LSIS Research Report (UMR CNRS 6168) Marseille France

Exchanging nogoods: an efficient cooperative search for solving constraint satisfaction problems
Research Report Number LSIS/2003/002, March 11th 2003

Cyril Terrioux

Laboratoire des Sciences de l'Information et des Systèmes
LSIS (UMR CNRS 6168)
Campus Scientifique de St Jérôme
avenue Escadrille Normandie Niemen
13397 MARSEILLE Cedex 20

Abstract
We propose a new cooperative concurrent search for solving the constraint satisfaction problem. Our approach consists in running independently many solvers (each one being associated with a process). These solvers exploit the algorithm Forward-Checking with Nogood Recording and they differ from each other in the heuristics they use for ordering variables and values. The cooperation is then based on exchanging nogoods (i.e. instantiations which cannot be extended to a solution). It is realized by two cooperation forms. On the one hand, we record every produced nogood in a shared memory. Thanks to these nogoods, each solver can then prune its own search tree. On the other hand a solver communicates directly a nogood to some other solvers by sending a message. We propose three different schemes for implementing the second cooperation form. Two of them reduce the communication cost by only sending to a solver the nogoods which are useful for it. Furthermore, we add to each solver a interpretation phase whose role is to limit the size of the search tree according to the received nogoods. Finally, we explain why a trade-off between cooperation and concurrency is required, before proposing such a trade-off.
From a practical point of view, the interest of our approach is shown experimentally on random instances and on real-world instances. First, we establish that exchanging nogoods appears to be an efficient cooperation form. In particular, on random instances, we obtain linear or superlinear speed-up for consistent problems, like for inconsistent ones, up to about ten solvers. Then, we compare our approach with some classical state-of-the-art enumerative algorithms. In a multiprocessor system, our approach with at least two or four solvers is faster than these classical algorithms. In a monoprocessor system, in most cases, it is equivalent to or better than Forward-Checking with Nogood Recording but it is often worse than the other classical algorithms. However, in a few cases, in particular for some real-world instances, it outperforms the classical algorithms.

Key words : Constraint satisfaction problem, concurrent search, cooperative search

---

1. Introduction

Constraint satisfaction problems (CSPs) define a powerful formalism for representing knowledge. The CSP formalism enables us to express different kinds of problems like configuration, design, scene labeling, planning, graph coloring, ... A CSP consists of a set of variables (each one having a finite domain) and a set of constraints. Each constraint forbids some combinations of values for a subset of variables. Solving a CSP requires to assign a value to each variable such that the assignment satisfies all constraints. Unfortunately, determining whether a CSP has a solution is a NP-complete task.
In the past years, a lot of methods have been developed for solving constraint satisfaction problems. The basic one is Chronological Backtracking (noted BT). BT is well known for its practical inefficiency. So, several techniques have been proposed to improve BT by reducing the size of the search tree. The first ones are look-ahead techniques which simplify the problem by filtering before or during the resolution like

Forward-Checking (noted FC [9]) or Maintaining Arc-Consistency (noted MAC [17]). Then, look-back techniques have been developed. They consist in analyzing the failure and then coming back as higher as possible in the search tree, like Backjumping [8], Graph-based Backjumping [7] or Conflict-Directed Backjumping [16]. Finally, learning techniques have been proposed like, for instance, Constraint Learning [7] or Nogood Recording [19]. They avoid some redundancies in the search tree by recording some informations. From these three kinds of improvements, hybrid methods have been produced like Forward-Checking with Nogood Recording (noted FC-NR [19]) or Forward-Checking with Conflict-directed Backjumping (noted FC-CBJ [16]). Jointly, many heuristics have been defined for the purpose of guiding the algorithms for the choice of variables and values to assign first. All these improvements aim to reduce the computation time.

Before solving a problem (or a collection of problems), the first task consists in choosing the method and the heuristics we are going to use. These choices are very important since they determine our ability to solve the problem. For instance, they are crucial for the computation time. The comparisons between methods (or heuristics) turn out to be helpful for choosing. However, most of them are empirical and concern only some benchmarks or some classes of random instances. With regard to theoretical comparisons [14, 4], they often require some limitations like the use of a particular variable ordering. Hence, in spite of some theoretical or empirical studies, choosing a method and/or an heuristic remains a difficult task.

The concurrency concept consists in running several independent solvers on the same problem, each one using a different method and/or a different heuristic. As solvers are independent, the search is finished as soon as a solver solves the problem (by finding a solution or by proving there is none). This concept can be used as a solution to avoid some bad choices. Indeed, such an approach allows us not to favor a particular method or a particular heuristic. Namely, the different heuristics/methods are run independently in the hope that one is well-adapted for the instance we want to solve. Furthermore, in the recent years, few new efficient methods or heuristics have been proposed. So the concurrency may appear as an interesting alternative approach.

In practice, concurrent searches are often efficient for solving consistent problems. On the other hand, these methods are seldom useful for inconsistent problems. Indeed, solving such problems requires to explore the whole search tree. Tested on the graph coloring problems [12], this approach obtains better results than a classical method with a single solver, but the gains seem limited. Hogg and Williams then recommend the use of cooperation to improve the efficiency of concurrent searches.

During the search, solvers explicit some informations which are the result of some amount of work. By exchanging and exploiting such informations, solvers may save some work, and so some computation time. For instance, exchanged informations can be exploited to guide solvers to a solution. Using cooperation raises some questions. First, we have to choose the kind of informations the solvers are going to exchange. Then, we have to decide when the solvers can (or must) send or receive informations and how they can take advantage of these informations. Answering these questions is all the more difficult because the addition of cooperation to an algorithm may modify its features like, for example, its soundness.

Experimental results on cryptarithmetic problems [5, 10] and on graph coloring problem [10, 11] show a significant gain in time with respect to an independent search. In both cases, the exchanged informations correspond to partial consistent instantiations. In [15], a cooperation based on exchanging nogoods (i.e. instantiations which can't be extended to a solution [19]) is proposed. Each solver runs the algorithm Forward Checking with Nogood Recording (noted FC-NR [19]) with a different heuristic for ordering values or variables. Each produced nogood is added as a new constraint to the initial problem. Then any solver can use these constraints in order to remove some values by filtering. This constraint addition corresponds to a cooperation form. Indeed, the nogoods produced by a solver can be exploited by other ones for pruning their own search tree. So, one can expect to find more quickly a solution. The realized implementation is turned to a monoprocessor system since it gathers all solvers in a single process which simulates the parallelism. Experimentations on random CSPs show that cooperative search is better than concurrent one. However, although this approach seems interesting and promising, the weak gain with report to a single solver gives a doubt about the efficiency of such a method, in particular if we want to use several processes.

In this article, we extend the Martinez and Verfaillie's works [15]. Like Martinez and Verfaillie, we propose a cooperative concurrent search whose all solvers run the same algorithm (namely FC-NR), use different heuristics for ordering variables and/or values and exchange nogoods. We also add the produced nogoods as new constraints and so these recorded nogoods are exploited during the filtering. However, we associate a process to each solver (i.e. our approach is turned to multiprocessor systems). By so doing, we assume that each process is able to access a shared memory which contains the instance we want to solve. Then we exploit a second cooperation form, namely each solver communicates the nogoods it finds to a part of the other solvers. Such communications between the solvers may raise some problems. For instance, the global communication cost may penalize the practical efficiency of our approach. So, in order to reduce the impact of such problems on the practical efficiency, we restrict the exchange of nogoods such that a nogood is only conveyed to a solver if this solver is liable to use it immediately on its receipt. Furthermore, in order to exploit the received nogoods, we add an interpretation phase to the FC-NR algorithm. With regard to the heuristics, we explain why a trade-off is required between the concurrency and the cooperation and we propose such a trade-off. From a practical point of view, we experiment our approach with random instances and real-world instances. On the one hand, we focus on the interest of our approach from a parallel viewpoint by assessing its speed-up and its efficiency. On the other hand, we compare our method with four classical enumerative algorithms.
With respect to our previous work [20], this article provides many extensions. First, we propose three schemes (instead of one) in order to realize the second cooperation form. Then, we present an improved version of the interpretation phase. Furthermore, we propose a trade-off between the concurrency and the cooperation and we explain why it is necessary. Finally, this paper significantly extends the experimental section by using more classes of random instances, by testing on real-world instances and by comparing our approach with classical state-of-the-art algorithm.

The plan is as follows. In section 2, we give some basic notions about CSPs, nogoods and the FC-NR algorithm. Then, in section 3, we present our approach. Section 4 provides some experimental results. Finally, after discussing about some possible extensions of this work in section 5, we conclude in section 6.

2. Definitions

2.1. Definitions about CSPs

A *constraint satisfaction problem* (CSP) is defined by a quadruplet (X, D, C, R). X is a set $\{x_1, \ldots, x_n\}$ of n variables. Each variable x_i takes its values in the finite domain d_{x_i} from D. Variables are subject to constraints from $C = \{c_1, \ldots, c_m\}$. Each constraint c_i is defined by a set $\{x_{i_1}, \ldots, x_{i_k}\}$ of variables. A relation r_{c_i} (from R) is associated with each constraint c_i such that r_{c_i} represents the set of allowed tuples over $d_{x_{i_1}} \times \cdots \times d_{x_{i_k}}$.
A CSP is called *binary* if each constraint involves at most two variables, *n-ary* otherwise. Let x_i and x_j be two variables, we note c_{ij} the binary constraint involving x_i and x_j. In the remaining of this paper, we consider only binary CSPs. However, our ideas can be extended to n-ary CSPs.
Given $Y \subseteq X$ such that $Y = \{y_1, \ldots, y_k\}$, an *instantiation* of variables from Y is a tuple $(v_1, \ldots, v_k)$ from $d_{y_1} \times \cdots \times d_{y_k}$. Given an instantiation $\mathcal{A}$ and $Y \subseteq X$, we note $X_{\mathcal{A}}$ the set of variables which are instantiated in $\mathcal{A}$, and $\mathcal{A}[Y]$ the instantiation $\mathcal{A}$ restricted to variables appearing in both $X_{\mathcal{A}}$ and Y. An instantiation $\mathcal{A}$ is called *consistent* if $\forall c \in C, c \subseteq X_{\mathcal{A}}, \mathcal{A}[c] \in r_c$, *inconsistent* otherwise. In other words, an instantiation $\mathcal{A}$ is consistent if it satisfies each constraint c such that every variable constrained by c belongs to $X_{\mathcal{A}}$. We use indifferently the term *assignment* instead of instantiation. We note the instantiation $(v_1, \ldots, v_k)$ in the more meaningful form $\{y_1 \leftarrow v_1, \ldots, y_k \leftarrow v_k\}$. A *solution* is a consistent instantiation of all variables. Given an instance $\mathcal{P} = (X, D, C, R)$, determine whether $\mathcal{P}$ has a solution is a NP-complete problem.

Example 1 *Let us consider the binary CSP* $\mathcal{P} = (X, D, C, R)$ *with:*

- $X = \{x_1, x_2, x_3, x_4, x_5, x_6, x_7\}$,
- $D = \{d_1, d_2, d_3, d_4, d_5, d_6, d_7\}$ *with* $d_1 = d_2 = d_3 = d_6 = d_7 = \{a, b, c\}$ *and* $d_4 = d_5 = \{a, b\}$,

- $C = \{c_{12}, c_{13}, c_{15}, c_{23}, c_{24}, c_{35}, c_{56}, c_{57}, c_{67}\}$,
- $R = \{r_{12}, r_{13}, r_{15}, r_{23}, r_{24}, r_{35}, r_{56}, r_{57}, r_{67}\}$ *where* r_{12} *corresponds to* $x_1 \neq x_2$, r_{13} $x_1 \neq x_3$, r_{15} $x_1 \leq x_5$ *(*$\leq$ *is the alphabetical order),* r_{23} $x_2 \neq x_3$, r_{24} $x_2 \leq x_4$, r_{35} $x_3 \leq x_5$, r_{56} $x_5 \neq x_6$, r_{57} $x_5 \neq x_7$ *and* r_{67} $x_6 \neq x_7$.

