Cooperative search vs classical algorithms
Research Report Number LSIS/2002/006, June 14th 2002

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 have presented, in a previous work ([15]), a cooperative parallel search for solving the constraint satisfaction problem. We run independently p solvers based on Forward-Checking with Nogood Recording. The solvers exchange nogoods via a process ("the manager of nogoods") which regulates the exchanges. Solvers exploit the nogoods they receive to limit the size of their search tree. Experimentally, we have shown the interest of our approach from a parallel viewpoint, namely we have obtained linear or superlinear speed-up. However, we haven't studied its behavior with respect to classical algorithms.
In this paper, we first improve the exploitation of received nogoods. Then, we provide experimental comparisons between the cooperative method and some state-of-the-art algorithms. In particular, we observe that the cooperative search with at least four solvers (even in some cases from two solvers) is faster than classical algorithms like FC or MAC.

Key words : Constraint satisfaction problem, cooperative search.

1 Introduction

In the framework of the constraint satisfaction problem (CSP), one of main tasks consists in determining whether there exists a solution, i.e. an instantiation of all variables which satisfies all constraints. This task is a NP-complete problem. The basic enumerative algorithm for solving constraint satisfaction problem 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. Forward-Checking (noted FC [7]) and Maintaining Arc-Consistency (noted MAC [13]) are examples of such algorithms. 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 [6], Graph-based Backjumping [5] or Conflict-Directed Backjumping [12]. Finally, learning techniques which prevent some redundancies in the search tree by recording some informations have been proposed, for instance Constraint Learning [5] or Nogood Recording [14]. From these three kinds of improvements, hybrid methods have been produced like Forward-Checking with Nogood Recording (noted FC-NR [14]) or Forward-Checking with Conflict-directed Backjumping (noted FC-CBJ [12]). 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.
In the last years, in order to speed up the resolution of problems, more and more parallel searches are used. A basic one is *independent parallel search* which consists in running several solvers on the same problem instead of

a single solver. Each solver differs from other ones by using different algorithms or heuristics. The aim is that at least one of the solvers is suitable for the problem which we solve. Tested on the graph coloring problem ([10]), this approach has better results than a classical resolution with a single solver, but the gains seem limited. Hogg and Williams recommend then the use of a cooperative parallel search. A *cooperative parallel search* is based on the same ideas as the independent search with in addition an exchange of informations between solvers, in order to guide solvers to a solution, and then, to speed up the resolution. Experimental results on cryptarithmetic problems ([4, 8]) and on graph coloring problems ([8, 9]) show a significant gain in time with respect to an independent search. In both cases, the exchanged informations correspond to partial consistent instantiations.
In [11], a cooperation based on exchanging nogoods (i.e. instantiations which can't be extended to a solution) is proposed. Each solver runs the FC-NR algorithm and exchange nogoods it produces with the other solvers. Exchanged nogoods permit solvers to prune their own search tree, and so one can expect to find more quickly a solution. However, the experimental results they provide don't show the interest of such an approach. In [15], we define a new scheme of cooperation with exchange of nogoods. Like previously, each solver is based on FC-NR, but, in addition to sending and receipt of messages, we add a phase of interpretation to FC-NR, in order to limit the size of the search tree according to received nogoods. Furthermore, in a view to reducing the cost of communications, we introduce a manager of nogoods whose role is to regulate the exchange of nogoods.
In this paper, we first improve the phase of interpretation. In [15], this phase doesn't exploit all the received nogoods, in order to prevent the algorithm from loosing a fundamental property, namely that every instantiation built by FC-NR is FC-consistent. The improved phase we present corrects this drawback.
From a practical point of view, we can consider two assessments of the performance of a parallel algorithm. The first one is based on the notions of speed-up and efficiency, which measure the quality of the algorithm from a parallel viewpoint. The second one consists in comparing the parallel search and the main state-of-the-art algorithms (sequential ones included). Thanks to such comparisons, we can assess the benefit of the parallel search with respect to well-know existing methods and so we can choose more easily a method or an other one for solving a given problem. In [15], we show experimentally the interest of our cooperative scheme. In particular, we obtain linear or superlinear speed-up for consistent problems, like for inconsistent ones, up to about ten solvers and we observe that our method is faster than concurrent one. Nevertheless, we don't compare this approach with classical sequential algorithms like FC or MAC.
In this contribution, our second main aim is to assess the behaviour of this cooperative method with respect to some state-of-the-art sequential algorithms, namely FC, FC-CBJ, FC-NR and MAC.
The plan is as follows. In section 2, we give basic notions about CSPs, nogoods and FC-NR. Then, in section 3, we remind the cooperative scheme and we then present the improved phase of interpretation. Finally, after providing experimental comparisons between this cooperative method and state-of-the-art enumerative algorithms in section 4, we conclude in section 5.

