On Finding Minimally Unsatisfiable Cores of CSPs

Eric Gregoire, Bertrand Mazure, Cedric Piette

International Journal on Artificial Intelligence Tools

ON FINDING MINIMALLY UNSATISFIABLE CORES OF CSPs

Éric Grégoire Bertrand Mazure Cédric Piette
Université Lille-Nord de France, Artois, F-62307 Lens
CRIL, F-62307 Lens
CNRS UMR 8188, F-62307 Lens
rue Jean Souvraz SP18 F-62307 Lens France
{*gregoire,mazure,piette*}*@cril.fr*

Received (Day Month Year)
Revised (Day Month Year)
Accepted (Day Month Year)

When a Constraint Satisfaction Problem (CSP) admits no solution, it can be useful to pinpoint which constraints are actually contradicting one another and make the problem infeasible. In this paper, a recent heuristic-based approach to compute infeasible minimal subparts of discrete CSPs, also called Minimally Unsatisfiable Cores (MUCs), is improved. The approach is based on the heuristic exploitation of the number of times each constraint has been falsified during previous failed search steps. It appears to enhance the performance of the initial technique, which was the most efficient one until now.

Keywords: CSP, MUC, Explanation, Inconsistency

1. Introduction

Constraint Satisfaction Problems (CSPs) form a very active domain of research and application in Artificial Intelligence, that has found its way into many problem domains (see e.g. [1,12]). Roughly, a CSP is a set of constraints, involving a set of variables having their own instantiation domains. Solving a CSP consists in discovering values for the involved variables in such a way that all constraints are satisfied, or in showing that no values from the instantiation domains can satisfy all constraints simultaneously.

In this paper, we are concerned with unsatisfiable CSPs, namely CSPs for which no solution exists. More precisely, we address the problem of extracting *Minimally Unsatisfiable Cores* (MUCs) of finite CSPs, namely of CSPs involving finite numbers of constraints and variables with finite instantiation domains. A MUC is a set of infeasible constraints that is minimal in the sense that dropping any of its member makes the remaining subset of constraints feasible. Obviously enough, providing a user with such a piece of information can be highly valuable when a CSP exhibits no solution. Indeed, it provides one explanation of infeasibility that cannot be made smaller in terms of involved constraints. Assume for example, that a com-

plex scheduling problem is expressed in terms of a CSP, where different constraints represent the sequences of tasks to be performed, the required resources together with their time-dependent availability. When such a problem does not have a solution, it is important to pinpoint which constraints actually conflict with one another and cannot be solved. Indeed, circumscribing the smallest sets of constraints that are the actual sources of infeasibility can help the user to understand this infeasibility, and fix it.

Unfortunately, computing MUCs is a highly intractable problem in the worst case. For example, a specific case of CSPs is given by SAT, which is the NP-complete problem consisting in checking the satisfiability of a set of Boolean clauses, where a clause is a disjunction of literals, where a literal is a propositional variable that can be negated. Deciding whether an unsatisfiable set of clauses of a SAT instance is minimal or not is DP-Complete [26], which belongs to the second level of the polynomial hierarchy.

Very recently a novel approach has been presented in [17], called `DC(wcore)`, to compute MUCs. It appears to be the most efficient one for most CSPs classes. In particular, `DC(wcore)` improves a previous method introduced in [3] to extract a MUC, that was introduced in the specific context of model-based diagnosis. It also proves more competitive than the QuickXplain [18,19] method to compute MUCs, which is the seminal work in this domain of research. In this paper, a variant technique that improves `DC(wcore)` very often is introduced.

The paper is organized as follows. In the following section, the reader is provided with the necessary background about CSPs. In Section 3, the first step of Hemery and co-authors' `DC(wcore)` technique, namely the `wcore` procedure, is briefly recalled. In Section 4, our improvement to enhance `wcore` is introduced. In Section 5, the second step of `DC(wcore)` is presented and also improved from a practical point of view. In Section 6, extensive experimental results are described, showing the value of our proposed enhancements. Main related works are discussed in Section 7. In Section 8, interesting paths for future research are discussed.

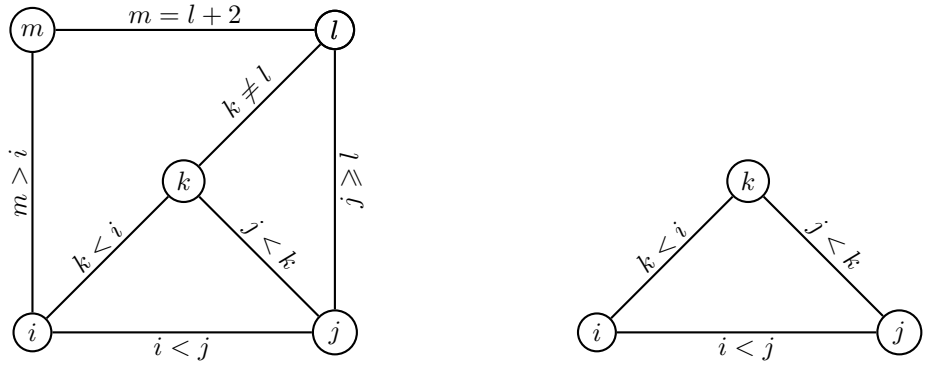
2. CSP: Technical Background

In this section, the reader is provided with the basic notions about CSPs and MUCs that are necessary in this paper.

