the constraint is considered again to be revised.

Search Heuristics. In NACRE, we have a small pool of variable ordering and value ordering heuristics for tuning search. The variable ordering heuristic **dom** [6, 17] (option `-dom`) was the first popular dynamic heuristic. It uses the current size of the variable domains to select the next variable to instantiate; the smaller, the more preferred. A second variable ordering heuristic is **dom/deg** [5] (option `-domdeg`), a dynamic heuristic that combines both the domain size and the degree of the variables. The degree of a variable is the number of constraints where the variable is involved. This heuristic selects the variable with the smallest ratio of its current domain size to its degree. A third variable ordering heuristic is the robust adaptive heuristic **dom/wdeg** [7, 40] (option `-domwdeg`, used by default) that aims at choosing in priority the variables involved in the most falsified constraints. Technically, in each constraint object, a counter (initialized to 1) is incremented whenever the constraint is at fault (i.e., fails while filtering) during the search.

Once a variable is selected, a value must be chosen from its domain. Different value ordering heuristics are available in NACRE. In addition to the classical value heuristics (min-value, max-value, ...), a weak version of the SAT-based heuristic `polarity` has been implemented [30]. This heuristic, called *saving* option (`-saving`), tries to assign the selected variable to the last value assigned to it. If the choice is not valid anymore, meaning that the value has already been deleted, then the underlying value heuristic is solicited. Two main options are available, each one having its opposite. On the one hand, with **min-value** (option `-valMin`), we choose the minimal value in the domain of the variable; it is also possible to select the *max-value* inside NACRE (option `-valMax`). Choosing the minimal or maximal value, depending on the problem, can greatly improve the performance of the solver. On the other hand, the heuristic **first-value** (option `-valFirst`), called *lexico* sometimes, is a second way of selecting values, based on the implementation of the domains. As we use structures similar to sparse sets, we can select the first PV inside the DOMVALUES set. We can also select the last one before the limit (option `-valLast`). It is also possible to fully randomize value selection (option `-valRand`).

Restart Policies. NACRE has some built-in restart policies that allow us to test nogood or clause learning under different settings. Because restart policies can have a strong impact on search efficiency, we have implemented the most popular policies, from geometric progression to Luby sequence [25]. The first restart policy available is the no restarting policy (option `-noRst`), sometimes useful to be used as a baseline. Then comes the Luby-based sequences: first terms are of the form $(1, 1, 2, 1, 1, 2, 4, 1, \ldots)$. There are 3 available Luby-based sequences, `Luby(#rst)` $\times N$ where N can take the values 10, 50 and 100 (options are respectively `-luby10`, `-luby50` and `-luby100`) and $\#rst$ is the number of runs already done in order to get the correct term from the sequence. This is useful for trying more or less aggressive restarts policies. Finally, geometric sequences are available for a more constant progression of the cutoff value. Four of them are implemented inside NACRE. The default one has a 3% progression, others are 10% (option `-10perc`), 50% (option `-50perc`), 100% (option `-double`). It allows the user to choose from a wide variety of policies, even possibly changing it during search with a few lines of code.

3.3 Nogood Recording

The benefit of recording nogoods [36] is to avoid some form of thrashing [16], i.e., exploring the same unsatisfiable subtrees several times. There are two classical methods to identify and store nogoods: during the search or at restarts. Nogood from restart extracts information by analyzing the current search tree when a restart is triggered. Two well-known and effective methods have been implemented into our framework, the negative last-decision nogoods [23]