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_i from D. Variables are subject to constraints from C. Each constraint c involves a set $X_c = \{x_{c_1}, \ldots, x_{c_k}\}$ of variables. A relation R_c (from R) is associated with each constraint c such that R_c represents the set of allowed k-uplets over $D_{c_1} \times \cdots \times D_{c_k}$.
A CSP is called *binary* if each constraint involves two variables. Let x_i and x_j be two variables, we note c_{ij} the corresponding constraint. Afterwards, we consider only binary CSPs. However, our ideas can be extended to n-ary CSPs.
Given $Y \subseteq X$ such that $Y = \{x_1, \ldots, x_k\}$, an *instantiation* of variables from Y is a k-uplet $(v_1, \ldots, v_k)$ from $D_1 \times \cdots \times D_k$. It is called *consistent* if $\forall c \in C, X_c \subseteq Y, (v_1, \ldots, v_k)[X_c] \in R_c$, *inconsistent* otherwise. We use indifferently the term *assignment* instead of instantiation. We note the instantiation $(v_1, \ldots, v_k)$ in the more meaningful form $\{x_1 \leftarrow v_1, \ldots, x_k \leftarrow v_k\}$. 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.
Given a CSP $\mathcal{P} = (X, D, C, R)$ and 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:

(i) $\forall j, 1 \leq j \leq i, D_j(\mathcal{A}_i) = \{v_j\}$

(ii) $\forall j, i < j \leq n, D_j(\mathcal{A}_i) = \{v_j \in D_j | \forall c_{kj} \in C, 1 \leq k \leq i, (v_k, v_j) \in R_{c_{kj}}\}$

(iii) $\forall j, j', R_{c_{jj'}}(\mathcal{A}_i) = R_{c_{jj'}} \cap (D_j(\mathcal{A}_i) \times D_{j'}(\mathcal{A}_i))$.

$\mathcal{A}_i$ is said *FC-consistent* if $\forall j, D_j(\mathcal{A}_i) \neq \emptyset$. According to the definition of an induced CSP, $D_j(\mathcal{A}_i)$ 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_j is the initial domain of x_j. Likewise, $R_{c_{jj'}}(\mathcal{A}_i)$ corresponds to the initial relation $R_{jj'}$ projected on 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 and the algorithm Forward-Checking with Nogood Recording (noted FC-NR [14]).
A nogood corresponds to an assignment which can't be extended to a solution. More formally ([14]), given an instantiation $\mathcal{A}$ and a subset J 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 $\mathcal{A}$. We note $\mathcal{A}[Y]$ the restriction of $\mathcal{A}$ to variables which are both in $X_\mathcal{A}$ and in Y. For instance, every inconsistent assignment corresponds to a nogood. The converse doesn't hold.
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 [14]) to compute justifications. 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_j for $\mathcal{A}_i$ if one of the following conditions holds:

1. c_{kj} is a value-killer of v_j for $\mathcal{A}_{i-1}$
2. $k = i$ and $(v_k, v_j) \notin R_{c_{kj}}(\mathcal{A}_i)$ and $v_j \in D_j(\mathcal{A}_{i-1})$
3. $\{x_k \leftarrow v_k, x_j \leftarrow v_j\} \in N$