Definition 2.1. A *Constraint Satisfaction Problem*, in short CSP, is a pair $P = (V,C)$ where

(i) V is a finite set of n variables s.t. each variable $x \in V$ has an associated finite instantiation domain, denoted $dom(x)$, which contains the set of values allowed for x,

(ii) C is a finite set of e constraints s.t. each constraint $c \in C$ involves a subset of variables of V, called *scope* and denoted $vars(c)$, and is given an associated relation $rel(c)$, which contains the set of tuples allowed for the variables of its scope.



(a) CSP: an example (b) extracted MUC

Fig. 1: Graphical Representation of the CSP of Example 2.1

Definition 2.2. Solving a CSP $P = (V,C)$ consists in checking whether P admits at least one solution, i.e. an assignment of values for all variables of V s.t. all constraints of C are satisfied. If P admits at least one solution then P is called to be satisfiable else P is called to be unsatisfiable.

Example 2.1. Let V be $\{i,j,k,l,m\}$ where each variable has the same domain $\{0,1,2,3,4\}$. Let $C = \{m > i, m = l + 2, k < i, k \neq l, j < k, i < j, j \geq l\}$ be a set of 7 constraints. In Figure 1a, the CSP $P = (V,C)$ is represented as a non-oriented graph, where each variable is a node and each constraint is an edge, labelled with its corresponding relation. P is unsatisfiable. Indeed, no assignment of values for all variables of V allows all constraints of C to be satisfied at the same time.

A MUC is a subpart of a CSP that is unsatisfiable and that does not contain any proper subpart that is also unsatisfiable.

Definition 2.3. Let $P = (V, C)$ and $P' = (V', C')$ be two CSPs. P' is an *unsatisfiable core*, in short a *core*, of P iff

(i) P' is unsatisfiable
(ii) $V' \subseteq V$ and $C' \subseteq C$

P' is a *Minimal Unsatisfiable Core* (MUC) of P iff

(i) P' is a core of P
(ii) there does not exist any proper core of P'

Example 2.2. In the above example, P is unsatisfiable. Indeed, P contains the MUC represented in Figure 1b: no values for i, j and k can be found such that all constraints are satisfied, and dropping one constraint leads to feasibility.

Solving a CSP is an NP-complete problem. There exists many complete and incomplete techniques to address it. Most "efficient" complete techniques rely on a complete depth-first search with backtracking. At each step the set of currently instantiated variables is incremented and some filtering consistency checks are performed. One widely used family of filtering algorithms is called MAC (Maintaining Arc Consistency) [29]. Roughly, MAC propagates the values of the currently instantiated variables and filters the remaining domains of possible values for the other variables by removing the values that are not consistent with the current state. When one domain of a variable becomes empty, this means that the lastly instantiated variable conducts some constraints to be violated. Hence, the algorithm needs to backtrack in order to consider another possible value for this variable. For more information about CSP solving, the reader is refered to [5].

In the following we consider a complete CSP solver based on the MAC implementation by Chmeiss and Saïs [9].

3. The `wcore` Technique by Hemery *et al.*

Basically, the `DC(wcore)` technique by Hemery *et al.*[17] is based on two successive steps, namely `wcore` and `DC`. First, a core that is not guaranteed to be minimal and thus to be a MUC is extracted using the so-called `wcore` procedure. Then, a form of fine-tune process is performed to deliver an actual MUC from this core. Our contribution consists of an improvement of both steps.

`wcore` is based on the following findings. First, it is well-known[3] that when the unsatisfiability of a CSP instance is proved thanks to a filtering search algorithm, this one can deliver a core of the CSP. It is formed of all the constraints that have been involved in the proof of unsatisfiability, namely all the constraints that have been used during the search to remove by propagation at least one value from the domain of any variable. Such constraints are called *active*. `wcore` makes use of the MAC algorithm which maintains arc consistency by exploiting the AC3 [24] procedure. As described in Hemery *et al.*[17], it involves successive revisions of arcs (i.e. pairs composed of a constraint and of a variable) in order to remove the values that are not consistent anymore with the current state. At the heart of the `wcore` system is thus the `revise` function depicted in Algorithm 1, which removes all the values of the domain of a given variable that are not currently supported the given constraint. The function also allows the *active* property to be triggered for the constraint causing such a removal.

When the CSP is shown unfeasible, active constraints form a core since the other constraints did not actually take part to this proof of inconsistency; consequently, constraints that are not active could be removed while the problem is kept unsatisfiable.

Clearly enough, the resulting core can depend on the the way the partial assignments are investigated, which is guided by the branching heuristic. In practice, `wcore` takes advantage of the powerful *dom/wdeg* heuristic [6], which consists in as-

Algorithm 1: `revise`

Input: a CSP: (V, C), a variable $v \in V$
Output: $false$ if a domain wipe-out occurs, otherwise $true$

```
1  begin
2      foreach a ∈ dom(v) do
3          foreach c ∈ C s.t. v ∈ scope(c) do
4              if find_a_support(a, c) = false then
5                  dom(v) ⟵ dom(v)\{a} ;
6                  active[c] ⟵ true ;
7                  if dom(v) = ∅ then
8                      weight[c] ⟵ weight[c] + 1 ;
9                      return false
10     return true ;
11 end
```

Algorithm 2: `wcore`

Input: a CSP: (V, C)
Output: a core: (V, C')

```
1  begin
2      foreach c ∈ C do weight[c] ⟵ 1 ;
3      C_core ⟵ C ;
4      repeat
5          C' ⟵ C_core ;
6          foreach c ∈ C do active[c] ⟵ false ;
7          MAC_revise(V, C) ;
8          C_core ⟵ {c ∈ C | active[c] = true} ;
9      until |C_core| < |C'| ;
10     return (V, C_core) ;
11 end
```

sociating for each constraint a counter initialized to 1 and incremented each time the corresponding constraint is involved in a conflict, namely each time it has been used by the filtering step to wipe out the domain of a variable.

