

A CSP solver focusing on FAC variables

Éric Grégoire, Jean-Marie Lagniez, Bertrand Mazure

▶ **To cite this version:**

Éric Grégoire, Jean-Marie Lagniez, Bertrand Mazure. A CSP solver focusing on FAC variables. 17th International Conference on Principles and Practice of Constraint Programming (CP'11), 2011, Perugla, Italy. pp.493-507. hal-00865539

HAL Id: hal-00865539
https://hal.science/hal-00865539

Submitted on 24 Sep 2013

HAL is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L'archive ouverte pluridisciplinaire **HAL**, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d'enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.

A CSP solver focusing on FAC variables *

Éric Grégoire, Jean-Marie Lagniez, and Bertrand Mazure

Université Lille-Nord de France
CRIL - CNRS UMR 8188
Artois, F-62307 Lens
{gregoire,lagniez,mazure}@cril.fr

Abstract. The contribution of this paper is twofold. On the one hand, it introduces a concept of FAC variables in discrete Constraint Satisfaction Problems (CSPs). FAC variables can be discovered by local search techniques and powerfully exploited by MAC-based methods. On the other hand, a novel synergetic combination schema between local search paradigms, generalized arc-consistency and MAC-based algorithms is presented. By orchestrating a multiple-way flow of information between these various fully integrated search components, it often proves more competitive than the usual techniques on most classes of instances.

1 Introduction

These last decades, many research efforts have been devoted in the Artificial Intelligence community to the design of general algorithms and solvers for discrete Constraint Satisfaction Problems (in short, CSPs). Tracing back to the seminal work on simulated annealing by Kirkpatrick *et al.* [17], stochastic local-search approaches (SLS) were investigated successfully in early pioneering works, mainly based on the so-called *min-conflicts* heuristic developed by Minton *et al.* [24]. They were considered powerful paradigms for CSPs -and their specific SAT case- in light of the results by e.g. Gu [12], Selman *et al.* [27] and Cheeseman *et al.* [4].

However, apart from the specific SAT domain and with only a few exceptions (e.g. [10], [16], [7], [8]), the current mainstream approaches to general CSPs solving rely on complete methods that do not include SLS components as main tools (*e.g.*, Abscon [23, 19], Choco [29], Mistral [15], Sugar [28], etc.). One reason lies in the fact that SLS is not an exhaustive search paradigm and does not allow by itself to prove the absence of any solution for a CSP. Moreover, SLS often entails significant computations and search-space explorations that advanced complete techniques are expected to attempt to avoid, at least partially. Finally, it is sometimes (but wrongly) believed that SLS should merely be devoted to situations where solutions are densely distributed throughout the state space, justifying some possible random aspects in the search.

* Part of this work was supported by the French Ministry of Higher Education and Research, Nord/Pas-de-Calais Regional Council and E.C. FEDER program through the 'Contrat de Projets État/Région (CPER) 2007-2013' and by the French National Research Agency (ANR) through the UNLOC and TUPLES projects.

On the contrary, this paper shows that complete and SLS techniques for solving CSPs can benefit one another. More precisely, it presents a synergetic combination of local search and elements of complete techniques that often outperforms the usual complete, SLS, or basic hybrid approaches involving (generalized) arc-consistency and SLS, in the following sense. This method is not only complete, it is also robust in the sense that it solves both satisfiable or unsatisfiable (structured or random) CSPs instances quite indifferently. Actually, our comprehensive experimental studies show that it solves more instances than the currently existing techniques.

One key issue is that the SLS computation that is guided as much as possible towards the most difficult subparts of the CSP can provide powerful oracles and information when some further steps of a complete search are required. Although this latter idea was already exploited in some previous works in the SAT domain [21], it is refined here thanks to an original concept of FAC variables. FAC variables of a CSP, as **F**alsified in **A**ll **C**onstraints, are variables occurring in all falsified constraints under some intrepretation, and thus in at least one constraint per minimal core (also called MUC, for Minimal Unsatisfiable Core) of the CSP when such cores exist. Interestingly, SLS often allows FAC variables to be detected efficiently and complete MAC-based techniques focusing first on FAC variables can have their efficiency boosted on many instances. Likewise, e.g. powerful heuristics (especially the $dom/wdeg$ [3]) developed within complete CSP techniques can play an essential role in the SLS computation. Actually, the proposed method, called FAC-SOLVER, is an elaborate imbrication of SLS and steps of complete techniques that orchestrates a multiple-way flow of information between various fully integrated search components.

The paper is organized as follows. In the next Section, some basic technical background about CSPs is provided. Then, the FAC variable concept is presented. In Section 4, the architecture of the FAC-SOLVER method is presented globally, before each component is detailed. Comprehensive experimental studies are discussed in Section 5. In the conclusion, the focus is on perspectives and promising paths for future research.