Given an instance $\mathcal{P}$ and an instantiation, we can define the notion of CSP induced by an instantiation:

Definition 1 (induced CSP)
Let $\mathcal{P} = (X, D, C, R)$ *be a CSP and* $\mathcal{A}_i$ *be an instantiation (*$\mathcal{A}_i = \{x_1 \leftarrow v_1, x_2 \leftarrow v_2, \ldots, x_i \leftarrow v_i\}$*).* $\mathcal{P}(\mathcal{A}_i) = (X, D(\mathcal{A}_i), C, R(\mathcal{A}_i))$ *is the* ***CSP induced*** *by* $\mathcal{A}_i$ *from* $\mathcal{P}$ *with a Forward Checking filter such that:*

- $\forall j, 1 \leq j \leq i, d_{x_j}(\mathcal{A}_i) = \{v_j\}$
- $\forall j, i < j \leq n, d_{x_j}(\mathcal{A}_i) = \{v_j \in d_{x_j} | \forall c_{kj} \in C, 1 \leq k \leq i, (v_k, v_j) \in r_{c_{kj}}\}$
- $\forall j, j', r_{c_{jj'}}(\mathcal{A}_i) = r_{c_{jj'}} \cap (d_{x_j}(\mathcal{A}_i) \times d_{x_{j'}}(\mathcal{A}_i))$.

$\mathcal{A}_i$ *is said FC-consistent if* $\forall j, d_{x_j}(\mathcal{A}_i) \neq \emptyset$.

According to this definition, $d_{x_j}(\mathcal{A}_i)$ represents the current domain of x_j (i.e. the domain whose some values have been deleted by the filterings inherent in the construction of $\mathcal{A}$) whereas d_{x_j} is the initial domain of x_j. Likewise, $r_{c_{jj'}}(\mathcal{A}_i)$ corresponds to the initial relation $r_{c_{jj'}}$ restricted to the current domains of x_j and $x_{j'}$.

2.2. Nogoods: definitions and properties

In this part, we give the main definitions and properties about nogoods. A nogood corresponds to an assignment which can't be extended to a solution. More formally:

Definition 2 (nogood [19])
Let $\mathcal{A}$ *be an instantiation and* J *a subset of constraints (*$J \subseteq C$*).* $(\mathcal{A}, J)$ *is a nogood if the CSP* (X, D, J, R) *doesn't have a solution which contains* $\mathcal{A}$*.* J *is called the nogood's justification (we note* X_J *the variables subject to constraints from* J*). The arity of the nogood* $(\mathcal{A}, J)$ *is the number of assigned variables in* $\mathcal{A}$*.*

For instance, every inconsistent assignment corresponds to a nogood. The converse doesn't hold.

Example 2 $(\{x_1 \leftarrow a, x_3 \leftarrow c\}, \{c_{15}, c_{35}\})$ *is a nogood although the affectation* $\{x_1 \leftarrow a, x_3 \leftarrow c\}$ *is consistent.* $(\{x_1 \leftarrow b\}, \{c_{12}, c_{13}, c_{15}, c_{23}, c_{24}, c_{35}\})$ *is a nogood too.*

In order to make the approach efficient in practice, we use the algorithm Nogood Recording based on Forward-Checking. So, the reasons of a failure may be many. Hence, we need the notion of "*value-killer*" (introduced in [19]) to compute justifications. A value-killer corresponds to a constraint which allows the deletion of a value by filtering. This constraint may be an initial constraint (condition (ii)) or a constraint which is added thanks to a nogood (condition (iii)). Note that we add the condition (iii) to the initial definition of Schiex and Verfaillie in order to exploit this notion in the cooperative concurrent search.

Definition 3 (value-killer)
Given a CSP $\mathcal{P}$*, an assignment* $\mathcal{A}_i$*, and the set* N *of produced nogoods, a constraint* c_{kj} $(j > i \geq k)$ *is a value-killer of value* v_j *from* d_{x_j} *for* $\mathcal{A}_i$ *if one of the following conditions holds:*

(i) c_{kj} *is a value-killer of* v_j *for* $\mathcal{A}_{i-1}$

(ii) $k = i$ *and* $(v_k, v_j) \notin r_{c_{kj}}(\mathcal{A}_i)$ *and* $v_j \in d_{x_j}(\mathcal{A}_{i-1})$

(iii) $\{x_k \leftarrow v_k, x_j \leftarrow v_j\} \in N$

The set of value-killers of a domain d_{x_j} *is defined as the set of constraints which are a value-killer for at least one value* v_j *of* d_{x_j}*.*

For a given value v, a value-killer of v is a constraint allowing to delete v by filtering, and so to explain the deletion of v. If a failure occurs because a domain d_{x_i} is wiped-out, the set of value-killers of d_{x_i} provides the reasons of the inconsistency (i.e. the justifications). The following theorem formalizes the creation of nogoods from dead-ends.

Theorem 1 *Let $\mathcal{A}$ be an assignment and x be an unassigned variable. Let K be the set of value-killers of d_x. If it doesn't remain any value in $d_x(\mathcal{A})$, then $(\mathcal{A}[X_K], K)$ is a nogood.*

The two next theorems make it possible to create new nogoods from existing nogoods. The first theorem builds a new nogood from a single existing nogood.

Theorem 2 *If $(\mathcal{A}, J)$ is a nogood, then $(\mathcal{A}[X_J], J)$ is a nogood.*

In other words, we only keep from the instantiation the variables which are involved in the failure. Thus, we produce a new nogood whose arity is limited to its strict minimum. Theorem 3 builds a new nogood from a set of nogoods:

Theorem 3 *Let $\mathcal{A}$ be an instantiation, x be an unassigned variable. Let K be the set of value-killers of d_x. Let $\mathcal{A}_j$ be the extension of $\mathcal{A}$ by assigning the value v_j to x ($\mathcal{A}_j = \mathcal{A} \cup \{x \leftarrow v_j\}$). If $(\mathcal{A}_1, J_1)$, ..., $(\mathcal{A}_{|d_x|}, J_{|d_x|})$ are nogoods, then $(\mathcal{A}, K \cup \bigcup_{j=1}^{|d_x|} J_j)$ is a nogood.*

Thanks to this theorem, when every extension of a node leads to a failure, we can build a new nogood.

A nogood can be used either to backjump or to add a new constraint or to tighten an existing constraint. If we produce the nogood $(\mathcal{A}, J)$, a constraint between the variables of $X_{\mathcal{A}}$ is added (or tightened if it already exists). This constraint forbids the tuple $\mathcal{A}$. Then, it can be exploited like any constraint of the initial problem, for instance for deleting values by filtering. The backjump provoked by a nogood is similar to one of the algorithm Conflict-directed BackJumping [16]. The next lemma characterizes this backjump phase:

Lemma 1 *If $(\mathcal{A}[X_J], J)$ is a nogood, it is correct to backjump to the deepest variable belonging to X_J.*

Of course, if there are many nogoods $(\mathcal{A}[X_J], J)$ involving the current instantiation, the solver must backjump to the deepest variable among all sets X_J. In both cases, it follows from the use of nogoods a pruning of the search tree. In particular, their use allows to avoid some redundancies in the search tree.

2.3. The Forward-Checking with Nogood Recording algorithm

We describe the algorithm FC-NR in figure 1. FC-NR visits the nodes of the search tree like Forward Checking. It tries to extend a FC-consistent instantiation $\mathcal{A}$. With this aim in view, FC-NR chooses the next variable x_i in V and tries to assign it a value in order to build an extension $\mathcal{A}'$ of $\mathcal{A}$. Enforce a Forward-Checking filter on the neighbouring unassigned variables of x_i determines whether$\mathcal{A}'$ is FC-consistent. The function $Forward - check$ realizes this work. It returns $\emptyset$ if the instantiation is FC-consistent, the set of constraints which are involved in the failure otherwise. Regarding the function $Unforward$, it cancels the last filtering. During the search, FC-NR takes advantage of inconsistencies and failures, which it finds, to create and record nogoods. These nogoods are then used as described above to prune the search tree. To do this work, sets of constraints J and J' are introduced. They represent the reasons of failures which are found when FC-NR tries to extend respectively $\mathcal{A}$ and $\mathcal{A}'$.
The main drawback of FC-NR is that the number of nogoods is potentially exponential (because it corresponds to the number of inconsistent assignments). So, we limit the number of nogoods, following the proposition of Schiex and Verfaillie [19], which consists in recording only nogoods whose arity is at most i. We fix i to 2.

3. A new cooperative concurrent search

3.1. Presentation

From the idea of Martinez and Verfaillie, we propose a new cooperative concurrent search where the cooperation is based on exchanging nogoods. Like Martinez and Verfaillie, we run independently p

FC-NR$(\mathcal{A}, V)$
In : a FC-consistent instantiation $\mathcal{A}$ and V the set of unassigned variables.
At beginning, $\mathcal{A} = \emptyset$ and $V = X$.
Out: $\emptyset$ if the problem is consistent ($\mathcal{A}$ is a solution),
the set of justifications of the failure otherwise.

1. If $V = \emptyset$ Then $\mathcal{A}$ is a solution, Stop
2. Else
3. Let $x_i \in V$, $d_i \leftarrow D_i(\mathcal{A})$, $J \leftarrow \emptyset$, $BackJump \leftarrow false$
4. While $d_i \neq \emptyset$ and $BackJump = false$
5. Choose $v \in d_i$
6. $d_i \leftarrow d_i - \{v\}$
7. $\mathcal{A}' \leftarrow \mathcal{A} \cup \{x_i \leftarrow v\}$
8. $K \leftarrow Forward\text{-}check(\mathcal{A}', x_i)$
9. If $K = \emptyset$
9. Then
10. $J_{sons} \leftarrow$ **FC-NR**$(\mathcal{A}', V - \{x_i\})$
11. $Unforward(x_i)$
12. If $x_i \in X_{J_{sons}}$ /* *If x_i is involved in the failure of $\mathcal{A}'$* */
12. Then $J \leftarrow J \cup J_{sons}$
13. Else
14. $J \leftarrow J_{sons}$
15. $BackJump \leftarrow true$
16. EndIf
17. Else
18. $Unforward(x_i)$
19. $J \leftarrow J \cup K$
20. Record $(\mathcal{A}'[X_K], K)$ /* *theorem 1* */
21. EndIf
22. EndWhile
23. If $BackJump = false$
23. Then
24. $J \leftarrow J \cup value\text{-}killer(x_i)$
25. Record $(\mathcal{A}[X_J], J)$ /* *theorems 2 and 3* */
26. EndIf
27. Return J
28. EndIf

Figure 1: The Forward Checking with Nogood Recording algorithm (FC-NR).

solvers. All solvers exploit the same algorithm, namely FC-NR, but they use different heuristics for ordering values and/or variables, what ensures that each solver visits a different search tree. The search is finished as soon as a solver finds a solution or proves there is none. Unlike Martinez and Verfaillie, we associate a process to each solver, and so our approach is turned to multiprocessor systems. The cooperation is based on exchanging nogoods. A solver can then prune a part of its search tree thanks to nogoods produced by other solvers. In our approach, we consider two cooperation forms for the exchange of nogoods. These two forms require the following assumptions:

- each process can access a shared memory which is big enough in order to contain the instance we want to solve, the set of recorded nogoods and the current instantiation of each solver,

- each process can communicate, by exchanging messages, with any other process.