In this respect, the *dom/wdeg* heuristic selects the variable with the smallest ratio between the current domain size and a weighted degree, which is defined as the sum of the counters of the constraints in which the variable is involved. This technique allows one to take the *difficulty* to satisfy the constraints related to each variable into consideration, in order to quickly encounter a conflict if the current instantiation does not lead to a model. Hence, it is a dynamic and adaptive

variable ordering heuristic that can be expected to guide the systematic search toward unsatisfiable or hard parts of the considered CSP.

Thus, the first step of `DC(wcore)`, depicted in Algorithm 2, is a loop where calls to a complete MAC-based solver (using the filtering procedure involving the `revise` function) are iterated on a CSP instance as long as the number of active constraints decreases. Importantly, the counters, or weights of the aforementioned *dom/wdeg* heuristic associated to each variable are preserved from one call of MAC to the next one. By keeping these counters, or weights, from one call to a complete method to the next one, the solver focuses on some over-constrainted part of the problem, and reduces more and more the number of constraints that are useful during the computation. It has been shown that recording those counters is extremely valuable for obtaining smaller cores at each iteration step, from an empirical point of view.

Accordingly, `wcore` delivers a core when the last call to the MAC-based solver leads to a larger or equal number of active constraints than a previous call. We then consider the smallest computed core, in terms of the number of involved constraints.

4. First Improvement: the MAC-based Solver Backtracks too Early

The power of `wcore` relies on the efficiency of the MAC-based solver. Such a solver increments the counters of the constraints that are violated at filtering steps, and resumes its exploration by focusing on "difficult" constraints first, thanks to the use of the *dom/wdeg* heuristic.

The goal of the MAC-based solver is to show in the most efficient manner that a CSP is either unsatisfiable or exhibits at least one solution. However, we believe that it could prove useful to modify the solver when the final goal is to get a MUC. More precisely, when the MAC-based solver has shown that one constraint is violated due to the propagation of the value of the last instantiated variable, it backtracks. We believe that such a backtrack occurs too early. Other constraints are also perhaps violated in the same circumstances and it could prove useful to take all those violations into consideration, too. Indeed, such a more systematic checking feature has already been proved useful in other contexts (see e.g. [30]). This could be recorded through the counters associated with the constraints that will be used further on by the *dom/wdeg* ordering heuristic. Clearly, such a policy could require (a small) computation overhead. However, our experimental studies show us that collecting this strategic information proves useful and makes the whole procedure become more efficient, most often.

Example 4.1. For instance, let us consider the CSP $P = (\{a, b, c, d\}, \{a \neq b, b+c = 2, a+c = 2, c \leq d, b+d \neq 2\})$, with $\{0, 1, 2\}$ as instantiation domain for each variable. This problem is represented in Figure 2 *(1)*, with nodes labelled by both variables names and their respective domains.

Assume that a search for satisfiability is run, starting by assigning b to 0. First, the domains of neighboring variables are filtered according to arc-consistency: values

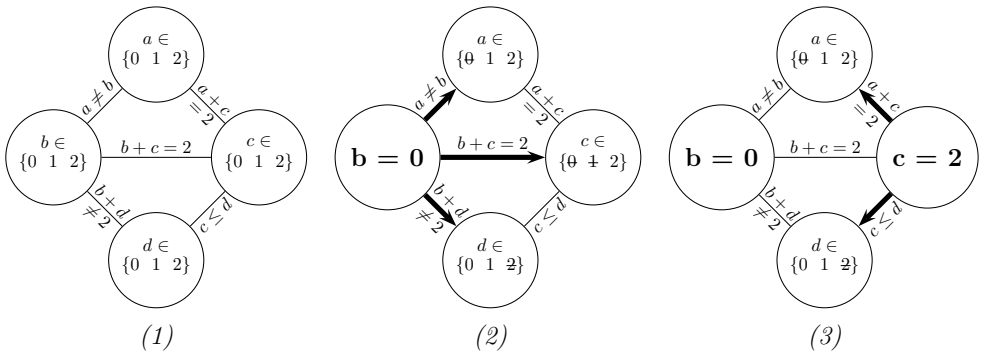


Fig. 2: Filtering domains of variables w.r.t. arc-consistency

that do not satisfy constraints w.r.t. the current partial assignment are removed from the domains of those variables. For instance, the value *2* is removed from the domain of d, since it falsifies the constraint $b + d \neq 2$, assuming that $b = 0$. The resulting instantiation domains are depicted in Figure 2 *(2)*. As a result of this first step of arc-consistency enforcement, the only remaining value for c is 2. This variable is thus assigned to *2* thanks to this filtering step. Then, arc-consistency is performed w.r.t. this new piece of information. Assume the domain of variable a is first filtered by this second step. Clearly, its domain becomes empty since no previously remaining value satisfies the $a + c = 2$ constraint. Accordingly, a backtrack is triggered and `wcore` increments the weight of this latter constraint. However, this constraint is not the only one that is falsified by the current partial instantiation. Indeed, $c \leq d$ is also violated. It seems natural to increment the weight of all violated constraints, rather than the first discovered one, only.