The set of value-killers of a domain D_j is defined as the set of constraints which are a value-killer for at least one value v_j of D_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_i is wiped-out, the set of value-killers of D_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_i be an unassigned variable. Let K be the set of value-killers of D_i. If it doesn't remain any value in $D_i(\mathcal{A})$, 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 ([14]) *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_i be an unassigned variable. Let K be the set of value-killers of D_i. Let $\mathcal{A}_j$ be the extension of $\mathcal{A}$ by assigning the value v_j to x_i ($\mathcal{A}_j = \mathcal{A} \cup \{x_i \leftarrow v_j\}$). If $(\mathcal{A}_1, J_1), \ldots, (\mathcal{A}_d, J_d)$ are nogoods, then $(\mathcal{A}, K \cup \bigcup_{j=1}^{d} J_j)$ is a 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) which forbids the tuple $\mathcal{A}$. This constraint 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 ([12]). 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.*

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.
FC-NR explores the search tree like Forward Checking. During the search, it takes advantage of dead-ends to create and record nogoods. These nogoods are then used as described above to prune the search tree. The main drawback of FC-NR is that the number of nogoods is potentially exponential. So, we limit the number of nogoods by recording nogoods whose arity is at most 2 (i.e. unary or binary nogoods), according to the proposition of Schiex and Verfaillie ([14]). Nevertheless, the ideas we present can be easily extended to n-ary nogoods.

3 The cooperative scheme

3.1 Description

The approach we are going to describe assumes that:

- the shared memory is big enough to contain the CSP instance, the set of recorded nogoods and the current instantiation of each solver,
- each process can communicate, by exchanging messages, with any other process.

The cooperative scheme (illustrated in figure 1) consists of p sequential solvers which run independently FC-NR on the same CSP. Each solver differs from another one in ordering variables and/or values with different heuristics. Thus, each one has a different search tree. The cooperation consists in exchanging nogoods. A solver can use nogoods produced by other solvers in order to prune a part of its search tree, which speeds up the resolution.
During the search, when a solver finds a nogood, it informs the manager, which communicates at once this new information to a part of other solvers. In this way, a solver sends only one message (instead of $p-1$ in a scheme without manager) and gets back quickly to the resolution of the problem. The paragraph 3.2 describes more precisely the manager of nogoods.
Finally, the produced nogoods are added to the initial problem by creating new constraints (or by tightening existing ones). Each solver, thanks to the FC-NR algorithm, can use these nogoods for backjumping or for filtering. By so doing, we don't fully exploit the recorded nogoods. Indeed, assume that a solver may have started visiting an area of the search tree before receiving a nogood which forbids the exploration of this area. Then, in such a case, the solver finishes its exploration as though it has never received the nogood. So, we add to FC-NR a phase of interpretation, which, thanks to received nogoods, limits the size of the search tree by stopping developing some branches which lead to failure. In the paragraph 3.3, we present an improved version of our phase of interpretation.

3.2 The manager of nogoods

The manager's task is to update the set of nogoods and to communicate new nogoods to solvers. Update the set of nogoods 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. Each nogood communicated to manager is added to this set.
In order to limit the cost of communications, the manager must inform only solvers for which nogoods may be useful. A nogood is said *useful* for a solver if it allows this solver to limit the size of its search tree. Among the useful nogoods for a solver, we can distinguish ones which are useful as soon as they are received and ones which are useful later via the filtering. For the last ones, note that they are exploited via the constraints added or tightened