(nld-nogoods) and the increasing-nogoods [24]. Nld-nogoods (option `-nld`) are extracted at each restarts from the last branch of the search tree. Let us assume that the sequence of labels all along the rightmost branch of the current search tree being developed is $\Sigma = \langle \delta_1, \ldots, \delta_m \rangle$, where each decision of Σ is either a positive or a negative decision. It is known that for any i such that $1 \leq i \leq m$ and $neg(\delta_i)$, the set $\Delta = \{\delta_j : 1 \leq j < i \wedge pos(\delta_j)\} \cup \{\neg\delta_i\}$ is a reduced nld-nogood. A reduced nld-nogood $\{\delta_i, \delta_j\}$ can be written $\bigwedge_{\delta_i \in \Delta} \delta_i \Rightarrow \neg\delta_j$ in directed form. Increasing-nogoods (method option `-incng`) are a compressed form of all the reduced nld-nogoods that can be extracted at each restart, without losing any effectiveness. Indeed, it is rather easy to deduce that there are some similarities between nld-nogoods: the left part of the implication is shared between several nogoods. Then, it is possible to rewrite the set of nld-nogoods as a constraint and use a dedicated propagator that ensures GAC.

Another way to learn nogoods is during search when a conflict occurs. The concepts of nogoods and clauses are closely related. The former is mostly used in the CSP context and the latter in the SAT one. NACRE implements a generic clause reasoning engine using lazy explanations (option `-ca`). We consider the g-nogood learning procedure [18], and the lazy explanations, proposed in [13, 14] as a starting point. A way of generating a nontrivial nogood at each conflict consists in deeply analyzing the sequence of propagation steps by constructing an implication graph *à la* SAT and identifying a reason of the failure [41]. An implication graph is a directed acyclic graph (DAG) that records the relationships existing between literals, as can be observed during the solving process. Each vertex represents a literal λ (with its associated level), and its incoming arcs in the DAG represent the reasons that force it. When a conflict occurs, each cut in the implication graph that leads to the conflict can explain it. This set of literals, used as a conflict explanation, can be blocked in the future by deriving a clause from it. Generally, we are only interested in Unique Implication Points (UIPs) [37] that are vertices at the current decision level dominating the conflict. Importantly, they can be used to safely perform non-chronological backtracking (a form of back jumps) up to the decision level that is the maximum value among all decision levels associated with the literals of the conflict explanation.

In practice, the explicit construction of the implication graph can be avoided. Indeed, conflict clauses can be derived from the current formula if we simply save independently the explanation $\texttt{expl}(\lambda)$ of each deduction λ. More precisely, assuming that all such explanations are stored, and starting from a conflicting set Λ of literals, the conflict analysis procedure iteratively selects a literal $\lambda \in \Lambda$ implied at the most recent decision level (i.e., the current largest decision level) and replaces λ by $\texttt{expl}(\lambda)$ in Λ. This process is stopped when only one literal from the current level remains in Λ. Consequently, the literals remaining in Λ, after the procedure is complete, forms a UIP. Because Λ represents an explanation for the conflict, the clause $\bigvee_{\lambda \in \Lambda} \neg\lambda$ is a logical consequence of the problem (at least one literal of Λ must be false). We can observe that upon backtracking this clause remains unit (with literal λ_{unit}) until the highest decision level of the other literals in $\Lambda \setminus \{\lambda_{unit}\}$ is reached.

On one hand, SAT formalism is very simple (each time a literal is deduced, its explanation can be computed by considering the clause that triggers it) and very well suited to nogood extraction by an analysis of the implication graph. On the other hand, CSP formalism is more sophisticated (e.g., dealing with global constraints) and makes more difficult the precise identification of the reason of a simple deduction: building the implication graph is no more straightforward. Moreover, even if we may consider constructing such graphs, computing the explanation of each deduction can drastically slow down the search algorithm if no proper care is taken. In NACRE, a generic non-intrusive method that allows us to compute explanations has been implemented.