2 CSPs Technical Background

A CSP or *Constraint Network* $\mathcal{CN}$ is a pair $\langle \mathcal{X}, \mathcal{C} \rangle$ where $\mathcal{X}$ is a finite set of n variables s.t. each variable X of $\mathcal{X}$ is associated with a finite set $dom(X)$ of candidate values for X. $\mathcal{C}$ is a finite set of m constraints on variables from $\mathcal{X}$ s.t. each constraint C in $\mathcal{C}$ is associated with one relation $rel(C)$ indicating the set of tuples of authorized values for the variables occurring in C. An assignment $\mathcal{I}$ of $\mathcal{CN}$ associates a value $\mathcal{I}(X) \in dom(X)$ to every variable $X \in \mathcal{X}$. We note $false(\mathcal{X}, \mathcal{C}, \mathcal{I})$ the set of variables that appear in at least one falsified constraint under the assignment $\mathcal{I}$. $\langle \mathcal{X}, \mathcal{C} \rangle|_{X=v}$ is the resulting CSP obtained from the CSP $\langle \mathcal{X}, \mathcal{C} \rangle$ by reducing $dom(X)$ to the singleton $\{v\}$ while $\langle \mathcal{X}, \mathcal{C} \rangle|_{X \neq v}$ is obtained by deleting the v value in $dom(X)$. We say that the assignment $\mathcal{I}$ is a local minimum for $\mathcal{CN}$ when no single change of value of any variable leads to a decrease of the total number of falsified constraints of $\mathcal{CN}$.

Solving a constraint network $\mathcal{CN}$ consists in checking whether $\mathcal{CN}$ admits at least one assignment that satisfies all constraints of $\mathcal{CN}$ and in delivering such an assignment in the positive case.

In the following, we consider both binary and non-binary constraints. Most current complete approaches to solve constraints networks are based on algorithms implementing *maintaining arc consistency* techniques (in short, MAC) [25]. Roughly, these techniques perform a depth-first search procedure with backtracking, while maintaining some forms of local (Generalized Arc) Consistency (in short GAC and AC), which are filtering techniques expelling detected forbidden values (see e.g. [20, 2, 18]).

3 FAC variables

One key factor of the efficiency of the FAC-SOLVER approach relies on the following FAC *(Falsified in All Constraints) variable* concept.

Definition 1. *Let $\mathcal{CN}$ be a constraint network under an assignment $\mathcal{I}$. A* FAC *variable is a variable occurring in every falsified constraint of $\mathcal{CN}$ under $\mathcal{I}$.*

This concept can be related to the notion of *boundary* point introduced by Goldberg in the SAT domain [11]. For a CNF formula, a variable is boundary under an assignment of all propositional variables if this variable belongs to all clauses that are falsified by the assignment. This definition is similar to the FAC one, but we have adopted an alternative name for a simple reason: in the CSP context, this kind of variables is not at the so-called "boundary", *i.e.*, a situation where it is sufficient to inverse the truth value of a boundary variable to satisfy all falsified clauses. A FAC variable in the SAT domain thus draws a boundary line between satisfiabiliy and unsatisfiability of a part of the formula. In CSP, changing the value of a FAC variable does not ensure that constraints become satisfied. Accordingly, the notion of boundary as underlied by Goldberg cannot be applied in theCSP domain. For this reason we have decided to not use the same name. Nevertheless some interesting properties of boundary are preserved which can help understand the possible role of FAC variables for solving unsatisfiable CSPs.

Property 1. Any FAC variable X of $\mathcal{CN}$ occurs in at least one constraint per MUC of $\mathcal{CN}$ when $\mathcal{CN}$ is unsatisfiable.

Indeed, under any assignment $\mathcal{I}$, any MUC contains at least one falsified constraint. Thus, if a variable occurs within all constraints that are falsified under $\mathcal{I}$, it occurs within at least one constraint per MUC.

Property 2. Unsatisfiable CSPs that exhibit at least two MUCs sharing no variable do not possess any FAC variable.

FAC variables can play a key role in the inconsistency of a CSP since they are involved in all of its unresolvable minimal sets of conflicting constraints. Accordingly, focusing a MAC-based search component first on FAC variables (when they exist) might thus help.

In the worst case, checking whether a constraint belongs to at least one MUC, belongs to the Σ_2^p complexity class [9]. Moreover, a CSP can possess an exponential number of MUCs. Thus, detecting FAC variables by first computing all MUCs is untractable in the worst case. On the contrary, SLS provides a heuristic way to detect FAC

variables at low cost. One direct but inefficient way to detect some of them would consist in looking for FAC variables for each assignment crossed by SLS. For efficiency reasons, we will look for FAC variables only for assignments that are local minima w.r.t. the number of currently falsified constraints of the CSP.

Satisfiable CSPs can also exhibit FAC variables. Interestingly, it appears that FAC variables can also be expected to play a positive role for solving those CSPs. Indeed, to some extent, these variables can also be expected to take part in the difficult part of those CSPs since they are involved in all falsified constraints under at least one assignment. Accordingly, it could be also useful to focus on them during a complete search.

Finally, it must be noted that when some variables are instantiated, a new CSP is actually created. FAC variables w.r.t. this new CSP can exist; they are not necessarily FAC variables w.r.t. the initial CSP.

In the FAC-SOLVER method, all FAC variables that can be detected when local minima are reached during a SLS will be collected. When a MAC-based component must be run thereafter, it will focus first on the FAC variables in hope for an improved efficiency.

Using FAC variables in a further systematic search component appears to be a refinement of some heuristics e.g. used in the SAT framework and involving hybrid SLS-DPLL algorithms. For example, [21] advocates to select the next variables to be assigned in a DPLL-based search for satisfiability among the variables belonging to the most often falsified clauses during a preliminary failed SLS, as those variables probably belong to minimal cores of the instance. Also [13] recommends the use of critical clauses, *i.e.*, falsified clauses during a failed SLS that are such that any flip of a variable causes at least another clause to be falsified. Critical clauses were also shown to often belong to minimal cores. Branching on variables occurring in them appeared to boost the further complete search process [1]. FAC variables do not only occur in one minimal core but in all of them. Branching on them might thus increase the efficiency of the search process even more significantly. The FAC-SOLVER method described in the next Section was intended to implement and check these ideas on a large panel of instances.