We discuss in section 5 about a possible relaxation of the first hypothesis. The first form consists in adding each produced nogood to the initial problem by creating a new constraint or by tightening an

existing constraint. Then, as the problem (the recorded nogoods included) is stored in shared memory, any solver can classically use any nogood in order to backjump or to enforce an additional filtering. However, this first cooperation form (which is the only one used in [15]) is turned out to be insufficient for fully exploiting the produced nogoods. In fact, a solver may start exploring a subtree before receiving a nogood which allows it to prune this subtree. In such a case, by only exploiting the first cooperation form, the solver can't prune the subtree (because it doesn't check backward the constraints). Hence, depending on the number and the size of such subtrees, the loss of efficiency may become significantly important. So, in order to avoid such a problem, we propose an additional cooperation form. This second form is based on the explicit exchange of messages between the solvers. When a solver finds a nogood, it first adds it to the initial problem according to the first cooperation form. Then, it informs some other solvers of its discovery by sending a message which contains the instantiation of the nogood. Note that the communication of the justification isn't necessary because the nogood has just been added to the problem as a constraint. Finally, we add to FC-NR an interpretation phase which aims to limit the size of the search tree by pruning some inconsistent branches thanks to the received nogoods.
We can implement the second cooperation form in several ways. We propose three schemes. In the first scheme (noted $\mathcal{S}_0$ and presented in figure 2), each solver communicates the nogoods it produces to all the other solvers. In this basic scheme, every time a nogood is found, the solver sends $p-1$ messages to its partners. Although the number of nogoods is bounded by $O(n^2d^2)$ (since their arity is limited to two), the global cost of communications may become very important, prohibitive even. Moreover, a given nogood isn't necessarily useful for all solvers. So we define a second scheme (noted $\mathcal{S}_{light}$ and depicted in figure 3) in which we restrict the exchanges of nogoods. In fact, a nogood is only conveyed to solvers which are liable to use it as soon as it is received. This notion of usefulness is described in the next subsection. A potential drawback of this scheme is the time spent for determining which solvers are the recipients of a given nogood and for sending the messages since, during this time, the solver don't look for a solution. Hence, in a third scheme (noted $\mathcal{S}_{gest}$ and shown in figure 4), we add to the p solvers a process called the "*nogood manager*", whose role is to relieve the solvers from the communication of nogoods and from their addition to the constraint set. So, when a solver produces a nogood, it informs the manager which adds it to the constraint set and then communicates at once this new information to a restricted part of the solvers according the notion of usefulness. By so doing, the solver only sends one message per nogood and then gets back more quickly to the resolution of the problem.

First, we describe how we restrict the exchanges and we present the nogood manager. Then we assess the number of exchanged messages in each scheme. Afterwards, we present the interpretation phase. Finally, we explain why a trade-off is required between the concurrency and the cooperation and we propose such a trade-off.

3.2. Restriction of exchanges and nogood manager

The manager's task is to update the nogood set and to communicate new nogoods to solvers. Update the nogood set consists in adding constraints to initial problem or in tightening the existing constraints. To a unary (respectively binary) nogood corresponds a unary (resp. binary) constraint. The manager adds to the constraint set all the nogoods it receives. In schemes S_0 and S_{light}, this work is directly achieved by the solver which produces the nogood. With regard to the communications, we restrict the exchange of nogoods according the same way in $\mathcal{S}_{light}$ and in $\mathcal{S}_{gest}$. The main difference between these two schemes consists in the nature of the process which applies these restrictions. In $\mathcal{S}_{light}$, it's the solver which finds the nogood whereas in $\mathcal{S}_{gest}$, this task is done by the manager.
By restricting the exchanges of nogoods in schemes $\mathcal{S}_{light}$ and $\mathcal{S}_{gest}$, we aim not only to limit the number of communications, but also not to interrupt needlessly the solvers in their search for a solution. So, we must only inform the solvers for which the produced nogood may be useful.

Definition 4 (usefulness of a nogood) *A nogood is said **useful** for a solver if it allows this solver to reduce the size of its search tree.*

Among the useful nogoods for a solver, we can distinguish two kinds of nogoods:

- the nogoods which are useful on their receipt: they are nogoods whose receipt allows a solver to reduce the size of its search tree (and even to put immediately an end to the search),

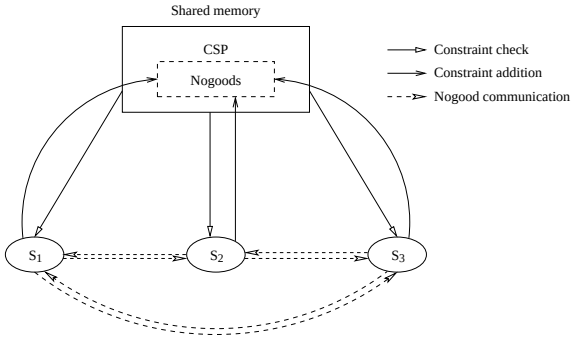

Figure 2: The cooperative scheme S_0 for three solvers. Solid arrows represent accesses to the shared memory, dashed ones communications between processes.

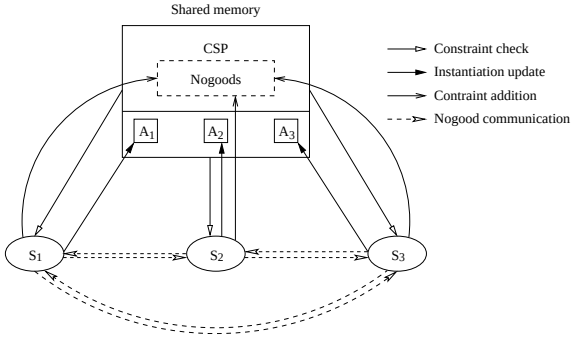

Figure 3: The cooperative scheme $\mathcal{S}_{light}$ for three solvers. Solid arrows represent accesses to the shared memory, dashed ones communications between processes.

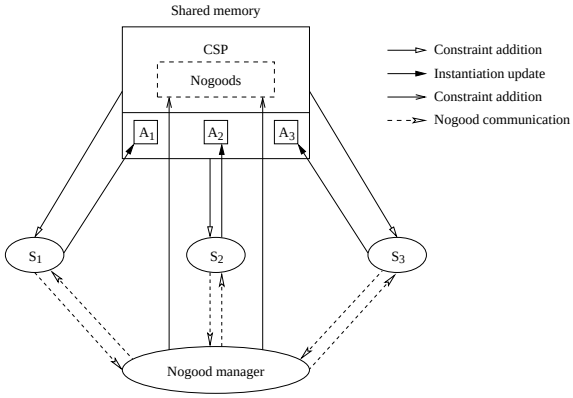

Figure 4: The cooperative scheme $\mathcal{S}_{gest}$ for three solvers. Solid arrows represent accesses to the shared memory, dashed ones communications between processes.

- the nogoods which are useful later thanks to the filtering.

For the last ones, note that they are exploited via the constraints added or tightened (i.e. thanks to the shared memory). So we only need to communicate the nogoods which are useful as soon as they are received. Hence, in $\mathcal{S}_{light}$, the solver which finds a nogood (respectively the manager in $\mathcal{S}_{gest}$) has to establish whether a nogood may be useful for a solver on its receipt. With this aim in view, the only knowledge it has about the possible recipients is their current instantiation (since this information is stored in the shared memory). Note that these accesses don't appear in figures 3 and 4 in order to keep them readable. The following lemma characterizes the usefulness of these nogoods depending on their arity and the current instantiation of the considered solver.

Lemma 2 *Let S be a solver and $\mathcal{A}_S$ be its current instantiation.*

(a) a unary nogood is always useful for S on its receipt,

(b) a binary nogood $(\{x_i \leftarrow a, x_j \leftarrow b\}, J)$ is useful for S on its receipt if $\mathcal{A}_S[x_i] = a$ and $\mathcal{A}_S[x_j] = b$,

(c) a binary nogood $(\{x_i \leftarrow a, x_j \leftarrow b\}, J)$ is useful for S on its receipt if $\mathcal{A}_S[x_i] = a$ and $b \in D_j(\mathcal{A}_S)$ or if $\mathcal{A}_S[x_j] = b$ and $a \in D_i(\mathcal{A}_S)$.

Proof:

(a) A unary nogood corresponds to a unary constraint. So it allows the solver to remove permanently a value from the corresponding domain. Consequently, such nogoods are always useful.

(b) Let $(\{x_i \leftarrow a, x_j \leftarrow b\}, J)$ be a binary nogood. Thanks to this nogood, we can ensure that any branch which contains $x_i \leftarrow a$ and $x_j \leftarrow b$ can't lead to a solution. So, we don't need to explore such branches. In particular, it's the case if the current instantiation of S contains both x_i assigned with a and x_j assigned with b. In such a situation, the nogood is useful for S.

(c) Let $(\{x_i \leftarrow a, x_j \leftarrow b\}, J)$ be a binary nogood. Like in the previous case, any branch which contains $x_i \leftarrow a$ and $x_j \leftarrow b$ can't lead to a solution. Assume that x_i is assigned with a and x_j i not assigned yet. Then, in order to avoid an inconsistent branch, the solver can delete by filtering the value b from the domain d_{x_j}. And so, the nogood is the useful for S. □

Thanks to this lemma, we now provide the following policy according to which a solver (or the manager) sends a given nogood to a part of the other solvers:

(a) every unary nogood is sent to all solvers (except the solver which finds it),

(b) every binary nogood $(\{x_i \leftarrow a, x_j \leftarrow b\}, J)$ is sent to each solver (except the solver which finds it) whose instantiation contains $x_i \leftarrow a$ or $x_j \leftarrow b$.

In other words, the nogoods which are communicated to a solver are ones which may be useful as soon as they are received. However, we can't guarantee that all the received nogoods will be used in practice. For instance, in case (b), the solver may have backtracked between the sending of the nogood by the manager and its receipt by the solver.

Example 3 *Let us consider the instance presented in example 1. We run the cooperative concurrent search with three solvers S_1, S_2 and S_3 according to the scheme S_{light}. S_1 exploits $(x_1, x_2, x_3, x_4, x_5, x_6, x_7)$ as an ordering over the variables, S_2 $(x_2, x_1, x_3, x_4, x_5, x_6, x_7)$ and S_3 $(x_4, x_1, x_3, x_5, x_2, x_6, x_7)$. For values, solvers use the alphabetical order. Furthermore, we assume that the current instantiation of S_2 (respectively S_3) is $\{x_2 \leftarrow a, x_1 \leftarrow b\}$ (resp. $\{x_4 \leftarrow a, x_1 \leftarrow a\}$). If S_1 produces the nogood $(\{x_1 \leftarrow a, x_3 \leftarrow c\}, \{c_{15}, c_{35}\})$, it only sends it to S_3 since only S_3 is liable to exploit it on its receipt (by removing c from x_3's domain). When S_2 will study the affectation $\{x_2 \leftarrow b, x_1 \leftarrow a\}$, S_2 will need this nogood. It will then exploit it via the shared memory. Indeed, when S_1 produced the nogood, it has added it to the constraint set by deleting (a, c) from the relation $r_{c_{13}}$. On the other hand, if S_1 finds the nogood $(\{x_1 \leftarrow b\}, \{c_{12}, c_{13}, c_{15}, c_{23}, c_{24}, c_{35}\})$, it will communicates it to S_2 and S_3. By so doing, as soon as this information is received, S_2 and S_3 are able to delete b from x_1's domain. In both cases, only the instantiation is conveyed, namely $\{x_1 \leftarrow a, x_3 \leftarrow c\}$ or $\{x_1 \leftarrow b\}$.*

3.3. Assessment of the number of messages

In this part, we assess, for each scheme, the number of explicit messages exchanged between the solvers. Let N be the total number of nogoods which are exchanged by all solvers during the whole search. We count U unary nogoods and B binary ones. Note that, among these N nogoods, doubles may exist as two solvers can find independently a same nogood.