Explanations are generated on demand in three situations. Firstly, if a literal λ is propagated from a clause Λ, then the explanation is given by the negation of literals of $\Lambda \setminus \{\lambda\}$. Secondly, a literal can be propagated by the domain constraints: `atMostOne` and `atLeastOne`. The `atLeastOne` constraint is used as explanation when only one value a is remaining is the domain of x, in this case we have $\texttt{expl}(x = a) = \{x \neq b : b \in \texttt{dom}^{init}(x) \setminus \{a\}\}$. The `atMostOne` constraint is used as explanation when a value b is deleted from x when x is assigned to a, in this case we have $\texttt{expl}(x \neq b) = \{x = a\}$. Finally, a literal λ from x can be propagated while running the filtering algorithm ϕ_c associated with a constraint c that is not a clause. In this case, it is possible to explain why λ has been propagated by considering the deleted values for the variables of $scp(c) \setminus \{x\}$. To avoid systematic computation of explanations at each propagation, NACRE uses a stamp system that associates an integer with each value. When a value a from x is deleted its stamp is updated in such a way that $stamp(x, a)$ is greater than all $stamp(x', a')$ such that a' is deleted from x'. In order to be backtrack free, we use an integer that is incremented at each deletion. Thus, when an explanation is required for a value a from x, it is enough to keep in memory which constraint has been used to perform the propagation. Indeed, it is enough to consider the value a' of $x' \in scp(c) \setminus \{x\}$ such that $stamp(x, a) > stamp(x', a')$. Once the conflict analysis process is over, a new clause is added to the nogood base and a non-chronological backtrack is performed. Of course, for managing the nogood base, we adopt the classical *two-watched-literals strategy* [28].

Algorithm 2 is an variation of Algorithm 1 that handles clause propagation too. We choose to fully propagate the constraints (line 3 to 10) before the clauses (line 12 to 18). Indeed, playing with clauses and nogoods can lure an adaptive heuristic like `dom/wdeg` [15]. Triggering in priority conflicts from the CSP core avoids this kind of behavior. These two propagation processes are done until a fixed point is reached (when clause propagation loop – `bcp`– becomes useless). The constraint propagation loop is similar to Algorithm 1 with one slight difference at line 5. The picked variable is added into a new queue Q^{SAT}. It allows the loop at line 12 to iterate over all the variables that were touched during GAC enforcement.

Clause Database Reduction. The procedure that ensures unit propagation on nogoods (clauses) highly depends on the size of the base. To maintain manageable the base of nogoods, and perform unit propagation at a reasonable cost, it is common to reduce the base by deleting clauses considered to be irrelevant to the next search steps. Several measures have been proposed for this purpose [2, 3, 11]. Generally, SAT solvers use one of the following strategies: *activity* that considers a learned clause as irrelevant if its activity or its involvement in recent conflict analysis is marginal [11]; or *literal block distance (LBD)* that uses the number of different levels involved in a given learned clause to quantify the quality of learned clauses [3]; clauses with smaller LBD are considered more relevant.

Even if these measures are well adapted for SAT, there are not well suited in our case: a hybrid solving method requires a hybrid clause quality measure. Indeed, it is often the case that clauses contain a large number of literals. But contrary to SAT, it is possible to identify literals that are linked by a common CSP variable. In this case, clauses that are related to few CSP variables are more powerful regarding unit propagation. To take this information into account, we propose to use a new measure that considers the number of CSP variables involved in the clause through its associated propositional variables. When two clauses have the same number of CSP variables, then their activity is used to decide between them. An important point regarding the reduction of the base concerns its frequency. In NACRE, we chose to set the initial size limit of the base to $4,000$ nogoods (pragmatic choice, made from experiments). When the limit is reached, half of the learned clauses are removed using our custom measure.

Algorithm 2: propagate^{hybrid}(x: Variable): Constraint

```
1  Q^SAT ← ∅, Q ← {x};
2  repeat
3      while Q ≠ ∅ do
4          y ← pickVariable(Q);
5          Q^SAT ← Q^SAT ∪ {y};
6          forall c ∈ C | y ∈ scp(c) ∧ stamp[c] < stamp[y] do
7              touched ← filter(c);
8              if touched = WIPE_OUT then return c ;      // conflict constraint
9              Q ← Q ∪ touched;
10             updateStamp(c);
11     continue ← false;
12     while Q^SAT ≠ ∅ do
13         v ← pickVariable(Q^SAT);
14         (c, touched) ← bcp(v) ;
15         if c ≠ null then return c ;                    // conflict clause
16         if touched ≠ ∅ then
17             continue ← true;
18             Q ← Q ∪ touched;
19 until ¬continue;
20 return null ;                                          // no detected conflict
```

After each reduction, the size limit is increased by 500.