4 FAC-SOLVER Approach

The FAC-SOLVER approach deeply integrates three search components in a novel synergetic way: a SLS, a MAC and an hybrid solver, which is itself mixing SLS and GAC. These components interact in several ways and share all information about the current global search process. The global architecture of FAC-SOLVER is described in Figure 1. Roughly, the process starts with a call to one SLS solver, then in the case of failure a hybrid solver is run followed by a limited MAC. Calls between different components depend on dynamical threshold values for two variables that play a strategic role, namely $SLSprogress$ and $\#conflicts$.

Algorithm 1 describes the FAC solver. First, a call to GAC ensures arc-consistency (or delivers a final inconsistency proof) and leads to some possible filterings (lines 4-5). Next, while a solution is not found or the problem is not proved inconsistent, the solver sequentially performs the three components described in the next sections. The number of conflicts $maxConf$ controls the restart associated to the Hybrid and MAC parts. It is initialized to 10 (line 2) and is geometrically increased at each iteration step of the

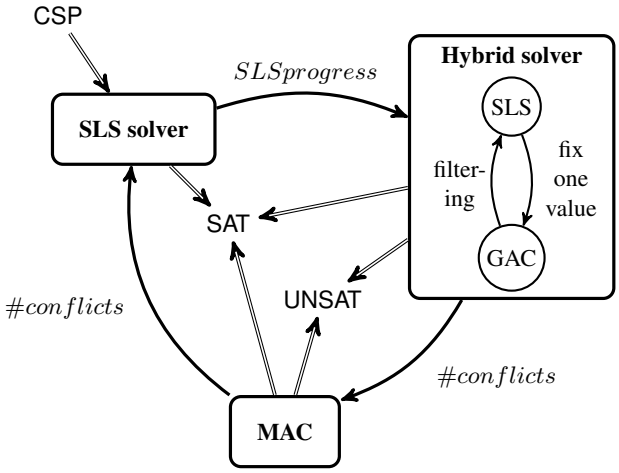


Fig. 1. Interactions between FAC-SOLVER basic search components.

main loop (line 16). The complete assignment used by the local search is initialized randomly. The CSP $\langle \mathcal{X}, \mathcal{C} \rangle$ is shared by all components. It is simplified by successive assignments and refutations. At each new iteration, this CSP is reinitialized (only the filtering computed at level 0 are kept (line 9)). The $SLSprogress$ variable controls the duration of the SLS component run. It is initialized to $maxConf \times 8$. This variable will be increased and decreased by the SLS component. All components are detailed in the next sections.

4.1 The SLS Component

Let us detail the SLS procedure first which is described in a simplified way in the algorithm named Procedure `SLS`. It is a random-walk local search procedure *à la walksat* [26] with a *novelty* escape strategy [22]. In parallel, this SLS also tries to detect FAC variables each time a local minimum is reached. The variable controlling the progress of the SLS is $SLSprogress$, which is increased in two situations: when the number of falsified constraints reaches a new minimum value (line 13) and when FAC variables are discovered (line 5). It is decreased when no FAC variable is discovered in a local minimum (line 6). This variable is intialized to 10.000 if the CSP is binary and to 1.000 otherwise. This way to estimate the progress of the SLS is inspired from the adaptative noise introduced by Hoos *et al.* in [14]. When the SLS fails to prove the consistency of the CSP but seems rather stuck in its exploration, the $SLSprogress < 0$ test (line 7) allows the so-called Hybrid component to be activated, which will exploit in its turn all the information collected so-far. Intuitively, in addition to looking for an assignment satisfying the CSP, the SLS solver collects information about FAC variables. Due to the larger increment of the $SLSprogress$ control variable when FAC variables are discovered, it focuses its exploration on assignments that are close to local minima involving FAC variables.

Algorithm 1: FAC-solver

Data: A CSP $\langle \mathcal{X}, \mathcal{C} \rangle$
Result: $true$ if the CSP is satisfiable, $false$ otherwise

```
1  result ⟵ unknown ;
2  maxConf ⟵ 10 ;
3  S_e ⟵ ∅ ;          //Set of FAC var. found with their FAC values
4  GAC() ;
5  if ∃X ∈ 𝒳 s.t. dom(X) = ∅ then return false ;
6  𝒜 ⟵ a random assignment of 𝒳 ;
7  while (result = unknown) do
8      initialize SLSprogress variable ;
9      Backjump(0) ;                          // backjump to level 0
10     SLS() ;
11     if (result ≠ unknown) then return true ;
12     Hybrid() ;
13     if (result ≠ unknown) then return result ;
14     MAC() ;
15     if (result ≠ unknown) then return result ;
16     maxConf ⟵ maxConf × 1.5 ;
```

The SLS procedure is also used by the hybrid component. When this procedure is called by the hybrid component, the SLS works on a sub-CSP that is downsized by the various calls to the `FIX` procedure which is described in the next section. At the opposite, during the initial local search, SLS is handling the full CSP.