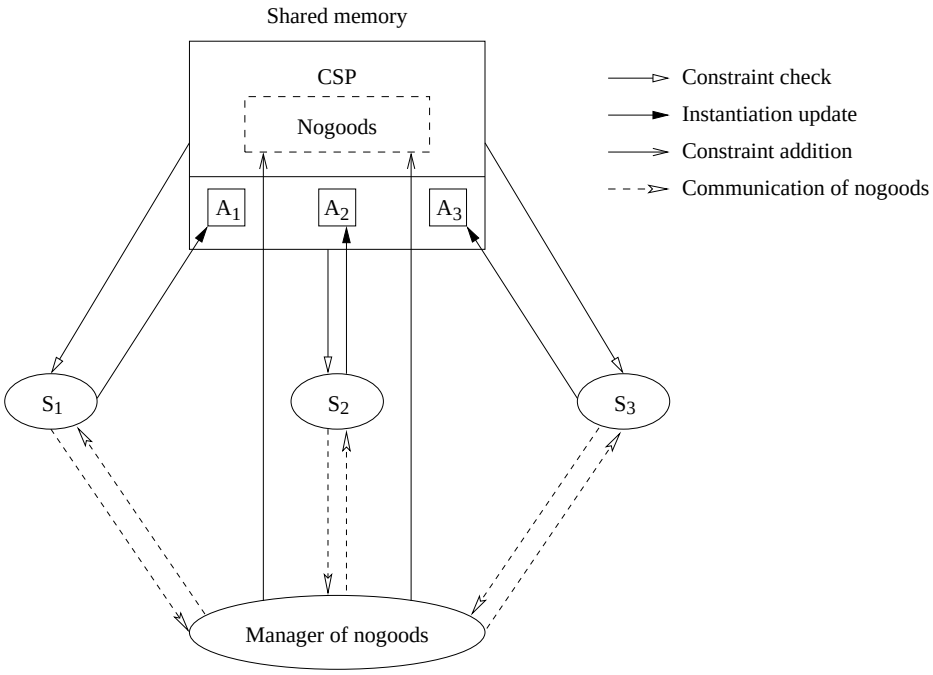

Figure 1: The cooperative scheme based on exchanging nogoods for three solvers. Solid arrows represent accesses to the shared memory, dashed ones communications between processes.

(i.e. thanks to the shared memory). So we only need to communicate the nogoods which are useful as soon as they are received. The next theorem characterizes the usefulness of these nogoods according to their arity and the current instantiation of the considered solver.

Theorem 4 *Let S be a solver and $\mathcal{A}_S$ be its current instantiation.*

(a) a unary nogood is always useful for S,

(b) a binary nogood $(\{x_i \leftarrow a, x_j \leftarrow b\}, J)$ is useful for S if x_i and x_j are assigned respectively with a and b in $\mathcal{A}_S$,

(c) a binary nogood $(\{x_i \leftarrow a, x_j \leftarrow b\}, J)$ is useful for S if, in $\mathcal{A}_S$ x_i (resp. x_j) is assigned with a (resp. b) and $b \in D_j(\mathcal{A}_S)$ (resp. $a \in D_i(\mathcal{A}_S)$).

From this theorem, when the manager of nogoods receives a nogood, it conveys it to some solvers (according to its usefulness for these solvers):

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

(b) the binary nogoods $(\{x_i \leftarrow a, x_j \leftarrow b\}, J)$ 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 manager sends to a solver the nogoods 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.
Finally, with a view to limit the cost of communications, only the instantiation $\mathcal{A}$ of nogood $(\mathcal{A}, J)$ is conveyed. Communicate the justification isn't necessary because this nogood is added to the problem in the form of a constraint c. Thanks to received information, solvers can forbid $\mathcal{A}$ with justification c.
In [15], we show the contribution of the manager of nogoods to our scheme with respect to a version without manager. Theoretically, in the overall search, our scheme may exchange N additional messages (where N is the

number of produced nogoods). Nevertheless, in practice, we observe that our scheme exchanges significantly fewer nogoods than a scheme without manager because the manager doesn't send each nogood to every solver.