In scheme S_{gest}, the produced nogoods are first sent to the manager by the solvers. During the whole search, solvers convey to the manager U messages for unary nogoods and B for binary ones. Then, the manager sends only u unary nogoods to $p-1$ solvers. These u nogoods correspond to U nogoods minus the doubles. Likewise, for binary nogoods, doubles aren't communicated. Furthermore, for the remaining binary nogoods, the manager restricts the number of recipients. Let b be the number of messages sent by manager for binary nogoods. In this scheme, we exchange $U + u(p-1)$ messages for unary nogoods and $B + b$ messages for binary ones.

In scheme $\mathcal{S}_{light}$, the solver which produces a nogood establishes its usefulness for a solver before sending it. Moreover doubles aren't communicated again. Therefore, only $u(p-1)$ messages are sent for unary nogoods and b for binary ones. On the other hand, in scheme S_0, each solver communicates directly the nogoods it finds to $p-1$ other solvers. So, $U(p-1)$ messages are sent for unary nogoods and $B(p-1)$ for binary ones.

In the worst case, scheme S_{gest} produces up to N additional messages in comparison with schemes S_0 and S_{light}. But, in general, we have observed that u and b are little enough so that S_{gest} produces fewer messages than S_0. In particular, the number of messages for binary nogoods is significantly restricted. Finally, note that this comparison assumes a ideal framework where the solvers visit the same search tree and produce the same nogoods independently of the used scheme. However, in practice, the search tree and so the produced nogoods may differ according to the used scheme.

3.4. Phase of interpretation

Before describing the interpretation phase we introduce the following notation: given an instantiation $\mathcal{A}$, we note $\mathcal{A}_{x_k}$ the restriction of $\mathcal{A}$ to variables assigned before x_k, including x_k.

Each solver exploits an interpretation phase independently of the used scheme. This phase is applied whenever a nogood is received. For information, solvers check whether a message is received after developing a node and before filtering. In the phase of interpretation, solvers analyze received nogoods in order to limit the size of their search tree by stopping branches which can't lead to solution or by enforcing additional filtering.

For the unary nogoods, this phase leads to the permanent deletion of a value and to an eventual backjump. Method 1 describes this phase for such nogoods:

Method 1 (phase of interpretation for unary nogoods)
Let $\mathcal{A}$ be the current instantiation. Let $(\{x_i \leftarrow a\}, J)$ be the received nogood. We denote K the set of value-killers of d_{x_i}.
We first delete the value a from d_{x_i}. Furthermore:

(a) If x_i isn't assigned and $d_{x_i}(\mathcal{A})$ is empty:

- *(i) we record the nogood $(\mathcal{A}[X_K], K)$.*
- *(ii) If recording this nogood wipes out a domain, then we backjump to the highest variable between the deepest variable in X_K and the deepest one in $X_{K'}$ (with K' the set of value-killers of the wiped-out domain).*
  Otherwise, we backjump to the deepest variable in X_K.

(b) If x_i is assigned with the value a:
we backjump to x_i. Let $\mathcal{A}' \cup \{x_i \leftarrow a\}$ be the obtained instantiation.
If $d_{x_i}(\mathcal{A}')$ is empty:

- *(i) we record the nogood $(\mathcal{A}'[X_K], K)$.*
- *(ii) If recording this nogood wipes out a domain, then we backjump to the highest variable between the deepest variable in X_K and the deepest one in $X_{K'}$ (with K' the set of value-killers of the*

wiped-out domain).
Otherwise, we backjump to the deepest variable in X_K.

Note that we remove permanently a from the domain d_{x_i}, which implies that, for any instantiation $\mathcal{A}$, the value a doesn't belong to $d_{x_i}(\mathcal{A})$. The backjump phases in the cases (a)(ii) et (b)(ii) allow to keep on exploring the search tree with a FC-consistent instantiation (unlike the phase of interpretation in [20]). In other words, these backjump phases guarantee that there is no empty current domain, before extending the current instantiation. Finally, if x_i is assigned with a value which differs from a, there is nothing else to do after having deleted the value a from the domain d_{x_i}. In fact, on the one hand, the domain $d_{x_i}(\mathcal{A})$ can't be wiped out (because x_i is assigned with a value different from a). On the other hand, receiving the nogood $(\{x_i \leftarrow a\}, J)$ doesn't give to the solver any additional information about the success or the failure of the extensions of the instantiation $\mathcal{A}$.

Theorem 4 *The interpretation phase for the unary nogoods is correct.*

Proof:
As $(\{x_i \leftarrow a\}, J)$ is a nogood, the instantiation $\{x_i \leftarrow a\}$ can't be extended to a solution. Therefore, we can remove a from the domain D_i without modifying the problem's consistency. Now, we study the correctness of the two cases:

(a) If x_i isn't assigned:
The deletion of a may wipe out the domain $d_{x_i}(\mathcal{A})$. If $d_{x_i}(\mathcal{A})$ is empty, then the instantiation $\mathcal{A}$ is FC-inconsistent. According to theorems 1 and 2, $(\mathcal{A}[X_K], K)$ is a nogood. Furthermore, thanks to lemma 1, it is correct to backjump to the deepest variable in X_K. If recording the nogood $(\mathcal{A}[X_K], K)$ wipes out the domain of a variable x_l, thanks to lemma 1, backjumping to deepest variable in X'_K is correct too. So, in such a case, we can backjump to the highest one.

(b) If x_i is assigned with the value a:
As $(\{x_i \leftarrow a\}, J)$ is a nogood, every instantiation which contains x_i assigned with a can't lead to a solution. Therefore, it's useless to develop such instantiations and so backjumping to x_i is correct.
Let $\mathcal{A}' \cup \{x_i \leftarrow a\}$ be the obtained instantiation. After removing a from d_{x_i}, $d_{x_i}(\mathcal{A}')$ may be empty. In such a case, according to theorems 1 and 2, we can record the nogood $(\mathcal{A}'[X_K], K)$. Finally, the correctness of the backjump phase (b)(ii) can be established like in the first case. □

Example 4
We go on with the example 3. When S_3 receives the nogood $(\{x_1 \leftarrow b\}, \{c_{12}, c_{13}, c_{15}, c_{23}, c_{24}, c_{35}\})$, it permanently removes b from d_{x_1}. For S_2, receiving this nogood leads to the permanent deletion of b from d_{x_1} and to the end of the current search. S_2 then keeps on its exploration with the instantiation $\{x_2 \leftarrow a, x_1 \leftarrow c\}$.

For binary nogoods, the phase corresponds to enforce an additional filtering and to a possible backjump as described in method 2:

Method 2 (phase of interpretation for binary nogoods)
Let $\mathcal{A}$ be the current instantiation and $(\{x_i \leftarrow a, x_j \leftarrow b\}, J)$ be the received nogood.

(a) If $\{x_i \leftarrow a\} \subseteq \mathcal{A}$ (resp. $\{x_j \leftarrow b\} \subseteq \mathcal{A}$) and $x_j \notin X_\mathcal{A}$ (resp. $x_i \notin X_\mathcal{A}$):
we delete by filtering b (resp. a) from $d_{x_j}(\mathcal{A}_{x_i})$ (resp. $d_{x_i}(\mathcal{A}_{x_j})$).
If $d_{x_j}(A)$ (resp. $d_{x_i}(A)$) is empty:

(i) we record the nogood $(\mathcal{A}[X_K], K)$ with K the set of value-killers of d_{x_j} (resp. d_{x_i}).

(ii) If recording this nogood wipes out a domain, then we backjump to the highest variable between the deepest variable in X_K and the deepest one in $X_{K'}$ (with K' the set of value-killers of the wiped-out domain).
Otherwise, we backjump to the deepest variable in X_K.

(b) If $\{x_i \leftarrow a, x_j \leftarrow b\} \subseteq \mathcal{A}$*:*
we backjump to the deepest variable among x_i *and* x_j*.*
If x_j *(resp.* x_i*) is this variable, we note* $\mathcal{A}' \cup \{x_j \leftarrow b\}$ *(resp.* $\mathcal{A}' \cup \{x_i \leftarrow a\}$*) the obtained instantiation.*
We delete by filtering b *(resp.* a*) from* $d_{x_j}(\mathcal{A}_{x_i})$ *(resp.* $d_{x_i}(\mathcal{A}_{x_j})$*).*
If $d_{x_j}(\mathcal{A}')$ *(resp.* $d_{x_i}(\mathcal{A}')$*) is empty:*

(i) we record the nogood $(\mathcal{A}'[X_K], K)$ *with* K *the set of value-killers of* d_{x_j} *(resp.* d_{x_i}*).*

(ii) If recording this nogood wipes out a domain, then we backjump to the highest variable between the deepest variable in X_K *and the deepest one in* $X_{K'}$ *(with* K' *the set of value-killers of the wiped-out domain).*
Otherwise, we backjump to the deepest variable in X_K*.*

Like previously, the backjump phases allow to keep on exploring the search tree with a FC-consistent instantiation and, by so doing, to avoid the drawback of the phase presented in [20]. Remark that, here, the deletions aren't permanent unlike the phase of interpretation for unary nogoods. Indeed, they are similar to ones obtained by filtering.

Theorem 5 *The interpretation phase for the binary nogoods is correct.*

Proof:

(a) Assume that x_j is the deepest variable among x_i and x_j (the proof is similar if x_i is the deepest one). The nogood $(\{x_i \leftarrow a, x_j \leftarrow b\}, J)$ prevents x_j from taking the value b as long as x_i is assigned with the value a. Therefore, we can remove b from $d_{x_j}(\mathcal{A}_{x_i})$ by filtering.
If $d_{x_j}(\mathcal{A})$ is wiped out, we reason like in the proof of theorem 4 in order to establish the correction of recording the nogood and of the backjumping phase.

(b) As $(\{x_i \leftarrow a, x_j \leftarrow b\}, J)$ is a nogood, every instantiation which contains $\{x_i \leftarrow a, x_j \leftarrow b\}$ can't lead to a solution. So, backjump to the deepest variable among x_i and x_j is correct.
Assume that x_j is the deepest variable among x_i and x_j (the proof is similar if x_i is the deepest one). $\mathcal{A}' \cup \{x_j \leftarrow b\}$ is the instantiation obtained after the backjump. We demonstrate like for the case (a) that the filtering, the recording and the backjumping phase are correct. □

Example 5 *We go on with the example 3. If* S_3 *receives the nogood* $(\{x_1 \leftarrow a, x_3 \leftarrow c\}, \{c_{15}, c_{35}\})$*, it removes by filtering* c *from* x_3*'s domain. This deletion is cancelled as soon as* S_3 *develops the instantiation* $\{x_4 \leftarrow a, x_1 \leftarrow b\}$*.*

3.5. Trade-off between concurrency and cooperation

On the one hand, the concurrency concept is based on running different solvers on the same problem with the aim that one of them is well-adapted to the instance we want to solve. So if we want to improve the efficiency of the concurrency, we have to increase the diversity of the solvers. In our approach, the solvers exploit the same algorithm, but they differ in using different heuristics for ordering values and/or variables. Then, in order to improve the efficiency of our approach, we need to exploit heuristics as different as possible.

On the other hand, making the cooperation efficient requires that exchanged informations can be exploited by a major part of the solvers. In our approach, the cooperation is based on exchanging nogoods, that is to say informations which depend directly on the explored search tree. So if we want a nogood to be useful for some other solvers, we must impose that the different search trees explored by the solvers are close to each other. In other words, as the solvers only differ in using different heuristics for ordering values and/or variables, these heuristics must be relatively close in terms of developed search tree.

To sum up, the cooperation requires some proximity of heuristics while the concurrency needs some diversity. As these requirements are opposed, a trade-off is needed in order to make our approach efficient in practice. The trade-off we propose consists in running solvers by pair. In a same pair, the solvers exploit two similar heuristics for ordering variables. These two heuristics are close enough for starting with the same variable. With regards to value orderings, the two solvers use different heuristics. Of course, solvers from different pairs exploit different heuristics for ordering variables (i.e. they start with a different variable). By so doing, as the two solvers of a same pair start their exploration with the same variable, we ensure a cooperation at less between these two solvers.