Hence, we have modified a MAC-based solver in such a way that it does not backtrack when a constraint is shown infeasible under a partial instantiation. On the contrary, all *relevant* constraints are checked for feasibility under a given partial assignment of the set of variables.

First, the `revise` function has been adapted to this end. The new function is called `full-revise` and is depicted in Algorithm 3. Contrary to `revise`, the new function does not stop its computation as soon as a domain wipe-out occurs. Instead, a list L_a of all the constraints that would cause the removal of a tested value a from the domain of the variable v is recorded. Then, the value a is removed provided that the list L_a is not empty (line 7). When a domain wipe-out occurs for v, the weight of each constraint of L_a is incremented (lines 13-14) whereas `revise` would increment the weight of one constraint, only. The set of *active* constraints is updated in the following way. The active constraints must form a somewhat irredundant proof of unsatisfiability since they are intended to form a MUC. Accordingly, a new constraint is set active only when the no constraint from the L_a list is already active

Algorithm 3: `full-revise`

Input: a CSP: (V, C), a variable $v \in V$
Output: $false$ is a domain wipe-out occurs, otherwise $true$

```
1  begin
2    foreach a ∈ dom(v) do
3      L_a ⟵ ∅ ;
4      foreach c ∈ C s.t. v ∈ scope(c) do
5        if find_a_support(a, c) = false then
6          L_a ⟵ L_a ∪ {c} ;
7      if L_a ≠ ∅ then
8        dom(v) ⟵ dom(v)\{a} ;
9        if ∄c ∈ L_a s.t. active[c] = true then
10         c_a ⟵ pick_a_constraint(L_a) ;
11         active[c_a] ⟵ true ;
12       if dom(v) = ∅ then
13         foreach c ∈ L_a do
14           weight[c] ⟵ weight[c] + 1 ;
15         return false ;
16   return true ;
17 end
```

(lines 9-11). Such a constraint is selected randomly within L_a.

Second, the MAC-based solver has also be modified in order to take the following phenomenon into account. Whenever the domain of a variable is wiped out, other variables can have their domains wiped out in their turn if arc-consistency is performed until a fixed-point occurs. Indeed, any constraint linking a variable with an empty domain is violated, leading the domain of the involved variables to be wiped out. In order to avoid this kind of avalanche effect, the filtering process has been controled in the following way: let us assume that the arc-consistency procedure is filtering the domains of variables related to a given variable v (namely variables linked by a non tautological constraint to v). If one of those domains becomes empty, then arc-consistency continues on the remaining variables linked to v, and the process is then stopped.

Finally, `wcore` has also be revisited in the following way. Instead of iterating calls to the MAC-based solver with the same initial CSP instance as input, these calls are focused on the previously obtained set of active constraints, delivering at each step a decreasing core. Such a policy that concentrates on refining a specific core has been shown more efficient from an experimental point of view.

We call the resulting procedure `full-wcore` (weighting *all* falsified constraints)

Algorithm 4: `full-wcore`

Input: a CSP: (V, C)
Output: a core: (V, C')

```
1  begin
2      foreach c ∈ C do weight[c] ⟵ 1 ;
3      C_core ⟵ C ;
4      repeat
5          C' ⟵ C_core ;
6          foreach c ∈ C_core do active[c] ⟵ false ;
7          MAC_full-revise(V, C_core) ;
8          C_core ⟵ {c ∈ C | active[c] = true} ;
9      until |C_core| < |C'| ;
10     return (V, C_core) ;
11 end
```

as a reference to the `wcore` (weight core) name; it is depicted in Algorithm 4.

As our extensive experimental studies show, taking all the constraints that trigger infeasibility into consideration when a conflict occurs improves the performance of both `wcore` and `DC(wcore)`.

5. Second Improvement: DC is not Fully Exploiting the Counting Heuristic

Both `wcore` and `full-wcore` provide a core P formed of e constraints that is an upper-approximation of a MUC. The second step of `DC(wcore)` is intended to extract one MUC from this core; it is based on the following property.

Let any ordering c_1,...,c_e of the constraints in P. P always contains one transition constraint c_i, which is such that c_1,...,c_{i-1} is satisfiable and c_1,...,c_i is unsatisfiable. Clearly, c_i belongs to at least one MUC of P, and all constraints from c_{i+1},...,c_e do not belong to this MUC, and can be left aside. Once the transition constraint c_i has been found, the ordering c_1,...,c_i is reorganized as c_i,c_1,...,c_{i-1}. The second transition constraint c_j is now to be found in c_i,c_1,...,c_{i-1}. When it is found, the ordering becomes c_i,c_j,c_1,...,c_{j-1}. The process is iterated and stops when the set of transition constraints that has been found is shown unsatisfiable. This set is then a MUC and the final result can be delivered. The principle of this iterative technique has already been exploited in [11,18,27].