4.2 Hybrid SLS-GAC Component

The hybrid component allows to focus on expected difficult subparts of the instance. This allows to get FAC variables that are linked to the (expected) most difficult subparts to satisfy. This component is described in Procedure `Hybrid`. Roughly, starting with the current assignment $\mathcal{A}$ provided by the SLS component, a variable in a violated constraint is selected and is assigned according to $\mathcal{A}$ (lines 4-5). The $dom/wdeg$ heuristic [3] is used as a tiebreaker amongst the set of variables of a violated constraint. A call is made to the `FIX` procedure (line 6), which operates and propagates GAC (Generalized Arc-Consistency) filtering steps. This procedure detects also conflicts (*i.e.*, empty domains for variables in $\mathcal{X}$) which trigger a backtrack on the last fixed variables (line 4-11). The GAC version used in the solver is based on AC3 [20]. The `FIX` procedure reduces the domain variables of $\mathcal{X}$ which is shared by all components. In fact, during the `Hybrid` procedure, SLS is still running but waits for decisions and does neither revise them nor the propagations done by the FIX procedure. The assignment $\mathcal{A}$, used by SLS, is thus in part fixed by this hybridization. In some sense, those fixed variables are tabu for SLS.

The variable $\#conf$ measures the number of encountered conflicts. After the `FIX` procedure and when this number has become strictly larger than the dynamical $maxConf$ threshold, it is estimated that the hydrid component is stuck and a call to MAC is

Procedure `SLS`

```
1  while ∃C ∈ 𝒞 s.t. C is violated by 𝒜 do
2      if a local minimum is reached then
3          if ∃ FAC variables then
4              add new FAC variables to 𝒮_e;
5              SLSprogress ⟵ SLSprogress + 1000 ;
6          else SLSprogress ⟵ SLSprogress − 1 ;
7          if SLSprogress < 0 then return;
8          else
9              Change the value in 𝒜 of one var. of 𝒳 according to the novelty escape
               strategy ;
10     else
11         Change the value in 𝒜 of one var. of 𝒳 s.t. the number of violated constraints
           decreases ;
12     if A new best configuration is obtained then
13         SLSprogress ⟵ SLSprogress + 1000 ;
14 result ⟵ true ;
```

made. When this threshold is not reached the search goes back to the SLS component, the collected filtering information being preserved.

Procedure `Hybrid`

```
1  level ⟵ 0;
2  #conf ⟵ 0 ;
3  while (#conf < maxConf) do
4      X ⟵ pick a variable according to dom/wdeg s.t. X appears in violated constraint
       by 𝒜 ;
5      v ⟵ the value of X in 𝒜 ;
6      FIX(X, v) ;
7      if (result = false) then return ;
8      SLS() ;
```

4.3 MAC-based Component

The MAC-based component starts with the initial CSP with the exception of filterings computed at level 0 during the SLS and the calls to `FIX` (line 1). This procedure is a standard MAC algorithm except that the focus is on collected FAC variables. The heuristic used to selected variable is $dom/wdeg$. For the first choices (lines 8-9), the variable is selected amongst the FAC variables collected during the SLS procedure. The next choices (line 11) are made within all variables. The use of FAC variables only for the first choices can be explained by the fact that the $dom/wdeg$ heuristics allows to focus on the same inconsistent part (*i.e.,* on the same core) whereas fixing another FAC

Procedure `FIX(`X, v`)`

```
1  dom(X) ⟵ {v} ;
2  level ⟵ level + 1;
3  GAC() ;
4  while ∃X' ∈ 𝒳 s.t. dom(X') = ∅ do
5      if level = 0 then
6          result ⟵ false ;
7          return;
8      Backtrack() ;
9      level ⟵ level − 1;
10     #conf ⟵ #conf + 1 ;
11     dom(X) ⟵ dom(X) \ {v};
12     GAC() ;
```

variable can lead the search to be dispersed and slowing down the discovery of a small proof of inconsistency.

The weights used by the $dom/wdeg$ heuristic are preserved from one iteration to the next one in Algorithm 1, and are shared by all search components.

Moreover, this MAC procedure does not necessarily perform a complete search since if the number of conflicts $\#conf$ becomes larger than the $maxConf$ before a final decision is obtained, then the process goes back to the SLS component. In this latter case, the $maxConf$ control variable is increased in a geometric manner.

As $maxConf$ is increased whenever the MAC component fails, this component will eventually give a final result when this boundary becomes larger than the number of conflicts needed by MAC to solve the CSP. Accordingly, FAC-SOLVER is complete.

Procedure `MAC`

```
1  Backjump(0) ;                                  // backjump to level 0
2  level ⟵ 0;
3  #conf ⟵ 0 ;
4  while (#conf < maxConf) do
5      if 𝒳 = ∅ then
6          result ⟵ true ;
7          return ;
8      if (#conf = 0) and (∃X ∈ 𝒮_e ∩ 𝒳) then
9          X ⟵ pick a variable in 𝒮_e ;
10     else
11         X ⟵ pick a variable according to dom/wdeg ;
12     v ⟵ pick randomly a value in dom(X) ;
13     FIX(X, v) ;
14     if (result = false) then return ;
```