4. Experimental results

In this section, we experiment our concurrent cooperative approach in order to determine whether it corresponds to an efficient cooperation form and if it can be an interesting alternative to classical enumerative algorithms. We first study the behaviour of our concurrent cooperative method on classical random instances from a parallel point of view. Then we provide experimental comparisons between our cooperative scheme and some state-of-the-art sequential algorithms on random instance. Finally, we experiment our approach on some real-world instances.

4.1. Implementation

4.1.1. Algorithms

We have implemented many methods:

- our concurrent cooperative method for the schemes S_0, S_{light} and S_{gest},
- a concurrent version without cooperation,
- the classical algorithms FC [9], FC-CBJ [16], MAC [17], and FC-NR [19].

The implementation of the cooperative concurrent search is based on *pthreads*. Pthreads (and most generally threads) have the advantage of using shared memory. A pthread is associated with each solver. Likewise, a pthread is associated to the nogood manager in S_{gest}. These pthreads are run in parallel by the operating system until the problem is solved (by finding a solution or by proving there is none). As soon as a solver solves the problem, all pthreads are stopped. As, in practice, our experimentations are done on a monoprocessor computer, we consider in fact pseudo-parallelism. The execution of pthreads by the operating system then exploits the notion of process scheduling in multitasking operating systems. The used scheduling policy corresponds to a variant of the round-robin policy. Communications between solvers are realized thanks to messages which are sent from a pthread to another one.
The concurrent method without cooperation consists in running the solvers with exactly the same heuristics as the cooperative method. In other words, compared with the cooperative search, it doesn't exploit a shared memory and it doesn't exchange any message between the solvers. So, each solver has its own copy of the problem and it only adds the nogoods it finds to its local copy. The search is finished as soon as a solver solves the problem. Like previously, the implementation is also based on pthread. It is similar to one of the cooperative method, except that each pthread uses its own copy of the problem we want to solve.
With regards to the classical algorithms, we use the AC-2001 algorithm [2] to enforce and maintain the arc-consistency in MAC. For the FC-NR algorithm, we limit the arity of recorded nogoods to 2.

4.1.2. Heuristics

About the heuristics for ordering variables, FC, FC-CBJ and MAC use the dom/deg heuristic [1], for which the next variable to assign is one which minimizes the ratio $\frac{|d_{x_i}|}{|\Gamma_{x_i}|}$ (where d_{x_i} is the current domain of x_i and Γ_{x_i} is the set of variables which are connected to x_i by a binary constraint). This heuristic is generally considered as better than other classical heuristics. That's why we choose it. However for FC-NR, we prefer exploit the dom/st heuristic for which the next variable to assign is one which minimizes the ratio $\frac{|d_{x_i}|}{S_{x_i}}$ (with S_{x_i} the sum of tightness of constraints involving x_i). Thanks this heuristic, FC-NR obtains better results with respect to dom/deg.
For the cooperative method, in order to guarantee distinct search trees, each solver orders variables and/or values with different heuristics. If these heuristics must be different, they must also be close to each other in order to improve the cooperation. Furthermore, it appears important that they have a similar efficiency in order to avoid the problem being always solved by the same solver without any benefit from cooperation. As there exist few efficient heuristics for choosing variables, we produce several different orders from the heuristics dom/deg and dom/st by choosing differently the first variable and then applying either dom/deg or dom/st.

Finally, note that, in our implementation, only the size of domains varies for each instantiation. The degree $|\Gamma_{x_i}|$ is only updated when a new constraint is added thanks to a nogood. Likewise, the tightness of a constraint is modified when the constraint is tightened.
As regards the choice of next value to assign, we consider values in a static order for the classical algorithms. In practice, we use an arbitrary order σ, which the same for each algorithm. For the concurrent method with/without cooperation, according to the trade-off we propose, we run solvers by pair. In a same pair, the solvers start with the same variable and then one uses dom/deg while the other one exploits dom/st. For value ordering, one solver uses the order σ while the other one exploits the reverse order.

4.2. Experimental protocol

We experiment our method on two kinds of benchmarks: random instances and real-world instances. In this subsection, we describe, for each kind of problems, the experimental protocol we use. Note that, in both cases, the constraint checks take into account:

- the initial constraints and the added constraints for the cooperative method, for the concurrent one and for FC-NR,
- the initial constraints for other algorithms.

4.2.1. Experimental protocol for random instances

Experimentations on random instances are realized on a linux-based PC with an AMD Athlon XP 1800+ processor and 512 Mb memory. These instances are produced by random generator written by D. Frost, C. Bessière, R. Dechter and J.-C. Régin. This generator [1] takes 4 parameters n, d, m and t. It builds a CSP of class (n, d, m, t) with N variables which have domains of size d and m binary constraints $(0 \leq m \leq \frac{n(n-1)}{2})$ in which t tuples are forbidden $(0 \leq t \leq d^2)$. Experimental results we give afterwards concern classes which are near to the satisfiability threshold. Every problem we consider has a connected constraint graph.
The concurrent method and the cooperative method aren't deterministic because of the concurrency and the exchange of information. Hence, we solve each instance fifteen times in order to reduce the impact of non-determinism on results. Therefore, the results for a given instance corresponds to the average of results of fifteen resolutions. For a given resolution, the results we consider are ones of the solver which solves the problem first. By so doing, we assume that we have one processor per solver, even if, in practice, these experimentations are realized on a computer with a single processor. Of course, for classical algorithm, each instance is solved once. The results we provide are the average of results obtained by solving 100 instances per class.

4.2.2. Experimental protocol for real-world instances

Experimentations on real-world instances are realized on a linux-based PC with an Intel Pentium III 550 MHz and 256 Mb memory. The instances we consider are some real-world instances of the CELAR from the FullRLFAP archive[2]. These problems correspond to radio link frequency assignment problems. For our experimentation, we only look for a solution (i.e. we do not search an optimal solution). From this archive, we only keep the instances which FC-NR solves in at least 100 milliseconds. These instances have between 400 and 916 variables with different sizes of domains (see [3] for more details).
The results we give are the results of a unique run. For the cooperative method, like previously, we assume that we have one processor per solver, even if these experimentations are realized on a computer with a single processor. In other words, the presented results are ones of the solver which solves the problem first. We set a time limit for determining whether a problem is consistent or not. Beyond 15 minutes, the search is stopped.

4.3. Efficiency of the cooperative concurrent method

4.3.1. Speed-up and efficiency

In this part, we compute and present the speed-up and the efficiency obtained by our cooperative concurrent method, what allows us to assess the interest of our approach from a parallel viewpoint. We first

[1] downloadable at http://www.lirmm.fr/~bessiere/generator.html
[2] we thank the Centre d'Electronique de l'Armement (France).

recall these basic notions.

Definition 5 (speed-up) *Let T_1 (respectively T_p) be the run-time required by a method with a single solver (resp. p solvers) for solving a set of problems. The* ***speed-up*** *is defined by the ratio $\frac{T_1}{T_p}$. A speed-up is called* ***linear*** *with report to the number p of solvers if it equals to p,* ***superlinear*** *if it is greater than p,* ***sublinear*** *otherwise.*

A speed-up corresponds normally to a value between 1 and p. This result assumes that T_1 is the run-time of the best sequential version of the considered algorithm. For solving NP-complete problems like CSP, it's really difficult to determine the best version of an algorithm because this notion of "best version" generally depends on the instance we want to solve. So, in practice, we can often obtain superlinear speed-up.

Definition 6 (efficiency) *Let T_1 (respectively T_p) be the run-time required by a method with a single solver (resp. p solvers) for solving a set of problems. The* ***efficiency*** *is defined by the ratio $\frac{T_1}{p.T_p}$.*

A way of assessing the interest of a method from a parallel viewpoint consists in studying its efficiency. The greater the efficiency is, the more efficient the method is. Note that, due to superlinear speed-up, the efficiency can also become greater than 1. In practice, we consider that an efficiency greater or equal to 0.95 characterizes an efficient method. Finally, remark that the efficiency depends on the number of used solvers and that it generally decreases when the number of solvers increases.
When we define the notions of speed-up and efficiency, we assume that we use a processor per solver, even if the experimentations are done on a monoprocessor computer. More classically, the efficiency can be defined as the ratio $\frac{T_1}{q.T'_q}$ where T_1 (respectively T'_q) is the run-time required by a method which exploits a single processor (resp. q processors) for solving a set of problems. In our case (i.e. $q = 1$), the run-time T'_q corresponds to the sum of the run-time of each solver we run in concurrency. As all solvers finish as soon as a solver solves the problem, T'_q is almost equal to $p.T_p$ (with T_p the run-time of the solver which solves the problem first). So exploiting the definition 6 or the classical one doesn't change anything to obtained results. In practice, according to the definition we use, the obtained difference is generally about a few thousandths and at most two hundredths. Likewise, in scheme S_{gest}, we don't take into account the manager's run-time because it appears to be insignificant with respect to the solvers' one.

Class (n, d, m, t)	Scheme	p				
		2	4	6	8	10
(50,15,184,112)	S_0	1.207	1.130	1.150	1.105	1.048
	S_{light}	1.213	1.131	1.146	1.090	1.053
	S_{gest}	1.235	1.159	1.167	1.104	1.064
(50,15,245,93)	S_0	1.101	1.082	1.004	0.931	0.846
	S_{light}	1.099	1.085	1.008	0.934	0.851
	S_{gest}	1.133	1.094	1.018	0.942	0.849
(50,25,123,439)	S_0	1.112	1.073	0.939	1.010	0.932
	S_{light}	1.121	1.093	0.966	1.041	0.973
	S_{gest}	1.219	1.160	1.015	1.046	1.022
(50,25,150,397)	S_0	1.085	1.404	1.373	1.330	1.233
	S_{light}	1.101	1.408	1.354	1.324	1.240
	S_{gest}	1.030	1.381	1.353	1.321	1.227
(75,10,277,43)	S_0	1.339	1.319	1.285	1.218	1.183
	S_{light}	1.334	1.325	1.324	1.192	1.174
	S_{gest}	1.325	1.318	1.298	1.186	1.161

Table 1: Efficiency obtained by the cooperative concurrent method with the schemes S_0, S_{light} and S_{gest} for consistent and inconsistent problems.

Table 1 provides the efficiency obtained by the cooperative concurrent method for each proposed scheme on five classes of random CSPs (for consistent and inconsistent problems). Table 2 (respectively table 3) shows such results for consistent problems (resp. inconsistent problems). First, we note that the three schemes present similar results. Then, from table 1, we observe that the cooperative method obtains linear or superlinear speed-up for three classes out of five up to ten solvers. On the two other classes (namely classes (50,15,245,93) and (50,25,123,439)), the speed-up is either linear or superlinear, or sublinear depending on the number of used solvers. If we distinguish the problems according to their consistency, we observe that our method is generally more efficient on consistent problems. Indeed, on the one hand, the efficiency on such problems turns to be greater than one on inconsistent problems (except for class (50,25,150,397) with two solvers in S_{gest}). On the other hand, on consistent problems, the speed-up is linear or superlinear in most cases while on inconsistent problems, the results are mixed. For classes (50,15,245,93) and (50,25,123,439), the speed-up becomes sublinear above four or six solvers. For the other classes, it remains linear or superlinear up to ten solvers. Finally, we remark a decrease of efficiency when the number of solvers increases for both consistent and inconsistent problems. Such a decrease is a classical phenomenon for parallel searches.