3.3 Phase of interpretation

Before describing the phase of interpretation, we introduce the following notation: if $\mathcal{A}$ is an instantiation, we note $\mathcal{A}_{x_k}$ the restriction of $\mathcal{A}$ to variables assigned before x_k, x_k included.
The phase of interpretation is applied whenever a nogood is received. 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.
In [15], we present a first version of such a phase, but its integration to the FC-NR algorithm entails the loss of a fundamental property of FC-NR, namely that every instantiation built by FC-NR is FC-consistent. This problem happens when a solver forbids some values thanks to some nogoods while these values have been already removed thanks to the filtering. In such a case, if a domain is wiped out due to a nogood, the next instantiation built by the solver could be FC-inconsistent. Indeed, the solver may not restore some values deleted by the filtering because now they are forbidden by some nogoods and then the wiped-out domain remains empty even if the solver has just backtracked. In order to avoid the problem occurring, in our previous work, we add a backjump phase and we don't take into account all the nogoods which can be produced. Thus, the solvers don't fully exploit the nogoods they produce or receive. So, in order to correct this drawback, we now describe an improved phase of interpretation.
For unary nogoods, the phase of interpretation corresponds to a permanent deletion of a value and to a possible backjump. Method 1 details the 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_i.
We delete a from D_i. Furthermore:

(a) If x_i isn't assigned and $D_i(\mathcal{A})$ 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 $\mathcal{A}'$ be the obtained instantiation.
If $D_i(\mathcal{A}')$ 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_i, which implies that, for any instantiation $\mathcal{A}$, the value a doesn't belong to $D_i(\mathcal{A})$. The backjump phases in the cases (a)(ii) et (b)(ii) allow to pursue the exploration of the search tree with a FC-consistent instantiation (unlike the phase of interpretation in [15]). In other words, these backjump phases guarantee that there is no empty current domain, before extending the current instantiation.

Theorem 5 *The method 1 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 $D_i(\mathcal{A})$. If $D_i(\mathcal{A})$ 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 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_i, $D_i(\mathcal{A}')$ 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.□

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 $D_j(\mathcal{A}_{x_i})$ (resp. $D_i(\mathcal{A}_{x_j})$).
If $D_j(A)$ (resp. $D_i(A)$) is empty:

- *(i) we record the nogood $(\mathcal{A}[X_K], K)$ with K the set of value-killers of D_j (resp. D_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 $D_j(\mathcal{A}_{x_i})$ (resp. $D_i(\mathcal{A}_{x_j})$).
If $D_j(\mathcal{A}')$ (resp. $D_i(\mathcal{A}')$) is empty:

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

Theorem 6 *The method 2 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 $D_j(\mathcal{A}_{x_i})$ by filtering.
If $D_j(\mathcal{A})$ is wiped out, we reason like in the proof of theorem 5 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.□

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 [15]. Remark that, here, the deletions aren't permanent unlike the phase of interpretation for unary nogoods.

4 Experimental results

In this section, we provide experimental comparisons between our cooperative scheme and some state-of-the-art sequential algorithms, namely Forward-Checking (noted FC [7]), Forward-Checking with Conflict-directed Backjumping (noted FC-CBJ [12]), Forward-Checking with Nogood Recording (noted FC-NR [14]), and Maintaining Arc-Consistency (noted MAC [13]). About MAC, we achieve arc-consistency thanks to the AC-2001 algorithm ([2]). We experiment two kinds of benchmarks:

- random instances,
- real-world instances.

4.1 Heuristics

For each classical algorithm, we use the dom/deg heuristic ([1]), for which the next variable to assign is one which minimizes the ratio $\frac{|D_i|}{|\Gamma_i|}$ (where D_i is the current domain of x_i and Γ_i is the set of variables which are connected to x_i by a binary constraint). This heuristic is considered better, in general, than other classical heuristics. That's why we choose it.
In our implementation, only the size of domains varies for each instantiation. The degree $|\Gamma_i|$ is updated when a new constraint is added thanks to a nogood. Likewise, the tightness of a constraint is modified when the constraint is tightened.
For the cooperative method, in order to guarantee distinct search trees, each solver orders variables and/or values with different heuristics. As there exist few efficient heuristics for choosing variables, from the dom/deg heuristic, we produce several different orders:

- by defining the heuristic dom/st for which the next variable to assign is one which minimizes the ratio $\frac{|D_i|}{S_i}$ (with S_i the sum of tightness of constraints involving x_i),
- by choosing differently the first variable and then applying either dom/deg or dom/st.