5 Experimental Results

In order to assess the efficiency of FAC-SOLVER, we have considered benchmarks from the last CSP competitions [5, 6], which include binary vs. non-binary, random vs. real-life, satisfiable vs. unsatisfiable CSP instances. They were classified according to four types: 635 CSPs made of binary constraints in extension (BIN-EXT), 696 CSPs made of binary constraints in intension (BIN-INT), 704 CSPs involving n-ary constraints in extension (N-EXT) and 716 instances of CSPs of n-ary constraints in intention (N-INT). We have run four methods on all those instances: namely, our own implementation of SLS Walksat+Novelty, of an hybrid method combining SLS and GAC, of MAC and FAC-SOLVER. All tests have been conducted on a Xeon 3.2 GHz (2 G RAM) under Linux 2.6. Time-out has been set to 1200 seconds while a space limit has been set to 900 Mbytes. Note that the MAC version used in the experimentations makes use of a geometric restart policy that is similar way to our solver. Similarly, the SLS solver (Walksat+Novelty) used in our experimental comparison uses the same novelty heuristic than our solver. Note that, the solved unsatisfiable instances by novelty have been solved by GAC on the initial instance.

Table 1 summarizes the results in terms of the numbers of satisfiable and unsatisfiable instances that were solved. In each horizontal "total"-line, the solver that solves the most instances has been emphasized in gray. The main result is that FAC-SOLVER managed to solve more instances than any of the other methods globally for either satisfiable (SAT) or unsatisfiable (UNSAT) instances, and considering the subclasses of instances separately, for three types of CSPs. For the last type (binary CSP in extension), let us stress that the best solver is different in each of the three columns (SAT, UNS(AT) and TOT(AL)) and in each case, the number of solved instances by FAC-SOLVER is very close to the best one.

In Figure 2, five scatter points diagrams are given for a more detailed analysis. In each of them, two of the methods are pairwise compared w.r.t. instances that were solved by at least one of them. The X-axis represents the computing times in seconds by FAC-SOLVER whereas the Y-axis provides the performance of the second method. Results are expressed in seconds and represented according to a logarithmic scale. Instances were divided within the four classes detailed above. We provide separate diagrams for SAT and UNSAT instances. FAC-SOLVER is not compared with Walksat+Novelty on UNSAT instances since these instances are out of scope for the latter technique. The main information that can be drawn from these diagrams is as follows.

- More instances are located on the Y=1200 line than on the X=1200 one. This shows, as Table 1 summarizes it, that FAC-SOLVER solves more instances than any of the other considered methods.
- With the exceptions of UNSAT instances for MAC, there are more points located above the diagonals showing that FAC-SOLVER is generally more efficient than the other methods. Also, for UNSAT instances, the difference of time performance between pairs of methods is globally smaller than for SAT instances (points are less dispersed and are closer to the diagonal). FAC-SOLVER is generally more efficient than a mere combination of SLS and GAC. It is also more efficient than MAC on SAT instances (mainly due to the power of SLS). But there is no free lunch, the time

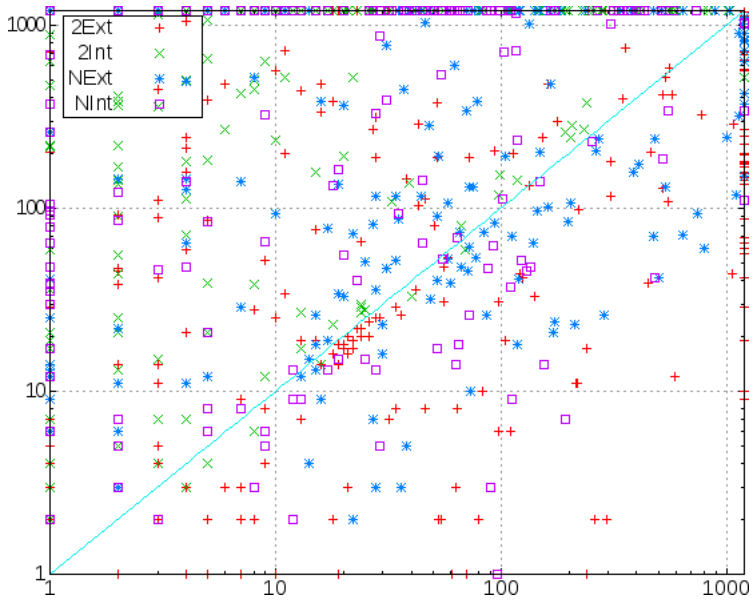
		NOVELTY			SLS+GAC			MAC			FAC-SOLVER		
		SAT	UNS	TOT	SAT	UNS	TOT	SAT	UNS	TOT	SAT	UNS	TOT
2-EXT	ACAD	7	0	7	7	2	9	7	2	9	7	2	9
	PATT	106	0	106	100	38	138	83	38	121	99	39	138
	QRND	24	0	24	24	51	75	24	51	75	24	51	75
	RAND	206	0	206	197	105	302	194	110	304	193	106	299
	REAL	6	0	6	7	0	7	7	0	7	7	0	7
	TOTAL	349	0	349	335	196	531	315	201	516	330	198	528
2-INT	ACAD	38	7	45	37	40	77	37	40	77	38	40	78
	BOOL	0	1	1	0	1	1	0	1	1	0	1	1
	PATT	112	0	112	150	60	210	146	62	208	152	62	214
	REAL	47	74	121	74	102	176	75	103	178	75	103	178
	TOTAL	197	82	279	261	203	464	258	206	464	265	206	471
N-EXT	BOOL	70	1	71	74	75	149	74	70	144	74	74	148
	PATT	6	0	6	30	0	30	29	0	29	30	0	30
	QRND	43	0	43	40	40	80	33	40	73	45	40	85
	RAND	70	0	70	68	32	100	72	34	106	70	34	104
	REAL	41	29	70	45	114	159	47	115	162	47	115	162
	TOTAL	230	30	260	257	261	518	255	259	514	266	263	529
N-INT	ACAD	40	0	40	39	23	62	36	23	59	40	23	63
	BOOL	145	1	146	156	12	168	146	12	158	162	13	175
	PATT	88	5	93	103	19	122	95	20	115	102	18	120
	REAL	85	2	87	152	3	155	150	3	153	152	3	155
	TOTAL	358	8	366	450	57	507	427	58	485	456	57	513
	TOTAL	**1134**	**113**	**1247**	**1293**	**717**	**2010**	**1255**	**724**	**1979**	**1317**	**724**	**2041**