4.3.2. Explanations of obtained results

First, we take an interest in explaining the quality of obtained results. The observed gains may come from the concurrency (i.e. the use of multiple orders of variables) or from the cooperation (i.e. the exchange of nogoods). In order to determine the origins of gains, we compare the cooperative concurrent method with concurrent method without cooperation. Table 4 shows the efficiency obtained by this concurrent method without cooperation for the five considered classes of random CSPs (for consistent and inconsistent problems). Table 5 (respectively table 6) provides these results for consistent problems (resp. inconsistent ones).

We first note that, in most cases, the cooperative method obtains better results than the concurrent method. Some exceptions appear for consistent problems. For such problems, the efficiency of the cooperative method is generally close to one of the concurrent method. That means that the good quality of the obtained results mainly comes from the concurrency. However, in many cases, the cooperative method obtains a significant better efficiency, what implies that the cooperation is involved too. On the other hand, for inconsistent problems, we observe that the concurrent method never obtains linear or superlinear speed-up. In other words, the linear or superlinear speed-up (and more generally the good results) presented by our cooperative method are mostly due to the cooperation. For consistent problems, the contribution of the cooperation is less important because solvers don't explore their whole search tree since the search is finished as soon as a solution is found.

For inconsistent problems, we must underline the predominant role of values heuristics. For each solver s (except one if the number of solvers is odd), there exists a solver which assigns first the same variable as s, which uses a similar variables heuristic and whose values heuristic is the reverse of s's one. Without exchanging nogoods, these two solvers visit similar search trees. With exchanging nogoods, each one explores only a part of its search tree thanks to received nogoods.

We focus then on possible reasons of efficiency decrease. With a method like ours, an usual reason of efficiency lack is the cost of communications. Our method doesn't make exception. But, in our case, there is another reason which explains the decrease of performances. Indeed, the efficiency of the concurrent method decreases too when p increases. As the concurrent method doesn't use any cooperation form, the cost of communications can't be involved in this efficiency decrease. In our concurrent methods with/without cooperation, solvers differ from each other in the heuristics they use for ordering values and variables. The diversity of these heuristics is the main feature of our approach from the concurrency viewpoint. Unfortunately, it seems that this diversity is not sufficient. In particular, we observe this phenomenon for classes (50,15,245,93) and (50,25,123,439). For these two classes, the concurrent method appears to be less efficient than for other classes. When we exploit the cooperative method, the cooperation permits to limit the decrease of performance due to the lack of diversity and we then observe a better speed-up. Nevertheless, the contribution of the cooperation is not important enough to obtain linear or superlinear speed-up. Remark that the lack of diversity is directly explained by the way we build the different heuristics. In other words, by improving the way we choose our heuristics (i.e. by improving the diversity), we may expect a better efficiency for our cooperative concurrent method.

Class (n, d, m, t)	Scheme	p				
		2	4	6	8	10
(50,15,184,112)	S_0	1.304	1.174	1.228	1.132	1.084
	S_{light}	1.313	1.167	1.220	1.108	1.087
	S_{gest}	1.356	1.219	1.267	1.162	1.135
(50,15,245,93)	S_0	1.485	1.401	1.053	0.984	0.842
	S_{light}	1.488	1.398	1.057	0.977	0.843
	S_{gest}	1.537	1.425	1.080	1.012	0.853
(50,25,,23,439)	S_0	1.236	1.189	0.941	1.093	1.001
	S_{light}	1.240	1.211	0.969	1.124	1.044
	S_{gest}	1.403	1.314	1.076	1.230	1.175
(50,25,150,397)	S_0	1.115	1.647	1.553	1.571	1.388
	S_{light}	1.132	1.657	1.523	1.570	1.395
	S_{gest}	0.980	1.517	1.441	1.473	1.309
(75,10,277,43)	S_0	1.368	1.444	1.368	1.298	1.258
	S_{light}	1.363	1.450	1.433	1.267	1.249
	S_{gest}	1.350	1.463	1.392	1.267	1.244

Table 2: Efficiency obtained by the cooperative concurrent method with the schemes S_0, S_{light} and S_{gest} for consistent problems.

Class (n, d, m, t)	Scheme	p				
		2	4	6	8	10
(50,15,184,112)	S_0	1.128	1.091	1.085	1.079	1.017
	S_{light}	1.132	1.099	1.084	1.073	1.022
	S_{gest}	1.134	1.106	1.083	1.053	1.003
(50,15,245,93)	S_0	1.024	1.014	0.991	0.917	0.847
	S_{light}	1.021	1.018	0.995	0.923	0.853
	S_{gest}	1.050	1.023	1.001	0.923	0.848
(50,25,123,439)	S_0	1.016	0.982	0.936	0.943	0.875
	S_{light}	1.028	1.000	0.963	0.974	0.915
	S_{gest}	1.082	1.041	0.962	0.913	0.906
(50,25,150,397)	S_0	1.048	1.174	1.190	1.104	1.075
	S_{light}	1.062	1.174	1.180	1.096	1.081
	S_{gest}	1.094	1.250	1.263	1.179	1.143
(75,10,277,43)	S_0	1.277	1.101	1.130	1.066	1.042
	S_{light}	1.271	1.107	1.128	1.051	1.031
	S_{gest}	1.270	1.075	1.124	1.034	1.006

Table 3: Efficiency obtained by the cooperative concurrent method with the schemes S_0, S_{light} and S_{gest} for inconsistent problems.

Class (n, d, m, t)	p				
	2	4	6	8	10
(50,15,184,112)	0.851	0.538	0.424	0.349	0.292
(50,15,245,93)	0.663	0.408	0.289	0.223	0.182
(50,25,123,439)	0.764	0.495	0.385	0.331	0.271
(50,25,150,397)	0.786	0.753	0.584	0.475	0.423
(75,10,277,43)	1.116	0.779	0.627	0.522	0.441

Table 4: Efficiency obtained by the concurrent method for consistent and inconsistent problems.

4.3.3. Number of exchanged messages

In this part, we compare, for the three schemes, the number of messages exchanged by the cooperative concurrent method. Table 7 (respectively table 8) shows the number of exchanged messages for unary nogoods (resp. binary nogoods) for consistent and inconsistent problems. For information, we observe a similar trend if we only focus on consistent problems or inconsistent ones.

First, we note that in scheme S_{gest}, when we only exploit a single solver, some messages are sent. This is due to the communications with the nogood manager.

Then, for unary nogoods, we observe that the cooperative method exchanges as many messages in S_0 as in S_{light}. Generally, it is the same for scheme S_{gest}. However, in some cases, fewer messages are sent because the number of produced unary nogoods turns out to be less important than in the two other schemes. The similarity of results for the three schemes is mostly explained by the weak restriction achieved in schemes S_{light} and S_{gest}. In effect, in these two schemes, the communications are limited since, for unary nogoods, doubles are only conveyed once. However, as there are few doubles, schemes S_{light} and S_{gest} allow our cooperative method to save only few messages and so we obtain similar results for the three scheme.

For binary nogoods, the number of exchanged messages is significantly more important than one for unary nogoods because solvers produces more binary nogoods than unary ones. Scheme S_{light} then appears to be better than both S_{gest} and S_0 while in S_{gest} solvers exchange fewer messages than in S_0. Scheme S_{light} only differs from S_0 in limiting the exchange of nogoods. Indeed, in S_{light}, we restrict the communication to its strict minimum by sending only binary nogoods to a part of solvers. It is the same for scheme S_{gest}. So, the gap between S_0 and S_{light} (or between S_0 and S_{gest}) shows the contribution of our communication restriction. Thanks to this limitation, the number of messages is significantly less important in S_{light} or in S_{gest} than in S_0. Thus we can expect that the number of solvers above of which the cost of communications penalizes the efficiency is greater in S_{light} or S_{gest} than in S_0. Regarding the difference between S_{light} and S_{gest}, it results from the systematic communication of every nogood to the manager. In particular, among all these communications, many of them correspond to doubles. For binary nogoods, the number of doubles is clearly more important (with a multiplicative factor between 10 and 100) in S_{gest} than one in S_{light}. Such a number is due to the period of time elapsed between the production of a nogood and its addition to the constraint set in shared memory. In S_{gest}, this period is greater than in S_{light} because nogoods are added to the constraint set by the manager, and not directly by the solver as in S_{light}.

Finally, if we focus on the results obtained by S_{light}, we note that the number of messages exchanged for binary nogoods is significantly less important than the number of binary nogoods. It ensues that a binary nogood is only useful for a few solvers (and even for none in some cases). Of course, it is the same in S_{gest}. So restricting the exchanges allows solvers to reduce the time spent for managing communications. In particular, it prevents solvers from receiving and managing useless informations. By so doing, solvers are fully devoted to solving the problem.

4.3.4. Summary

For the three schemes, the cooperative concurrent method obtains linear or superlinear speed-up up to ten solvers for three out of five classes of random instances we consider. For the two remaining classes, the speed-up is either linear or superlinear, or sublinear depending on the number of used solvers. On the overall, we have noted that the efficiency is better for consistent problems than for inconsistent ones. Nevertheless we have observed linear or superlinear speed-up for inconsistent instances of some classes. The good quality of these results is mainly due to the cooperation for inconsistent problems and to the concurrency for consistent problems. For consistent problems, the cooperation is partly involved too. We have remark a decrease of efficiency when the number of solvers increases. Such a decrease comes from the lack of diversity of solvers. Regarding to the exchanges of nogoods, we have remarked that binary nogoods are generally useful for a few solvers. This result explains that we exchange fewer messages in S_{light} than in S_{gest} or in S_0 and it shows the interest of the communication restriction. However, in our experimentations, it seems that the exchanges of messages is cheap since we obtain similar results independently of the used scheme. If these exchanges are more expensive, we would exploit the scheme S_{light}. That is why, in the following results, we consider that the cooperative method uses this scheme.

Class (n, d, m, t)	p				
	2	4	6	8	10
(50,15,184,112)	1.395	1.140	1.113	1.043	1.080
(50,15,245,93)	1.574	1.316	0.969	0.854	0.721
(50,25,123,439)	1.262	1.105	0.797	0.931	0.802
(50,25,150,397)	1.052	1.654	1.456	1.526	1.428
(75,10,277,43)	1.362	1.248	1.238	1.181	1.107

Table 5: Efficiency obtained by the concurrent method for consistent problems.

Class (n, d, m, t)	p				
	2	4	6	8	10
(50,15,184,112)	0.621	0.358	0.267	0.214	0.173
(50,15,245,93)	0.566	0.339	0.239	0.183	0.149
(50,25,123,439)	0.556	0.325	0.259	0.206	0.167
(50,25,150,397)	0.589	0.438	0.326	0.249	0.219
(75,10,277,43)	0.789	0.419	0.295	0.229	0.186

Table 6: Efficiency obtained by the concurrent method for inconsistent problems.

Class (n, d, m, t)	Scheme	p					
		1	2	4	6	8	10
(50,15,184,112)	S_0	0	2.84	21.09	47.99	74.67	100.99
	S_{light}	0	2.82	20.88	47.42	74.54	99.84
	S_{gest}	0.12	0.33	20.40	44.15	67.13	88.91
(50,15,245,93)	S_0	0	4.61	30.35	80.77	142.57	211.41
	S_{light}	0	4.56	29.98	79.87	140.87	207.51
	S_{gest}	0	0.05	28.22	80.12	138.84	203.52
(50,25,123,439)	S_0	0	8.66	47.81	103.35	161.37	224.54
	S_{light}	0	8.67	47.90	103.91	156.87	227.27
	S_{gest}	1.94	4.23	38.57	82.24	126.00	176.69
(50,25,150,397)	S_0	0	5.53	43.32	95.10	155.14	221.99
	S_{light}	0	5.50	43.05	94.42	154.30	222.60
	S_{gest}	0.30	0.67	43.37	89.96	146.34	208.76
(75,10,277,43)	S_0	0	1.70	13.75	31.51	52.39	72.76
	S_{light}	0	1.69	13.70	30.99	51.14	70.83
	S_{gest}	0.40	0.94	14.85	32.06	50.78	68.04

Table 7: Number of messages which are sent for exchanging unary nogoods by all solvers according the used scheme for consistent and inconsistent problems.