As regards the choice of next value to assign, we consider values in appearance order for classical algorithms, and in appearance order or in reverse order for the cooperative search. In following results, half solvers of the cooperative method use the appearance order to order domains, other half reverse order.

4.2 Results for random instances

In this paragraph, we compare our cooperative scheme with some classical algorithms for random CSPs. 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, C and T. It builds a CSP of class (N, D, C, T) with N variables which have domains of size D and C binary constraints ($0 \leq C \leq \frac{N(N-1)}{2}$) 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's threshold. Every problem we consider has a connected graph of constraints.
Given results are the averages of results obtained on 100 problems per class. For the cooperative method, each problem is solved 15 times in order to reduce the impact of non-determinism of solvers on results. Results of a problem are then the averages of results of 15 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

[1] downloadable at http://www.lirmm.fr/∼bessiere/generator.html

if, in practice, these experimentations are realized on a Linux-based PC with a single AMD Athlon XP 1800+ processor.
Tables 1 and 2 respectively provide the number of constraint checks (in thousands) and the run-time for our cooperative scheme with respectively one, two or four solvers and for FC, FC-CBJ, FC-NR and MAC.

Class	cooperative solver			FC	FC-CBJ	FC-NR	MAC
	$p = 1$	$p = 2$	$p = 4$				
$(50, 15, 184, 112)$	11,246	4,885	2,566	7,347	7,089	9,984	4,750
$(50, 15, 245, 93)$	85,394	41,487	19,506	64,695	63,629	81,947	53,354
$(50, 25, 123, 439)$	5,122	2,877	1,273	5,969	5,208	5,479	2,601
$(50, 25, 150, 397)$	54,249	18,163	9,888	34,743	32,648	52,620	21,813

Table 1: Number of constraint checks (in thousands) for random CSPs.

For one solver, we observe that, in some cases, our cooperative method performs more constraint checks than FC, FC-CBJ or MAC. These additional constraint checks are explained by the FC-NR algorithm itself. Indeed, in FC-NR (and so in the cooperative method), checking constraints means checking initial and added constraints. With respect to FC-NR, the cooperative method checks slightly more constraints. Such a result may be due at least in part to exchanges of nogoods with the manager. In effect, unlike the classical FC-NR algorithm, in our scheme, nogoods aren't added as new constraints as soon as they are produced. They are first sent to the manager. So, in the meantime, the cooperative solver doesn't exploit the corresponding new constraints because they don't exist yet, what permits the cooperative method either to save some constraint checks with respect to FC-NR, or, in the contrary, to visit some nodes which aren't developed by FC-NR.
About the run-time, the cooperative solver is less fast than the classical algorithms. This result is due to the number of additional constraint checks and to the cost of communications. About the cost of communications, note that, even if there is a single solver, the solver sends the produced nogoods to the manager and checks whether it receives a message.

Class	cooperative solver			FC	FC-CBJ	FC-NR	MAC
	$p = 1$	$p = 2$	$p = 4$				
$(50, 15, 184, 112)$	3,386	1,456	795	1,311	1,569	1,845	1,612
$(50, 15, 245, 93)$	25,908	12,599	6,259	10,770	13,088	16,054	19,863
$(50, 25, 123, 439)$	1,678	896	412	1,177	1,250	978	677
$(50, 25, 150, 397)$	16,014	5,285	3,049	6,614	7,576	8,823	6,487

Table 2: Run-time in milliseconds for random CSPs.