A technique to find the transition constraint is thus central in this approach. Hemery and his co-authors discussed three different families of approaches to discover transition constraints. The first-ones are called constructive because they consider and add constraints of the core successively in a set until this set becomes unsatisfiable. The last introduced constraint is the transition one. These approaches introduced in [11] do not appear competitive from a computational point of view. The

Algorithm 5: DS(`XXcore`): destructive algorithm

Input: a CSP: (V, C)
Output: a MUC: (V, C')

```
1  begin
2      (V, C' = {c_1, ..., c_|C'|}) ⟵ XXcore((V, C)) ;
3      k ⟵ 0 ;
4      repeat
5          i ⟵ |C'| ;
6          while (MAC((V, {c_1, ..., c_{i-1}})) proves unsatisfiability do
7              i ⟵ i − 1 ;
8          transitionConstraint ⟵ c_i ;
9          for j = (i − 1) downto 1 do
10             c_{j+1} ⟵ c_j ;
11         c_1 ⟵ transitionConstraint ;
12         C' ⟵ C' \ {c_{i+1}, ..., c_|C'|} ;
13         k ⟵ k + 1 ;
14     until k = |C'| ;
15     return (V, C') ;
16 end
```

second family of approaches are called destructive in the sense that they remove constraints from the core until it becomes satisfiable; the constraint that has been removed in the last place is the transition one (Algorithm 5, first introduced in [3]). Finally, Hemery and his co-authors introduce a dichotomic search on the range of considered constraints to find the transition one (Algorithm 6).

The worst-case complexity of the approaches based on the constructive, destructive and dichotomic approaches can be characterized by the number of calls to a complete MAC prover [17]. They are $\mathcal{O}(e.k_e)$, $\mathcal{O}(e)$ and $\mathcal{O}(log(e).\ k_e)$, respectively, where e is the number of constraints of the considered problem P and k_e is the number of constraints in the extracted final MUC.

Based on this worst-case analysis, Hemery *et al.* recommend the use of their dichotomic approach, which they call `DC(wcore)`. Especially, they show that its worst-case complexity is better than the complexity of QuickXplain [18]. They also recommend to order the constraints of P according to their decreasing aforementioned "hardness" scores collected during the first step.

Our intuition is that whereas this analysis is correct for the worst-cases, it misses some important practical heuristic information that has already been exploited in the `wcore` and `full-wcore` procedures. Indeed, unless we are faced with worst-cases situations, constraints with a high score are expected to exhibit a higher probability of belonging to MUCs than lower-scores constraints. Thus constraints that belong to the MUC are not expected to be uniformly dispersed among the constraints of

Algorithm 6: DC(`XXcore`): dichotomic algorithm

```
   Input: a CSP (V, C)
   Output: a MUC (V, C')
 1 begin
 2   (V, C' = {c_1, ..., c_|C'|}) ⟵ XXcore((V, C)) ;
 3   k ⟵ 0 ;
 4   repeat
 5     min ⟵ k + 1 ;
 6     max ⟵ |C'| ;
 7     while (min ≠ max) do
 8       med ⟵ (min + max)/2 ;
 9       if (MAC((V, {c_1, ..., c_med})) proves unsatisfiability) then
10         max ⟵ med ;
11       else
12         min ⟵ med + 1 ;
13     transitionConstraint ⟵ c_min ;
14     for j = (min − 1) downto 1 do
15       c_{j+1} ⟵ c_j ;
16     c_1 ⟵ transitionConstraint ;
17     C' ⟵ C' \ {c_{min+1}, ..., c_|C'|} ;
18     k ⟵ k + 1 ;
19   until k = |C'| ;
20   return (V, C') ;
21 end
```

P. On the contrary, they are expected to be grouped within the set of high-scores constraints whereas the constraints that do not belong to the MUC tend to be located in the low-score region. The dichotomic approach does not exploit such a heuristic information. In particular, assume that the core P is already a MUC. In this case the destructive approaches will require $\mathcal{O}(e)$ calls to MAC whereas the dichotomic one will require $\mathcal{O}(log(e).e)$ calls. Let us also note that `full-wcore` is expected to deliver a better approximation of a MUC than `wcore` does. On the other hand, it is natural to expect the dichotomic approach to be more efficient when constraints in the MUC are dispersed in P in a random way. Accordingly, we propose to replace the systematic calls to the dichotomic procedure by means of the following policy that we have found experimentally more efficient, based on extensive tests on various benchmarks. It is a trade-off between systematic calls to the dichotomic procedure and to the destructive approach.

Before reducing the size of the core, constraints are sorted with respect to their weight in the *dom/wdeg* heuristic. The first transition constraint is found using

Algorithm 7: CB(`XXcore`): combined algorithm

Input: a CSP (V, C)
Output: a MUC (V, C')

```
1  begin
2      (V, C' = {c_1, ..., c_|C'|}) ⟵ XXcore((V, C)) ;
3      C' ⟵ C' s.t. all constraints are sorted by decreasing weight ;
4      min ⟵ 1 ;
5      max ⟵ n ;
6      while (min ≠ max) do
7          med ⟵ (min + max)/2 ;
8          if (MAC((V, {c_1, ..., c_med})) proves unsatisfiability) then
9              max ⟵ med ;
10         else
11             min ⟵ med + 1 ;
12     C' ⟵ C' \ {c_min+1, ..., c_|C'|} ;
13     forall c ∈ C' do
14         if (MAC((V, C' \ {c})) proves unsatisfiability) then
15             C' ⟵ C' \ {c} ;
16     return (V, C') ;
17 end
```

the dichotomic approach. This first step takes advantage of the efficiency of the dichotomic technique and splits the set of constraints in two parts. Especially, it can allow us to drop "many" low-scores constraints that do not belong to the MUC. Then, the other transition constraints are discovered using the destructive approach. Clearly, this procedure exhibits the same worst-case complexity than the destructive approach, which requires a number of calls to MAC that is linear with respect to the size of core to be minimized. This new algorithm is called `CB(full-wcore)` (see Algorithm 7) since it "ComBines" the dichotomic and the destructive approaches, in opposition to `DC(wcore)`, `DC` and `DS` being shorthands for "dichotomic" and "destructive", respectively.