4.4. Cooperative method vs classical algorithms

4.4.1. For a monoprocessor system

In a monoprocessor system, we must take into account the work of each solver. So the results we consider for the cooperative method correspond to the sum of results obtained by each solver. Tables 9 and 10 present respectively the sum of the run-time and the sum of the number of constraint checks for the cooperative method (with 1, 2 or 4 solvers) and the run-time and the number of constraint checks for each classical algorithm, namely FC, FC-CBJ, FC-NR and MAC.

First, we observe that our cooperative method with a single solver does not obtain the same results as the FC-NR algorithm. This is simply explained by the variable ordering used by the cooperative method. This ordering consists in fixing the choice of the first variable and then using the heuristic dom/st. When the first variable is the same as FC-NR's one, FC-NR and the cooperative method with a single solver obtain the same results. We note that these two methods generally perform more constraint checks than FC, FC-CBJ and MAC. That is explained by the constraints which are added to the problem during the search. Indeed, FC-NR checks the initial constraints and the added constraints while the other classical algorithms only check the initial constraints. These additional constraint checks permit to enforce an additional pruning of the tree-search. Unfortunately, it seems that it does not generally allow to save enough run-time.

On the other hand, as soon as we use two or four solvers, the cooperative method obtains either equivalent results to FC-NR's ones, or better results (with a gain up to 21%). These results are due to the concurrency and the cooperation which allow our approach to reduce the number of constraint checks. For some classes, the obtained improvement makes the cooperative method faster than FC-CBJ or MAC while FC-NR is slower than these two algorithms. However, in most cases, the cooperative method appears slower than FC or FC-CBJ.

We observe a similar trend if we focus on consistent problems or on inconsistent ones. Nevertheless, for the consistent problems, the cooperative method with two or four solvers performs, for many classes, fewer constraint checks than FC (see table 12). However, these gains do not mean that our method is faster than FC (see table 11). Note that obtaining a better trend for consistent instances is foreseeable since the cooperative method is more efficient for such problems.

4.4.2. For a multiprocessor system

In a multiprocessor system (with a processor per solver), the results we consider for the cooperative method are ones of the solver which solves the problem first. As we do not have a multiprocessor computer, the results we provide are ones of a simulation on a monoprocessor computer. Tables 13 and 14 show respectively the run-time and the number of constraint checks for the cooperative method (with 1, 2 or 4 solvers) and for the four classical algorithms. As we observe a similar trend for consistent problems and for inconsistent ones, we only present the results for both consistent and inconsistent instances.

If the cooperative method only exploits one solver, the results are the same as one obtained in a monoprocessor system. In contrast, when it runs two or four solvers, it performs significantly fewer constraint checks than the classical algorithms we use. Then it is clearly faster than them. Note that we observe similar results when we increase the number of solvers. The quality of these results is mostly due to the great practical efficiency of the cooperative method.

The results we have provided are obtained by simulating parallelism. Of course, now, we must experiment this approach on a real parallel system in order to confirm the trend we have observed . Furthermore, it could be interesting to compare our approach with some parallel version of FC or MAC.

4.5. Behaviour for real-world instances

Now we study the behaviour of our cooperative concurrent method on some real-world instances from the FullRLFAP archive. Tables 15 and 16 show respectively the run-time and the number of constraint checks for the cooperative method (with 1, 2 or 4 solvers) and for the four classical algorithms.

For most of the exploited instances (SCEN-01, SCEN-11 and all the considered GRAPH instances), we obtain similar results for one, two or four solvers. On the one hand, this result is explained by a very limited cooperation due to a small number of nogoods. Note that these instances are consistent, what explains that few nogoods are produced. On the other hand, for these instances, the cooperative search does not take advantage of the concurrency. With respect to classical algorithm, the cooperative method

Class (n, d, m, t)	Scheme	p					
		1	2	4	6	8	10
(50,15,184,112)	S_0	0	357	1,322	2,437	3,655	4,944
	S_{light}	0	16	77	149	211	295
	S_{gest}	462	487	652	807	955	1,060
(50,15,245,93)	S_0	0	208	948	1,968	3,308	4,970
	S_{light}	0	11	85	183	310	463
	S_{gest}	240	247	440	625	840	1,077
(50,25,123,439)	S_0	0	1,375	4,573	8 561	12,100	16,017
	S_{light}	0	40	145	284	348	497
	S_{gest}	1,708	1,806	2,140	2,671	3,013	3,153
(50,25,150,397)	S_0	0	1,364	4,669	8,583	12,634	17,349
	S_{light}	0	37	174	326	464	653
	S_{gest}	1,668	1,761	2,132	2,502	2,788	3,111
(75,10,277,43)	S_0	0	434	1,474	2,564	3,885	5,237
	S_{light}	0	26	107	181	289	387
	S_{gest}	612	665	831	982	1,143	1,374

Table 8: Number of messages which are sent for exchanging binary nogoods by all solvers according the used scheme for consistent and inconsistent problems.

Class (n, d, m, t)	cooperative method			FC	FC-CBJ	FC-NR	MAC
	$p = 1$	$p = 2$	$p = 4$				
$(50, 15, 184, 112)$	1,599	1,325	1,426	795	949	1,331	1,472
$(50, 15, 245, 93)$	12,711	11,567	11,724	7,,72	8,462	11,758	18,612
$(50, 25, 123, 439)$	662	595	613	576	632	635	599
$(50, 25, 150, 397)$	7,164	6,513	5,099	3,462	4,046	6,191	5,848
$(75, 10, 277, 43)$	2,772	2,084	2,114	1,591	1,846	1,912	1,211

Table 9: Run-time (in ms) of the cooperative method with scheme S_{light} (in a monoprocessor system) and of the classical algorithms for consistent and inconsistent problems.

Class (n, d, m, t)	cooperative method			FC	FC-CBJ	FC-NR	MAC
	$p = 1$	$p = 2$	$p = 4$				
$(50, 15, 184, 112)$	11,246	9,234	10,045	7,347	7,089	9,984	4,750
$(50, 15, 245, 93)$	85,394	76,259	77,654	64,695	63,629	81,947	53,354
$(50, 25, 123, 439)$	5,122	4,669	4,888	5,969	5,208	5,479	2,601
$(50, 25, 150, 397)$	54,249	49,680	39,436	34,743	32,648	52,620	21,813
$(75, 10, 277, 43)$	17,297	12,820	13,089	12,279	11,406	12,496	3,533

Table 10: Number of constraint checks (in thousands) of the cooperative method with scheme S_{light} (in a monoprocessor system) and of the classical algorithms for consistent and inconsistent problems.

Class	cooperative method			FC	FC-CBJ	FC-NR	MAC
(n, d, m, t)	$p = 1$	$p = 2$	$p = 4$				
$(50, 15, 184, 112)$	1,256	964	1,088	616	738	1,016	1,073
$(50, 15, 245, 93)$	8,023	5,398	5,746	4,767	5,708	7,622	12,147
$(50, 25, 123, 439)$	826	666	683	675	741	738	545
$(50, 25, 150, 397)$	7,293	6,448	4,410	3,130	3,663	5,619	4,693
$(75, 10, 277, 43)$	2,717	1,998	1,900	1,561	1,811	1,885	1,088

Table 11: Run-time (in ms) of the cooperative method with scheme S_{light} (in a monoprocessor system) and of the classical algorithms for consistent problems.

Class	cooperative method			FC	FC-CBJ	FC-NR	MAC
(n, d, m, t)	$p = 1$	$p = 2$	$p = 4$				
$(50, 15, 184, 112)$	8,585	6,511	7,478	5,601	5,397	7,342	3,420
$(50, 15, 245, 93)$	53,200	34,602	37,021	43,236	42,481	51,955	34,613
$(50, 25, 123, 439)$	6,060	5,036	5,188	6,708	5,781	6,051	2,240
$(50, 25, 150, 397)$	54,224	48,267	33,372	30,732	28,772	46,519	17,088
$(75, 10, 277, 43)$	16,757	12,166	11,672	11,919	11,049	12,215	3,139

Table 12: Number of constraint checks (in thousands) of the cooperative method with scheme S_{light} (in a monoprocessor system) and of the classical algorithms for consistent and inconsistent problems.

Class	cooperative method			FC	FC-CBJ	FC-NR	MAC
(n, d, m, t)	$p = 1$	$p = 2$	$p = 4$				
$(50, 15, 184, 112)$	1,599	659	353	795	949	1,331	1,472
$(50, 15, 245, 93)$	12,711	5,781	2,928	7,072	8,462	11,758	18,612
$(50, 25, 123, 439)$	662	295	151	576	632	635	599
$(50, 25, 150, 397)$	7,164	3,254	1,272	3,462	4,046	6,191	5,848
$(75, 10, 277, 43)$	2,772	1,039	523	1,591	1,846	1,912	1,211

Table 13: Run-time (in ms) of the cooperative method with scheme S_{light} (in a multiprocessor system) and of the classical algorithms for consistent and inconsistent problems.

Class	cooperative method			FC	FC-CBJ	FC-NR	MAC
(n, d, m, t)	$p = 1$	$p = 2$	$p = 4$				
$(50, 15, 184, 112)$	11,246	4,593	2,507	7,347	7,089	9,984	4,750
$(50, 15, 245, 93)$	85,394	38,125	19,390	64,695	63,629	81,947	53,354
$(50, 25, 123, 439)$	5,122	2,339	1,236	5,969	5,208	5,479	2,601
$(50, 25, 150, 397)$	54,249	24,800	9,853	34,743	32,648	52,620	21,813
$(75, 10, 277, 43)$	17,297	6,391	3,253	12,279	11,406	12,496	3,533

Table 14: Number of constraint checks (in thousands) of the cooperative method with scheme S_{light} (in a multiprocessor system) and of the classical algorithms for consistent and inconsistent problems.

obtains similar results to FC-NR's ones.
For instances SCEN-04 and SCEN-08, solvers produce more nogoods. The cooperation and the concurrency then allow our method to improve the run-time when the number of solvers increases. However, the gains are too slight to obtain linear or superlinear speed-up. So running our method on a monoprocessor system is not interesting because the sum of the run-time is too important. For the SCEN-11 instance, the run-time is similar for one, two or four solvers. In contrast, for six solvers, the cooperative method requires only 130 milliseconds for solving the problem (not shown in table 15), which permits to obtain a superlinear speed-up. For information, this result is only obtained thanks to the concurrency, since no nogood is exchanged. If we consider a monoprocessor system, the run-time (obtained by summing the run-time of these six solvers) is 770 milliseconds. So our cooperative method is faster than the four classical algorithms. Finally, for SCEN-05 instance, the cooperative method with two solvers presents similar results with respect to ones obtained with a single solver. For four solvers, the concurrency and the cooperation improve significantly the run-time. The run-time in a monoprocessor system is then 1210 milliseconds. The number of saved constraint checks is also important. So, for this instance, the cooperative concurrent method turns out to be significantly better than the four classical algorithms in a monoprocessor system as in a multiprocessor system.
To sum up, among the real-world instances we consider, some problems cannot be efficiently solved by the cooperative method because when few nogoods are produced and the contribution of the concurrency is too slight . Of course, for such instances, our approach doesn't obtain interesting results. For some other instances like SCEN-04 and SCEN-08, it takes advantage of the cooperation and of the concurrency. However, the gain are too slight for running our approach on a monoprocessor system. For some instances like SCEN-05 or SCEN-11, the run-time in a monoprocessor system (i.e. the sum of the run-time of each solver) outperforms the run-time of the four classical algorithm. So, for such instances, running our cooperative method on a monoprocessor system can be seen as an interesting alternative solution to classical algorithms. For information, these results are then due to either concurrency, or both concurrency and cooperation. Finally, note that by using some heuristics specific to these instances, we might perhaps improve these results.