For two solvers, the cooperative search checks significantly fewer constraints than FC, FC-CBJ and FC-NR, and either slightly more or significantly fewer than MAC depending on the class we consider. With regard to the run-time, the cooperative search solves each class faster than FC-CBJ and FC-NR. However, it is either better or worse than FC or MAC. Nevertheless, when it is better, the saved time is significant.
For at least four solvers, we note that our cooperative scheme performs fewer constraint checks than the four classical algorithms we use. In each case, the observed gains are significant. Then, we notice that the cooperative search is faster than the classical algorithms. For information, if we run our cooperative search with more than four solvers, the obtained run-time and number of constraint checks are less important than ones for four solvers.
If we only consider consistent (respectively inconsistent) instances, we observe similar results as shown in table 3 (resp. 4).

Class	cooperative solver			FC	FC-CBJ	FC-NR	MAC
	$p = 1$	$p = 2$	$p = 4$				
$(50, 15, 184, 112)$	2,666	1,122	610	1,009	1,210	1,413	1,176
$(50, 15, 245, 93)$	16,384	8,397	3,150	7,229	8,797	10,427	12,971
$(50, 25, 123, 439)$	2,009	1,110	448	1,348	1,430	1,115	616
$(50, 25, 150, 397)$	16,014	4,293	2,656	5,913	6,782	7,958	5,204

Table 3: Run-time in milliseconds for consistent random CSPs.

Class	cooperative solver			FC	FC-CBJ	FC-NR	MAC
	$p = 1$	$p = 2$	$p = 4$				
$(50, 15, 184, 112)$	4,562	2,002	1,097	1,803	2,155	2,550	2,324
$(50, 15, 245, 93)$	31,264	14,962	8,008	12,762	15,502	19,220	23,740
$(50, 25, 123, 439)$	1,466	759	390	1,067	1,135	891	715
$(50, 25, 150, 397)$	16,014	6,547	3,551	7,505	8,587	9,924	8,121

Table 4: Run-time in milliseconds for inconsistent random CSPs.

4.3 Results for real-world instances

We now compare the considered algorithms on some real-world instances of the CELAR from the FullRLFAP archive[2]. These problems correspond to radio link frequency assignment problems. From this archive, we only keep the instances which MAC solves in at least 500 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 Linux-based PC with a single Intel Pentium III 550 MHz 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.
Tables 5 and 6 respectively provide the number of constraint checks (in thousands) and the run-time for our cooperative method with respectively one, two or four solvers and for FC, FC-CBJ, FC-NR and MAC. In these tables, the symbol "-" means that the search was stopped.

Problem	cooperative method			FC	FC-CBJ	FC-NR	MAC
	$p = 1$	$p = 2$	$p = 4$				
SCEN-01	176	176	176	176	176	176	1,858
SCEN-05	31,352	31,148	581	-	39,022	31,251	9,221
SCEN-11	5,460	5,460	262	32,095	5,460	5,460	22,521
GRAPH-09	191	191	191	191	191	191	1,820
GRAPH-10	1,498	1,497	1,497	-	848	1,498	2,531
GRAPH-14	174	174	174	174	174	174	1,599

Table 5: Number of constraint checks (in thousands) for some real-world instances from the FullRLFAP archive.

For SCEN-01, GRAPH-09 and GRAPH-11 instances, the cooperative method and the classical algorithms based on Forward-Checking obtain comparable results, which are slightly better than MAC's ones. For GRAPH-10 instance, we note that FC is unable to solve the problem unlike the other methods (included the cooperative one).

[2] we thank the Centre d'Electronique de l'Armement (France).

Problem	cooperative method			FC	FC-CBJ	FC-NR	MAC
	$p = 1$	$p = 2$	$p = 4$				
SCEN-01	110	110	110	100	120	110	610
SCEN-05	33,650	33,510	570	-	27,350	24,050	15,380
SCEN-11	3,450	3,450	160	23,930	3,910	3,290	25,520
GRAPH-09	150	140	140	120	90	130	640
GRAPH-10	1,320	1,320	1,320	-	520	900	890
GRAPH-14	130	120	110	100	100	120	510

Table 6: Run-time in milliseconds for some real-world instances from the FullRLFAP archive.