Table 1. Experimental results.

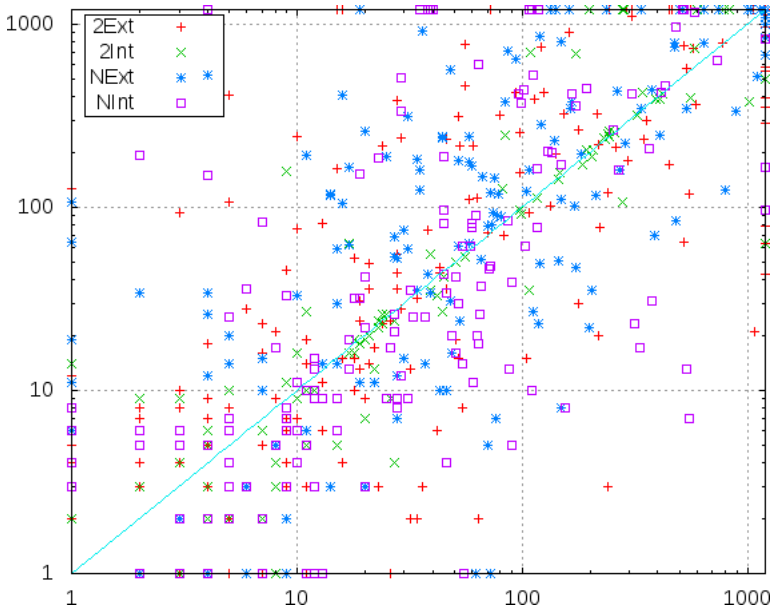
spent on SLS by FAC-SOLVER on UNSAT instances leads to some small and very acceptable time overheads, compared with MAC. This is perhaps the price to pay to solve more unsatisfiable instances than MAC within the same price constraints, thanks to the collected information by SLS.

An important issue is the way according to which the efficiency of FAC-SOLVER might depend on the specific initial assignment selected by its SLS component. Actually, it appears that, on average, this dependency is weak and is not a serious troubling factor affecting the results. To show this robustness, we have selected 96 instances within the above benchmarks in a random fashion but according to their relative importance in each of the four classes of instances. For each of these instances, 50 successive runs of FAC-SOLVER have been conducted with a different initial (randomly generated) assignment. When an instance was solved by at least one run, it was also solved by the 49 other runs in 97 % of the situations, with a very low 2.52 seconds average deviation.

To show the importance of FAC variables, we have run our solver with and without computing and using the FAC variables. Table 2 provides typical results. The use of FAC variables allows more instances to be solved. Most of the time, the use of FAC variables can solve benchmarks more quickly. In rare cases, the use of FAC variables