4 NACRE at XCSP3 Competitions

NACRE was submitted to the `CSP MiniTrack` of the XCSP3 competition [21]. This track is designed for sequential open-source software and allows constraint solver developers to enter the competition without having to implement all constraints from the so-called XCSP3-core. Solvers are evaluated on a restricted set of constraints: `intension`, `extension`, `allDifferent`, `sum` and `element`. Solvers were run on a cluster of Xeon@2.67GHz with 32GiB of memory. Sequential solvers were allocated 15500 MiB of memory and a time limit of 2400 seconds.

The 2018 (resp. 2019) `CSP MiniTrack` was a selection of 176 (resp. 200) instances. NACRE competed in its hybrid form with the following options: `-ca -luby100` and `-cm`[3]. It enables our conflict analysis method and uses the Luby sequence as restart policy. In 2019, we also submitted the standard MAC version (options: `-complete -l100 -cm`) to the competition in order to compare them.

Table 1[5] shows that, in 2018 (resp. 2019), NACRE solved 49% (68%) of the 176 (200) instances, which represents 76% (78%) of the Virtual Best Solver (`VBS`). It means that 76% (78%) of the instances that can be solved by any competing solvers are solved by NACRE. We can also see that NACRE performs well on both `SAT` and `UNSAT` instances. The impact of clause

[3]The `-cm` option stands for verbose mode 0: only the result and the solution, if one is found, are displayed.
[4]PicatSAT was not an official competitor but is given as a comparison with the main competition track.
[5]Generated from the raw data of the official competition website: `http://www.xcsp.org/competition`.

Table 1: `CSP MiniTrack` – Results in 2018 (Top 5) and 2019.

Rank	Solver (Version)	#solved(%)	#SAT/#UNS	% VBS	SumCPU	MedCPU	AvgCPU
	Virtual Best Solver (2018)	113 (64)	53/60	100	11899.26	0.62	105.30
1	NACRE (1.0.4 - Hybrid)	86 (49)	43/43	76	9948.56	0.40	115.68
2	miniBTD_12 (180727_12)	79 (45)	36/43	70	9534.55	0.88	120.69
3	miniBTD (180727_3)	75 (43)	32/43	66	13679.82	1.17	182.40
4	cosoco (1.12)	72 (41)	42/30	64	14074.67	2.12	195.48
5	minimacht (180727)	69 (39)	37/32	61	13299.29	4.85	192.74
	Virtual Best Solver (2019)	172 (86)	109/63	100	18636.55	4.37	108.35
-	*PicatSAT (Main track reference)* [4]	148 (74)	97/51	86	41856.68	105.79	282.82
1	NACRE (1.0.5 - Hybrid)	135 (68)	91/44	78	22691.90	8.80	168.09
2	miniBTD (19.06.16)	133 (67)	89/44	77	16899.04	7.75	127.06
3	cosoco (2.0)	127 (64)	85/42	74	25518.00	3.11	200.93
4	NACRE (1.0.5 - MAC)	116 (58)	81/35	67	20797.49	8.38	179.29

learning is stronger on the 2019 results (17% more solved instances).

5 Conclusion

In this paper, we have presented NACRE, a nogood and clause reasoning engine. Our main goal is to provide the community with an expansible tool, useful in particular for easily experimenting with nogoods and clauses. We paid attention to practical performances, and we demonstrated that in the 2018 and 2019 XCSP3 competitions. For the next major version, we project to handle all constraints from XCSP3-core.