The cooperative search performs as many constraint checks as FC-NR. However, in time, the classical algorithms (except FC) are better. More generally, for these four instances, the cooperative search doesn't seem taking advantage of cooperation and multiple orders for variables and values.
In contrast, for SCEN-05 and SCEN-11 instances, the cooperative method with four solvers performs fewer constraint checks and so is significantly faster than the classical algorithms.

5 Conclusions and discussion

We have presented, in a previous work ([15]), a cooperative parallel search for solving the constraint satisfaction problem. We run independently p solvers based on Forward-Checking with Nogood Recording. The solvers exchange nogoods via a process ("the manager of nogoods") which regulates the exchanges. Solvers exploit the nogoods they receive to limit the size of their search tree thanks to the phase of interpretation. In order to prevent FC-NR from loosing one of its fundamental properties, the phase described in [15] doesn't take into account all the nogoods. So, in this paper, we have first presented an improved version of the phase of interpretation in order to exploit every nogood.
Experimentally, we have shown the interest of our approach from a parallel viewpoint, namely we have obtained linear or superlinear speed-up [15]. However, we haven't studied its behavior with respect to classical algorithms. In this paper, we provide experimental comparisons between the cooperative method and some state-of-the-art algorithms. In particular, we observe that the cooperative search with at least four solvers (even in some cases from two solvers) is faster than classical algorithms like FC or MAC, if we use one processor per solver. If we only have a processor, we can simulate the parallelism and obtain, in some cases, better results than classical algorithms, like for the SCEN-05 or SCEN-11 instances.

Bibliography

1. C. Bessière and J.-C. Régin. MAC and Combined Heuristics : Two Reasons to Forsake FC (and CBJ?) on Hard Problems. In *Proc. of 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*, 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. S. Clearwater, B. Huberman, and T. Hogg. Cooperative Solution of Constraint Satisfaction Problems. *Science*, 254:1181–1183, Nov. 1991.
5. R. Dechter. Enhancement Schemes for Constraint Processing : Backjumping, Learning, and Cutset Decomposition. *Artificial Intelligence*, 41:273–312, 1990.
6. J. Gaschnig. Performance Measurement and Analysis of Certain Search Algorithms. Technical Report, 1979.
7. R. Haralick and G. Elliot. Increasing tree search efficiency for constraint satisfaction problems. *Artificial*

Intelligence, 14:263–313, 1980.
8. 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.
9. T. Hogg and C.P. Williams. Solving the Really Hard Problems with Cooperative Search. In *Proceedings of AAAI 93*, pages 231–236, 1993.
10. T. Hogg and C.P. Williams. Expected Gains from Parallelizing Constraint Solving for Hard Problems. In *Proceedings of AAAI 94*, pages 331–336, Seattle, WA, 1994.
11. D. Martinez and G. Verfaillie. Echange de Nogoods pour la résolution coopérative de problèmes de satisfaction de contraintes. In *CNPC 96*, pages 261–274, 1996.
12. P. Prosser. Hybrid Algorithms for the constraint satisfaction problem. *Computational Intelligence*, 9:268–299, 1993.
13. D. Sabin and E. Freuder. Contradicting Conventional Wisdom in Constraint Satisfaction. In *Proceedings of eleventh ECAI*, pages 125–129, 1994.
14. T. Schiex and G. Verfaillie. Nogood Recording for Static and Dynamic Constraint Satisfaction Problems. In *Proc. of the* 5^{th} *IEEE ICTAI*, 1993.
15. C. Terrioux. Cooperative Search and Nogood Recording. In *Proceedings of the 17th International Joint Conference on Artificial Intelligence*, pages 260–265, 2001.