6. Experimental Results

In order to validate these hypotheses, extensive experimentations on various CSP benchmarks have been conducted. First, several benchmarks (`scen*`) provided by the CELAR (Centre Électronique de L'ARmement) that encode a Radio Link Frequency Assignment Problem[7] (RLFAP) have been considered. Also, various instances of the Quasi-group Completion Problem (`qcp`) and a so-called Geometric problem (`geo`) proposed by Rick Wallace have also been tested. In addition, ran-

domly generated instances have been considered. For instance, the `ehi` family is a CSP translation of randomly generated 3-SAT instances. Instances of the `composed` class, introduced in [23], are composed of several randomly-generated fragments, each of them being grafted to a main one by means of some additional random binary constraints. For more information about those various benchmarks, the reader is refered to [4].

In the following, a sample of typical results are provided; our software system and the complete experimental data are available at `http://www.cril.fr/~piette/MUC`.

`full-wcore`, `DC(full-wcore)`, `DS(full-wcore)` and `CB(full-wcore)` have been implemented in `C`. As the `DC(wcore)` technique from [17] is implemented in Java, it has been re-implemented -together with its `DS` and `CB` variants- in `C` in order to conduct a fair comparison. All tests have been performed on a Pentium IV 3GHz, under Linux Fedora Core 4.

In Tables 1 and 2, `wcore` and `full-wcore` are compared, together with the 3 minimization procedures applied for both of them, since they are intended to find one core that is not guaranteed to be minimal. For each CSP, we list the number of variables (#V), constraints (#C) and provide the number of constraints in the discovered core ($|UC|$), together with the CPU time spent in seconds to obtain it. Next, these cores have been minimized with the three aforementioned approaches. For each of these latter ones, the numbers of calls to a complete CSP-solving method are provided, distinguishing the calls leading to satisfiability from calls leading to unsatisfiability (#S and #U, respectively), the size of the extracted MUC ($|MUC|$), and the computation time. A time-out was set to 3600 seconds.

As the results show, exploring all the constraints at the filtering step even after a violated constraint has been discovered helps the size of the extracted cores to be reduced. Indeed, most of the time, the size of the core extracted by `full-wcore` is smaller than the size of the core delivered by `wcore`. For example, considering the `scen1_f9` benchmark, `full-wcore` delivered a core made of 358 constraints in 6.86 seconds, whereas `wcore` delivered a 1421-constraints core in 3.67 seconds.

Exploring all constraints instead of backtracking as soon as a violated constraint has been found does not necessarily slow down the whole computation process. Although more time can be needed to compute the approximations, it appears that in practice the global computation time is often decreased, mainly because more appropriate choices of branching variables can be performed as the *dom/wdeg* heuristic is guided in a better way towards problematic constraints. For example, the same core made of the 793 constraints has been extracted from `qcp-o15-h120-268-15` ; however, `full-wcore` only spent 100 seconds to compute it, whereas `wcore` needed more than twice this time.

The tentative enhancement of the minimization step also appears successful in practice. Although the `CB` approach does not deliver the best result for *every* CSP, its average behavior is very satisfactory. For example, when the approximation is bad (e.g. `scen11_f12`), the destructive approach proves very inefficient, whereas the

Instance	#C	#V	wcore		DC(wcore)			DS(wcore)			CB(wcore)		
			$\|UC\|$	time	(#S,#U)	$\|MUC\|$	time	(#S,#U)	$\|MUC\|$	time	(#S,#U)	$\|MUC\|$	time
scen11_f10	4103	680	711	11.48	(96,46)	16	17.4	(16,695)	16	588.91	(22,513)	16	153.06
scen11_f12	4103	680	610	9.3	(96 ,40)	16	14.42	(-,-)	-	*time out*	(21,427)	16	129.05
scen1_f9	5548	916	1421	3.67	(-,-)	-	*time out*	(25,1396)	25	323.43	(31,1260)	25	564.06
composed-75-1-2-2	624	33	529	0.18	(112,23)	14	1291.96	(13, 516)	13	27.13	(20 ,511)	14	33.18
composed-25-1-40-2	262	33	236	0.48	(79,22)	13	13.48	(13,223)	13	13.63	(18,181)	13	6.6
composed-25-1-40-4	262	33	239	0.23	(-,-)	-	*time out*	(14,225)	14	5.26	(17,184)	13	5.68
composed-25-1-80-0	302	33	240	0.09	(64,22)	11	12.94	(13,227)	13	4.78	(18,146)	14	4.42
dual_ehi-85-297-1	4112	297	209	0.26	(159,36)	34	2.43	(42,167)	42	58.49	(39,108)	36	2.08
dual_ehi-85-297-24	4105	297	206	0.22	(165,29)	34	2.37	(40,166)	40	2.72	(38,105)	34	1.79
dual_ehi-85-297-26	4102	297	179	0.25	(139,32)	30	2.13	(40,139)	40	2.38	(46,113)	41	1.99
dual_ehi-85-297-44	4130	297	178	0.2	(135,29)	29	1.99	(26,152)	26	2.34	(33,85)	29	1.43
dual_ehi-85-297-49	4124	297	192	0.21	(207,28)	41	2.82	(42,150)	42	2.52	(42,64)	40	1.24
dual_ehi-85-297-65	4116	297	156	0.2	(1099,0)	156	13.65	(156,0)	156	1.66	(163,0)	156	1.74
dual_ehi-85-297-7	4111	297	160	0.22	(151,27)	30	2.34	(33,127)	33	2.17	(40,107)	34	1.95
dual_ehi-90-315-6	4365	297	200	0.27	(261,36)	50	3.81	(51,149)	51	3.1	(52,80)	47	1.67
dual_ehi-90-315-94	4380	297	174	0.23	(158,44)	33	2.61	(43,131)	43	2.4	(47,111)	42	2.08
geo50.20.d4.75.70	451	50	424	140.62	(-,-)	-	*time out*	(-,-)	-	*time out*	(-,-)	-	*time out*
qcp-o15-h120-b-268-15	3150	225	793	237.6	(7146,0)	793	237.78	(793,0)	793	26.42	(802,0)	793	26.45
qcp-o20-h187-b-27-20	7600	400	389	2.6	(3120,0)	389	116.11	(389,0)	389	14.38	(397,0)	389	14.65
qcp-o20-h187-b-29-20	7600	400	958	6.3	(8631,0)	958	551.79	(958,0)	958	63.8	(967,0)	958	64.47