References

[1] Alfred V. Aho, John E. Hopcroft, and Jeffrey D. Ullman. *The Design and Analysis of Computer Algorithms*. Addison-Wesley, 1974.

[2] Gilles Audemard, Jean-Marie Lagniez, Bertrand Mazure, and Lakhdar Sais. On freezing and reactivating learnt clauses. In *SAT*, pages 188–200. Springer, 2011.

[3] Gilles Audemard and Laurent Simon. Predicting learnt clauses quality in modern SAT solvers. In *IJCAI*, pages 399–404, 2009.

[4] Roberto J. Bayardo and Robert Schrag. Using CSP look-back techniques to solve real-world SAT instances. In *AAAI/IAAI*, pages 203–208, 1997.

[5] Christian Bessière and Jean-Charles Régin. MAC and combined heuristics: Two reasons to forsake FC (and CBJ?) on hard problems. In *CP*, pages 61–75. Springer, 1996.

[6] James R. Bitner and Edward M. Reingold. Backtrack programming techniques. *Commun. ACM*, 18(11):651–656, 1975.

[7] Frédéric Boussemart, Fred Hemery, Christophe Lecoutre, and Lakhdar Sais. Boosting systematic search by weighting constraints. In *ECAI*, pages 146–150. IOS Press, 2004.

[8] Frédéric Boussemart, Christophe Lecoutre, Gilles Audemard, and Cédric Piette. XCSP3: an integrated format for benchmarking combinatorial constrained problems. *CoRR*, abs/1611.03398, 2016.

[9] Karim Boutaleb, Philippe Jégou, and Cyril Terrioux. (no)good recording and robdds for solving structured (V)CSPs. In *ICTAI*, pages 297–304, 2006.

[10] Rina Dechter. Enhancement schemes for constraint processing: Backjumping, learning, and cutset decomposition. *Artif. Intell.*, 41(3):273–312, 1990.

[11] Niklas Eén and Niklas Sörensson. An extensible sat-solver. In *SAT*, pages 502–518. Springer, 2003.

[12] Thibaut Feydy and Peter J. Stuckey. Lazy clause generation reengineered. In *CP*, pages 352–366. Springer, 2009.

[13] Ian P. Gent, Christopher Jefferson, Lars Kotthoff, Ian Miguel, Neil C. A. Moore, Peter Nightingale, and Karen E. Petrie. Learning when to use lazy learning in constraint solving. In *ECAI*, pages 873–878. IOS Press, 2010.

[14] Ian P. Gent, Ian Miguel, and Neil C. A. Moore. Lazy explanations for constraint propagators. In *PADL*, pages 217–233. Springer, 2010.

[15] Gael Glorian, Frédéric Boussemart, Jean-Marie Lagniez, Christophe Lecoutre, and Bertrand Mazure. Combining nogoods in restart-based search. In *CP*, pages 129–138. Springer, 2017.

[16] Carla P. Gomes, Bart Selman, Nuno Crato, and Henry A. Kautz. Heavy-tailed phenomena in satisfiability and constraint satisfaction problems. *JAR*, 24(1/2):67–100, 2000.

[17] Robert M. Haralick and Gordon L. Elliott. Increasing tree search efficiency for constraint satisfaction problems. *Artif. Intell.*, 14(3):263–313, 1980.

[18] George Katsirelos and Fahiem Bacchus. Unrestricted nogood recording in CSP search. In *CP*, pages 873–877. Springer, 2003.

[19] George Katsirelos and Fahiem Bacchus. Generalized nogoods in CSPs. In *AAAI*, pages 390–396, 2005.

[20] Christophe Lecoutre. *Constraint networks: techniques and algorithms*. ISTE/Wiley, 2009.

[21] Christophe Lecoutre and Olivier Roussel. Proceedings of the 2018 XCSP3 competition. *CoRR*, abs/1901.01830, 2019.