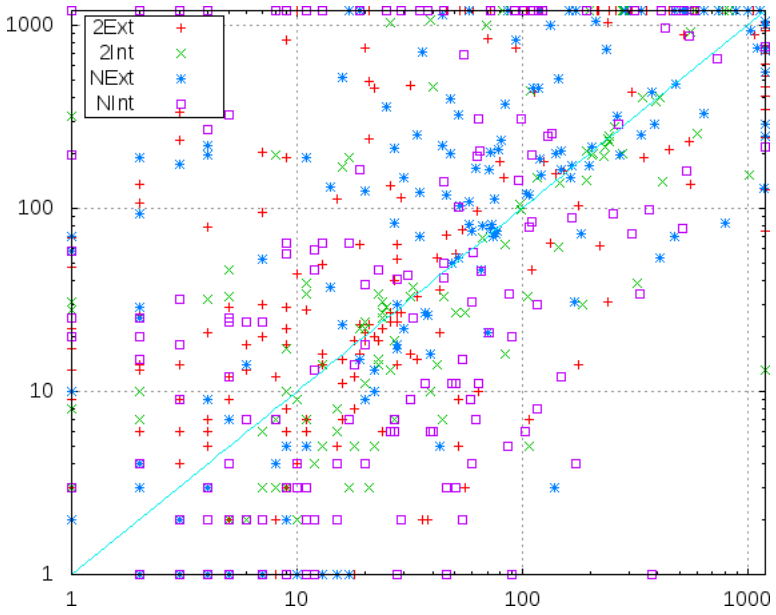


(a) FAC-SOLVER vs. NOVELTY

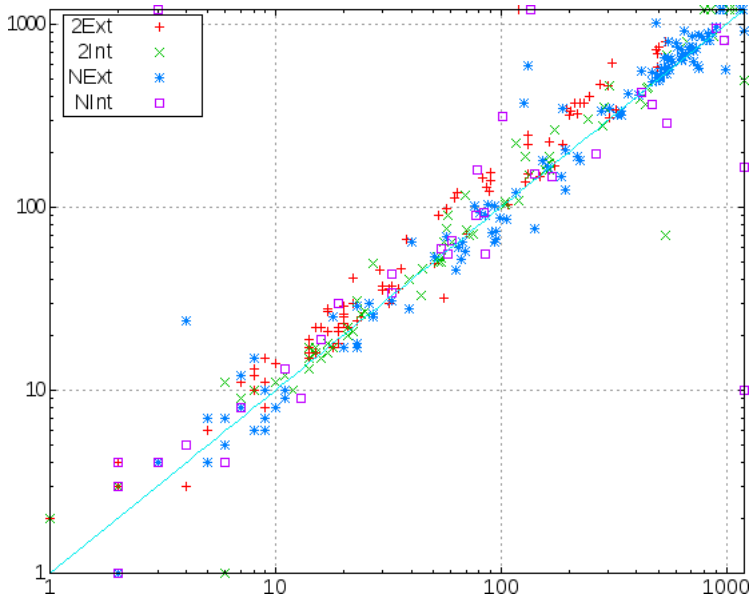


(b) FAC-SOLVER vs. SLS+GAC

Fig. 2. Pairwise comparisons between FAC-SOLVER and classical solvers: (a) (b) (c) satisfiable instances/(d) (e) unsatisfiable instances.

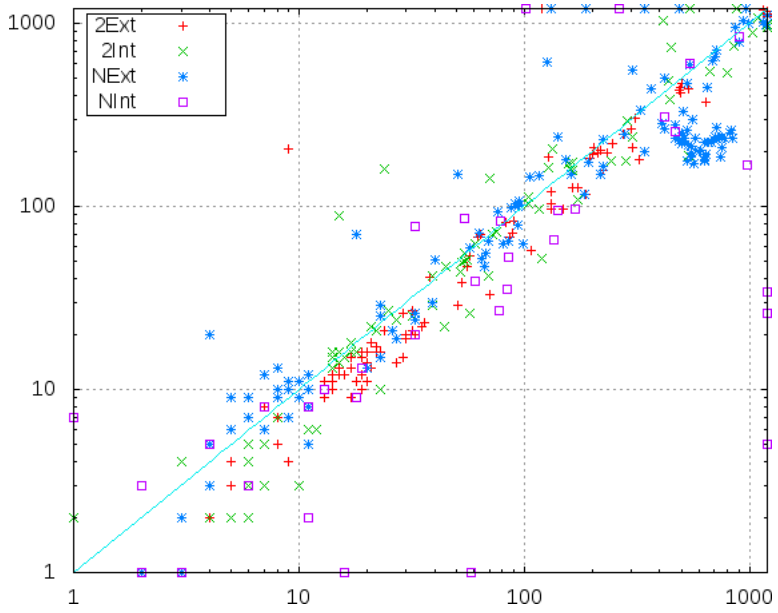


(c) FAC-SOLVER vs. MAC



(d) FAC-SOLVER vs. SLS+GAC

Fig. 2. Pairwise comparisons between FAC-SOLVER and classical solvers: (a) (b) (c) satisfiable instances/(d) (e) unsatisfiable instances (con't).



(e) FAC-SOLVER vs. MAC

Fig. 2. Pairwise comparisons between FAC-SOLVER and classical solvers: (d) (e) unsatisfiable instances (con't).

wastes time. This is because the solver wastes time to compute FAC variables that are not used when the instance is solved directly by the SLS solver or when the benchmarks is globally inconsistent. In this case, all variables are potentially FAC variables and their computation also wastes time.

6 Perspectives and Conclusions

In this paper a FAC variables concept has been introduced and investigated w.r.t. CSP solving. One goal of this study was to develop a CSP solving method that would at least match the efficiency of each best current approach on each class of traditional CSPs instances. In this respect, our experimental results show the extent to which this goal has been met.

One question that naturally arises is the extent to which the various findings and components implemented in FAC-SOLVER do actually take part in the increased efficiency. Actually, it appears that *each* finding and search component (FAC variables, use of $dom/wdeg$ heuristic in SLS, hybrid method involving SLS and filtering techniques) were necessary to ensure the supremacy of the method. Especially, we have e.g. measured that FAC variables were detected in 56 % of the instances and that they play a crucial role even in consistent instances.

FAC-SOLVER remains a basic algorithm and could be fine-tuned in several ways. Especially, comprehensive experimental studies could allow to optimize its various control

Instance	SAT/UNSAT?	time (FAC-SOLVER)	time (FAC-SOLVER without FAC variables)
uclid-elf-rf8	UNSAT	**305.15**	time out
uclid-37s-smv	UNSAT	**387.50**	659.58
par-16-5	SAT	**168.87**	329.06
primes-10-40-2-7	SAT	**891.01**	time out
primes-20-20-2-7	SAT	976.68	**313.31**
queensKnights-100-5-add	UNSAT	**1,120.18**	time out
queensKnights-100-5-mul	UNSAT	**1,165.81**	time out
queensKnights-80-5-mul	UNSAT	**343.68**	time out
rand-2-40-18	UNSAT	41.47	**1.61**

Table 2. Using or not FAC variables in FAC-SOLVER: typical results.

variables and factors, which we fixed quite arbitrarily. Moreover, our implementation does not include usual CSP simplification techniques like the exploitation of symmetries or global constraints. We believe that the integration of these techniques could also dramatically improve FAC-SOLVER. Also, it would be interesting to explore the relaxation of the FAC variable concept to encompass also variables that occur in *most* or *some preferred* falsified contraints (instead of all of them). This could prove useful for e.g. CSP instances containing non-overlapping MUCs.