Table 1: `wcore` experimental results

Instance	#C	#V	`full-wcore`		`DC(full-wcore)`			`DS(full-wcore)`			`CB(full-wcore)`		
			$\|UC\|$	time	(#S,#U)	$\|MUC\|$	time	(#S,#U)	$\|MUC\|$	time	(#S,#U)	$\|MUC\|$	time
scen11_f10	4103	680	707	15.27	(97,45)	16	17.29	(16,691)	16	565.71	(22,512)	16	150.18
scen11_f12	4103	680	606	12.98	(97,38)	16	14.22	(-,-)	-	*time out*	(22,425)	16	130.42
scen1_f9	5548	916	358	6.86	(-,-)	-	*time out*	(25,333)	25	49.26	(28,242)	25	188.06
composed-75-1-2-2	624	33	529	3.04	(112,23)	14	1293.73	(13,516)	13	27.27	(20,511)	14	32.93
composed-25-1-40-2	262	33	227	0.59	(78,23)	13	25.58	(13,214)	13	4.63	(17,174)	13	4.78
composed-25-1-40-4	262	33	226	0.63	(-,-)	-	*time out*	(14,212)	14	4.64	(18,171)	13	5.14
composed-25-1-80-0	302	33	232	0.7	(63,22)	11	689.94	(13,219)	13	4.65	(17,139)	14	10.44
dual_ehi-85-297-1	4112	297	162	0.42	(169,30)	34	2.42	(40,122)	40	2.06	(37,59)	35	1.13
dual_ehi-85-297-24	4105	297	187	0.38	(181,38)	38	2.66	(42,145)	42	8.55	(41,98)	38	1.7
dual_ehi-85-297-26	4102	297	148	0.45	(151,37)	32	2.34	(36,112)	36	1.88	(42,91)	37	1.64
dual_ehi-85-297-44	4130	297	103	0.35	(624,0)	103	7.28	(103,0)	103	0.98	(109,0)	103	1.06
dual_ehi-85-297-49	4124	297	166	0.37	(224,39)	44	3.19	(39,127)	39	2.12	(43,77)	39	1.44
dual_ehi-85-297-65	4116	297	156	0.37	(1099,0)	156	13.69	(156,0)	156	1.67	(163,0)	156	1.76
dual_ehi-85-297-7	4111	297	109	0.37	(152,20)	32	2.23	(29,80)	29	1.41	(34,37)	31	0.85
dual_ehi-90-315-6	4365	297	153	0.38	(236,24)	47	3.31	(46,107)	46	2	(51,52)	46	1.25
dual_ehi-90-315-94	4380	297	146	0.36	(158,40)	32	2.56	(31,115)	31	2.13	(32,87)	30	1.54
geo50.20.d4.75.70	451	50	417	171.14	(-,-)	-	*time out*	(-,-)	-	*time out*	(-,-)	-	*time out*
qcp-o15-h120-b-268-15	3150	225	793	100.04	(7146,0)	793	238.21	(793,0)	793	26.59	(802,0)	793	26.56
qcp-o20-h187-b-27-20	7600	400	389	3.29	(3120,0)	389	116.35	(389,0)	389	14.32	(397,0)	389	14.74
qcp-o20-h187-b-29-20	7600	400	853	7.85	(7686,0)	853	453.98	(853,0)	853	52.04	(862,0)	853	52.78

Table 2: `wcore` and `full-wcore` experimental results

dichotomic one is appropriate. The hybridization schema allows lots of constraints to be eliminated thanks to the dichotomic step, and a MUC can be obtained within a reasonable time. On the contrary, `full-wcore` has extracted a set of constraints from the `qcp-o20-h187-9-20` instance, and this set is in fact one exact MUC. For this kind of "approximation", the dichotomic procedure exhibits its worst case, whereas the destructive one efficiently proves the minimality of the core by performing a linear number of satisfiability calls, which are in practice very fast compared to unsatisfiability calls. Once again, `CB` behaves very well, since it returns this MUC within less than 1 minute (like `DC`), but `DS` can ensure that this core is minimal in more than 7 minutes.