5. Discussion about some possible extensions

In this section, we discuss about some possible extensions of this work. First, we focus on a generalization to any algorithm which maintains some level of consistency. Then we are interested in defining other cooperative schemes by relaxing some assumptions.

5.1. Generalization to any filtering

The schemes and the notions we have described above for the FC-NR algorithm can be extended to any algorithm which maintains some level of consistency and which uses nogood recording. First, the notion of induced CSP can be generalized with any filter ϕ. Then, all definitions which use this notion can also be extended. In particular, we can generalize FC-consistency to ϕ-consistency. Next, in the interpretation phase, according to the chosen filter ϕ, it is necessary to propagate some removals, in order to maintain the level of consistency. For instance, if we use the algorithm MAC with nogood recording, the receipt of a unary nogood induces a propagation of the deletion of the corresponding value whereas FC-NR only removes this value. Of course, the backjump phase we use must be adapted too.

5.2. Adaptations to some other schemes

This part is devoted to the relaxation of the assumption about the shared memory. We remind that in our cooperative concurrent method, we assume that we have a shared memory which is big enough in order to contain the instance we want to solve, the set of recorded nogoods and the current instantiation of each solver. In contrast, we do not study the relaxation of the second assumption because such a relaxation is opposite to the cooperation concept.
A first way consists in recording in shared memory only a part of the current instantiation of each solver. Such a relaxation may be interesting when writing some information in shared memory is too expensive or is significantly more expensive than recording this information in its own private memory. In such a case, updating the current instantiation after each modification can penalize the efficiency of the cooperative method. So, a solver only stores in shared memory the current instantiation of the shallow variables.

Instance	méthode coopérative			FC	FC-CBJ	FC-NR	MAC
	$p = 1$	$p = 2$	$p = 4$				
SCEN-01	100	100	90	100	110	100	640
SCEN-04	170	110	50	120	50	170	90
SCEN-05	19,170	19,090	300	-	335,300	18,430	14,310
SCEN-08	120	80	20	20	20	120	270
SCEN-11	2,620	2,610	2,610	12,740	1,250	2,930	25,630
GRAPH-08	110	90	90	70	60	110	430
GRAPH-09	120	110	110	100	110	130	670
GRAPH-10	690	690	690	-	-	680	930
GRAPH-14	110	100	100	100	110	120	530

Table 15: Run-time (in ms) of the cooperative method with scheme S_{light} (in a multiprocessor system) and of the classical algorithms for some problems of the FullRLFAP archive.

Instance	cooperative method			FC	FC-CBJ	FC-NR	MAC
	$p = 1$	$p = 2$	$p = 4$				
SCEN-01	176.3	176.3	176.3	185.4	185.4	176.3	1,857.7
SCEN-04	208.0	140.7	61.9	255.3	51.1	203.1	246.0
SCEN-05	31,259.2	31,199.4	630.0	-	829,057.7	31,251.1	9,220.9
SCEN-08	356.3	256.4	85.7	52.7	46.8	356.3	2,346.4
SCEN-11	5,459.7	5,459.7	5,459.7	32,095.2	2,828.3	5,459.7	22,520.8
GRAPH-08	204.2	183.7	183.7	158.4	158.4	204.2	1,251.8
GRAPH-09	191.0	190.8	190.8	199.2	199.2	191.0	1,819.7
GRAPH-10	1,500.3	1,500.1	1,500.1	-	-	1,498	2,531
GRAPH-14	174.0	173.9	173.0	183.4	183.4	174.0	1,599.2

Table 16: Number of constraint checks (in thousands) of the cooperative method with scheme S_{light} (in a multiprocessor system) and of the classical algorithms for some problems of the FullRLFAP archive.

We can then fix the depth limit according to the algorithm we use and the instance we want to solve. Intuitively, the nogoods which involve shallow variables are the most interesting because they permit a more powerful pruning. Furthermore, we can expect that beyond a given depth, the solver quickly finds a solution or encounters a failure. Then it is likely that receiving a nogood which prevents such a failure from occurring saves few nodes. So such a limitation of the instantiation updates could be interesting. Note that it may be use jointly with the limitation of exchanges of schemes S_{light} and S_{gest} in order to reduce the communication cost.

If we do not have a shared memory, we consider a distributed scheme. In this scheme, each solver has its own copy of the instance. In particular, it has its own constraint set (which includes the constraints we add thanks to nogoods). By so doing, when a solver finds a nogood, it must send it to all the other solvers in order that they add it to their own constraint set. Of course, doubles are communicated once only. With such a scheme, the cooperative concurrent method will be efficient if the communications are not too expensive. Finally, it is worth noting that this scheme may be useful, for instance, if we run our cooperative method on computing clusters.

6. Conclusion and future works

In [15], a cooperative concurrent method is presented. In this method, the cooperation consists in exchanging nogoods (i.e. instantiations which cannot lead to a solution). In this article, we have developed and extended this work. We have proposed a cooperative concurrent search whose all solvers run the same algorithm (namely FC-NR), use different heuristics for ordering variables and/or values and exchange nogoods. We have associated a process to each solver (i.e. our approach is turned to multiprocessor systems). For exchanging nogoods, we consider two cooperation forms which assume that each process is able to communicate with each other and to access a shared memory which is big enough to contain the instance we want to solve, the produced nogoods and the current instantiation of each solver. The first form consists in adding the produced nogoods to the problem as new or tightened constraints. As the problem is stored in shared memory, each solver can then use a nogoods produced by another solver in order to prune its own search tree. In the second form, each solver communicates the nogoods it finds to some other solvers by sending messages. We have then defined three cooperative schemes which exploit these two cooperation form. Two of them restrict the exchange of messages since a nogood is only conveyed to a solver if this solver is liable to use it immediately on its receipt. Furthermore, in order to exploit the received nogoods, we have added an interpretation phase to the FC-NR algorithm. With regard to the heuristics, we have explained why a trade-off is required between the concurrency and the cooperation and we have proposed such a trade-off.

We have first experimented our method on random instances. We have then obtained interesting results since we observed linear or superlinear speed-up for consistent problems as for inconsistent ones up to ten solvers. So exchanging nogoods appears to be an efficient cooperation form. Nevertheless, we have noted a decrease of efficiency as the number of solvers increases. In some cases, it ensues the appearance of sublinear speed-up. This decrease is mainly due to a lack of diversity (i.e. the heuristics are not different enough). Compared with some classical state-of-the-art algorithms, our cooperative concurrent method is often more efficient than the FC-NR algorithm and is sometimes faster than FC-CBJ or MAC (mainly for the consistent instances) if we use a monoprocessor system. In contrast, in a a multiprocessor system, the cooperative method turns out to be better than classical algorithms in most cases.

For real-world instances, we observe various different trends. For some instances, the cooperative method obtains poor results because it does not take advantage of the concurrency and few nogoods (or no nogood) are produced and exchanged. For some other instances, there exists a contribution of the cooperation and the concurrency. Unfortunately, their contribution is not sufficient to obtain linear or superlinear speed-up. On the other hand, when this contribution is significant, our cooperative method outperforms all the classical algorithms in a monoprocessor system as in a multiprocessor one.

A first extension of this work consists in experimenting our approach on a real multiprocessor computer in order to confirm the trends we have observed. In the same time, it could be interesting to propose several efficient and diverse heuristics in order to improve the efficiency as well as increase the number of solvers. We can also define a new trade-off between the concurrency and the cooperation. Then, we

can extend our method by applying any algorithm which maintains some level of consistency or by using different algorithms (which would permit to combine complete search methods and incomplete ones like in [11] and so to improve the diversity of solvers). Another extension consists in exchanging an other kind of informations. For instance, we can exchange structural nogoods and use the BTD algorithm [13] as basic method for the solvers. Finally, it seems natural to extend this works to valued CSPs [18] (which allows to express optimization problems). For this extension, a part of the framework is already defined in [6] (namely the notion of valued nogood and the valued Nogood-Recording algorithm).

Bibliography

1. C. Bessière and J.-C. Régin. MAC and Combined Heuristics: Two Reasons to Forsake FC (and CBJ?) on Hard Problems. In *Proceedings of the 2nd International Conference on Principles and Practice of Constraint Programming (CP'96)*, pages 61–75, 1996.
2. C. Bessière and J.-C. Régin. Refining the Basic Constraint Propagation Algorithm. In *Proceedings of the 17th International Joint Conference on Artificial Intelligence (IJCAI-01)*, pages 309–315, 2001.
3. C. Cabon, S. de Givry, L. Lobjois, T. Schiex, and J. P. Warners. Radio Link Frequency Assignment. *Constraints*, 4:79–89, 1999.
4. X. Chen and P. van Beek. Conflict-Directed Backjumping Revisited. *Journal of Artificial Intelligence Research*, 14:53–81, 2001.
5. S. Clearwater, B. Huberman, and T. Hogg. Cooperative Solution of Constraint Satisfaction Problems. *Science*, 254:1181–1183, Nov. 1991.
6. P. Dago and G. Verfaillie. Nogood Recording for Valued Constraint Satisfaction Problems. In *Proceedings of the 8th International Conference on Tools with Artificial Intelligence (ICTAI'96)*, pages 132–139, 1996.
7. R. Dechter. Enhancement Schemes for Constraint Processing: Backjumping, Learning, and Cutset Decomposition. *Artificial Intelligence*, 41:273–312, 1990.
8. J. Gaschnig. Performance Measurement and Analysis of Certain Search Algorithms. Technical Report CMU-CS-79-124, Carnegie-Mellon University, 1979.
9. R. Haralick and G. Elliot. Increasing tree search efficiency for constraint satisfaction problems. *Artificial Intelligence*, 14:263–313, 1980.
10. T. Hogg and B. Huberman. *Better Than The Best: The Power of Cooperation*, pages 164–184. SFI 1992 Lectures in Complex Systems. Addison-Wesley, 1993.
11. T. Hogg and C.P. Williams. Solving the Really Hard Problems with Cooperative Search. In *Proceedings of the 11th National Conference on Artificial Intelligence (AAAI-93)*, pages 231–236, 1993.
12. T. Hogg and C.P. Williams. Expected Gains from Parallelizing Constraint Solving for Hard Problems. In *Proceedings of the 12th National Conference on Artificial Intelligence (AAAI-94)*, pages 331–336, 1994.
13. P. Jégou and C. Terrioux. Hybrid backtracking bounded by tree-decomposition of constraint networks. *Artificial Intelligence*, (?), 2003. To appear.
14. G. Kondrak and P. van Beek. A Theorical Evaluation of Selected Backtracking Algorithms. *Artificial Intelligence*, 89:365–387, 1997.
15. D. Martinez and G. Verfaillie. Echange de Nogoods pour la résolution coopérative de problèmes de satisfaction de contraintes. In $2^{ème}$ *Conférence Nationale sur la Résolution de Problèmes NP-Complets (CNPC 96)*, pages 261–274, 1996. In french.
16. P. Prosser. Hybrid Algorithms for the constraint satisfaction problem. *Computational Intelligence*, 9:268–299, 1993.
17. D. Sabin and E. Freuder. Contradicting Conventional Wisdom in Constraint Satisfaction. In *Proceedings of the 11th European Conference on Artificial Intelligence (ECAI-94)*, pages 125–129, 1994.
18. T. Schiex, H. Fargier, and G. Verfaillie. Valued Constraint Satisfaction Problems: hard and easy problems. In *Proceedings of the 14th International Joint Conference on Artificial Intelligence (IJCAI-95)*, pages 631–637, 1995.
19. 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.
20. C. Terrioux. Cooperative Search and Nogood Recording. In *Proceedings of the 17th International Joint Conference on Artificial Intelligence (IJCAI-01)*, pages 260–265, 2001.