[22] Christophe Lecoutre, Lakhdar Sais, Sébastien Tabary, and Vincent Vidal. Nogood recording from restarts. In *IJCAI*, pages 131–136, 2007.

[23] Christophe Lecoutre, Lakhdar Sais, Sébastien Tabary, and Vincent Vidal. Recording and minimizing nogoods from restarts. *JSAT*, 1(3-4):147–167, 2007.

[24] Jimmy H. M. Lee, Christian Schulte, and Zichen Zhu. Increasing nogoods in restart-based search. In *AAAI*, pages 3426–3433. AAAI Press, 2016.

[25] Michael Luby, Alistair Sinclair, and David Zuckerman. Optimal speedup of las vegas algorithms. *Inf. Process. Lett.*, 47(4):173–180, 1993.

[26] Alan K. Mackworth. Consistency in networks of relations. *Artif. Intell.*, 8(1):99–118, 1977.

[27] Matthew W. Moskewicz, Conor F. Madigan, Ying Zhao, Lintao Zhang, and Sharad Malik. Chaff: Engineering an efficient SAT solver. In *DAC*, pages 530–535, 2001.

[28] Matthew W. Moskewicz, Conor F. Madigan, Ying Zhao, Lintao Zhang, and Sharad Malik. Chaff: Engineering an efficient SAT solver. In *DAC*, pages 530–535. ACM, 2001.

[29] Olga Ohrimenko, Peter J. Stuckey, and Michael Codish. Propagation via lazy clause generation. *Constraints*, 14(3):357–391, 2009.

[30] Knot Pipatsrisawat and Adnan Darwiche. A lightweight component caching scheme for satisfiability solvers. In *SAT*, pages 294–299. Springer, 2007.

[31] Patrick Prosser. Hybrid algorithms for the constraint satisfaction problem. *Computational Intelligence*, 9:268–299, 1993.

[32] Guillaume Richaud, Hadrien Cambazard, Barry O'Sullivan, and Narendra Jussien. Automata for nogood recording in constraint satisfaction problems. In *SAT/CP@CP*, page 113, 2006.

[33] Francesca Rossi, Peter van Beek, and Toby Walsh, editors. *Handbook of Constraint Programming*. Elsevier, 2006.

[34] Daniel Sabin and Eugene C. Freuder. Contradicting conventional wisdom in constraint satisfaction. In *PPCP*, pages 10–20. Springer, 1994.

[35] Daniel Sabin and Eugene C. Freuder. Contradicting conventional wisdom in constraint satisfaction. In *ECAI*, pages 125–129, 1994.

[36] Thomas Schiex and Gerard Verfaillie. Nogood recording for static and dynamic constraint satisfaction problems. *International Journal of Artificial Intelligence Tools*, 3(2):187–207, 1994.

[37] João P. Marques Silva and Karem A. Sakallah. GRASP: A search algorithm for propositional satisfiability. *IEEE Trans. Computers*, 48(5):506–521, 1999.

[38] Takehide Soh, Mutsunori Banbara, and Naoyuki Tamura. Proposal and evaluation of hybrid encoding of CSP to SAT integrating order and log encodings. *IJAIT*, 26(1):1–29, 2017.

[39] Toby Walsh. SAT v CSP. In *Proceedings of CP'00*, pages 441–456, 2000.

[40] Hugues Wattez, Christophe Lecoutre, Anastasia Paparrizou, and Sébastien Tabary. Refining constraint weighting. In *Proceedings of ICTAI'19*, pages 71–77, 2019.

[41] Lintao Zhang, Conor F. Madigan, Matthew W. Moskewicz, and Sharad Malik. Efficient conflict driven learning in boolean satisfiability solver. In *ICCAD*, pages 279–285. IEEE Computer Society, 2001.

[42] Neng-Fa Zhou and Håkan Kjellerstrand. The Picat-SAT compiler. In Marco Gavanelli and John H. Reppy, editors, *PADL*, pages 48–62. Springer, 2016.