Finally, we believe that the FAC variable concept is a good trade-off between the effective computational cost spent by SLS to find some of them, and what would be the theoretically best branching variable for MAC-based algorithm giving rise to the shortest proofs. FAC variables are variables taking part in all unsatisfiable minimal subsets of constraints, which often appear to be the difficult parts of unsatisfiable CSPs. However, it is easy to find out unsatisfiable CSPs where FAC variables do not conceptually take part in the real causes of unsatisfiability but rather simply appear as variables occurring in all MUCs while, at the same time, they are not related to the actual conflicting information. Refining the FAC variable concept to better capture the essence of unsatisfiability while keeping efficient heuristics that can help finding them remains an exciting challenge.

References

1. G. Audemard, J.M. Lagniez, B. Mazure, and L. Saïs. Boosting local search thanks to CDCL. In *LPAR'10*, pages 474–488, 2010.
2. C. Bessière, J.C. Régin, R. Yap, and Y. Zhang. An optimal coarse-grained arc consistency algorithm. *Artificial Intelligence*, 165(2):165–185, 2005.
3. F. Boussemart, F. Hemery, C. Lecoutre, and L. Saïs. Boosting systematic search by weighting constraints. In *ECAI'04*, pages 146–150, 2004.
4. P. Cheeseman, B. Kanefsky, and W. Taylor. Where the really hard problems are. In *IJCAI'91*, pages 331–340, 1991.
5. Third international CSP solver competition, 2008. `http://cpai.ucc.ie/08/`.
6. Fourth international CSP solver competition, 2009. `http://cpai.ucc.ie/09/`.

7. C. Eisenberg and B. Faltings. Making the breakout algorithm complete using systematic search. In *IJCAI'2003*, pages 1374–1375, 2003.
8. Carlos Eisenberg and Boi Faltings. Using the breakout algorithm to identify hard and unsolvable subproblems. In *CP'2003*, pages 822–826, 2003.
9. T. Eiter and G. Gottlob. On the complexity of propositional knowledge base revision, updates and counterfactuals. *Artificial Intelligence*, 57:227–270, 1992.
10. P. Galinier and J.K. Hao. Tabu search for maximal constraint satisfaction problems. In *CP'97*, pages 196–208, 1997.
11. E. Goldberg. Boundary points and resolution. In *SAT'09*, pages 147–160, 2009.
12. J. Gu. Design efficient local search algorithms. In *IEA/AIE'92*, pages 651–654, 1992.
13. É. Gégoire, B. Mazure, and C. Piette. Local-search extraction of muses. *Constraints*, 12(3):325–344, 2007.
14. H.Hoos. An adaptive noise mechanism for walksat. In *AAAI'02*, pages 655–660, 2002.
15. E. Hébrard. Mistral 1.529, 2006. `http://4c.ucc.ie/~ehebrard/mistral/doxygen/html/main.html`.
16. N. Jussien and O. Lhomme. Combining constraint programming and local search to design new powerful heuristics. In *MIC'2003*, 2003.
17. S. Kirkpatrick, D. Gelatt Jr., and M. Vecchi. Optimization by simulated annealing. *Science*, 220(4598):671–680, 1983.
18. C. Lecoutre and F. Hemery. A study of residual supports in arc consistency. In *IJCAI'2007*, pages 125–130, 2007.
19. C. Lecoutre and S. Tabary. Abscon 112: towards more robustness. In *CSC'08*, pages 41–48, 2008.
20. A. Mackworth. Consistency in networks of relations. *Artificial Intelligence*, 8(1):99–118, 1977.
21. B. Mazure, L. Saïs, and É. Grégoire. Boosting complete techniques thanks to local search methods. *Annals of Mathematics and Artificial Intelligence*, 22(3-4):319–331, 1998.
22. D. McAllester, B. Selman, and H. Kautz. Evidence for invariants in local search. In *AAAI'97*, pages 321–326, 1997.
23. S. Merchez, C. Lecoutre, and F. Boussemart. Abscon: A prototype to solve CSPs with abstraction. In *CP'01*, pages 730–744, 2001.
24. S. Minton, M. Johnston, A. Philips, and Philip Laird. Minimizing conflicts: A heuristic repair method for constraint satisfaction and scheduling problems. *Artificial Intelligence*, 58(1-3):161–205, 1992.
25. D. Sabin and E. Freuder. Contradicting conventional wisdom in constraint satisfaction. In *ECAI'94*, pages 125–129, 1994.
26. B. Selman, H. Kautz, and B. Cohen. Noise strategies for improving local search. In *AAAI'94*, pages 337–343, 1994.
27. B. Selman, H. Levesque, and D. Mitchell. A new method for solving hard satisfiability problems. In *AAAI'92*, pages 440–446, 1992.
28. N. Tamura, A. Taga, S. Kitagawa, and M. Banbara. Compiling finite linear CSP into SAT. *Constraints*, 14(2):254–272, 2009.
29. Choco Team. Choco: an open source Java constraint programming library. Research report 10-02-INFO, Ecole des Mines de Nantes, 2010.