Moreover, on many benchmarks (see e.g. `dual_ehi-85-297-24`), `CB` appears to be the most efficient approach in order to compute one exact MUC. Actually, this new method takes advantage of both previous ones while, at the same time, it avoids their main drawbacks as much as possible. By heuristically removing a lot of constraints in a dichotomic way and by testing all the remaining ones step by step, the minimization procedure has been improved in many cases, and appears more robust than previously proposed ones.

Let us also note that different MUCs can be computed using those various methods. For instance, from the core extracted with `full-wcore` on (`dual_ehi-85-297-24`), `DS`, `DC` and `CB` extract MUCs of different sizes. In fact, a core can exhibit several MUCs. Thus, the order according to which constraints are removed can conduct us to compute one MUS instead of another one.

7. Related Works

So far, there have been only a few other research results about extracting MUCs from CSPs. First, there have been several works about the identification of (minimal) conflict sets of constraints (e.g. [27]) that are recorded during the search in order to perform various forms of intelligent backtracking, like dynamic backtracking [13] [21] or conflict-based backjumping [28]. In [18] a non-intrusive method was proposed to detect them, and can be interpreted as the seminal piece of work in this domain of research. However, there have been few other research works about the problem of extracting MUCs themselves. A method to find all MUCs from a given set of constraints has been presented in [16] and in [10], which corresponds to an exhaustive exploration of a so-called CS-tree but is limited by the combinatorial blow-up in the number of subsets of constraints. Other approaches are given in [25] and in [19], where an explanation that is based on the user's preferences is extracted. Also, the `PaLM` framework [20], implemented in the `Choco` constraint programming system [22], is an explanation tool that can explain why there is no solution involving the v_i value for a variable A. Moreover, in case of unsatisfiability, `PaLM` is able to provide a core, which is however not guaranteed to be a minimal one.

In the Boolean case, MUCs correspond to MUSes (*Minimally Unsatisfiable Subformulas*). Whereas `DC(wcore)` was the best current technique to discover MUCs,

the most current efficient technique for computing MUSes is based on a heuristic that exploits the number of times a clause has been critical during a failed local search for satisfiability [14]. Local search has also been proved very efficient in practice to compute the exhaustive set of MUSes of a propositional formula, when it is hybridized with a complete approach [15].

Let us also note that the problem of finding *Irreductible Infeasible Subsystems* (corresponds to MUCs in CSPs) has also been addressed in the mathematical programming domain, using specific approaches [8,2].

8. Conclusions and Perspectives

Pinpointing an irreducible set of infeasible constraints is a "harder" problem than solving a CSP itself, since the former problem belongs to the second level of the polynomial hierarchy, whereas the latter one is "only" NP-complete. However, delivering one MUC is a very valuable piece of information since it can help one to diagnose, understand and fix a CSP that does not have any solution.

In this paper, the currently most efficient technique to address this problem has been improved. The key points were to allow the MAC-based solver to check all constraints for infeasibility during the standard filtering process even after a first violated constraint has been discovered. It also relied on using the heuristic information already exploited in the first step to refine the approximation into a MUC.

This result opens many research and application perspectives. First, the proposed algorithm could be grafted to current CSP solvers, in order to provide them with a powerful explanation mechanism when a CSP does not have any solution at all. Second, a promising path for further research concerns the implementation side. In particular, the procedures described in this paper make repeated calls to a MAC solver on similar data, without reusing pertinent results from the previous calls. Improving the efficiency of the next call to MAC by exploiting the results of the previous calls clearly opens many new interesting issues from both the conceptual and computational points of view. Some interesting ideas in that direction can be found in [19]. Then, it should be noted that this study has been conducted with the goal of finding one MUC. However, a given CSP might exhibit several MUCs and the number of MUCs is even exponential in the worst case (it is in $\mathcal{O}(C_n^{n/2})$ where n is the number of constraints of the CSP). Clearly, the technique introduced in this paper can be used in a direct way to find a cover of MUCs, namely a series of MUCs that would render the CSP feasible if they were deleted from the initial CSP instance. To this end, it suffices to iterate the technique of this paper and drop successive MUCs as soon as they are discovered. However, MUCs can have non-empty intersections. In this respect, it should be noted that the approach presented in this paper requires the MAC-based solver to conduct a more systematic search for infeasible constraints at each instantiation step. In this respect, it could better apprehend the topology of all MUCs inside the CSP instance, and could be

an essential ingredient of a future method allowing one to deliver all MUCs, modulo a possible exponential blow-up restriction.

Acknowledgment

The authors thanks the anonymous reviewers for their very valuable suggestions and advices.

References

1. K. Apt. Principles of Constraint Programming. Cambridge University Press, 2003.
2. M.K. Atlihan and L. Schrage. Generalized filtering algorithms for infeasibility analysis. *Computers and Operations Research*, 35:1446–1464, 2008.
3. R. R. Bakker, F. Dikker, F. Tempelman, and P. M. Wognum. Diagnosing and solving over-determined constraint satisfaction problems. In *Proceedings of the 13th International Joint Conference on Artificial Intelligence (IJCAI'93)*, volume 1, pages 276–281. Morgan Kaufmann, 1993, 1993.
4. CSP Benchmarks. http://www.cril.fr/~lecoutre/research/benchmarks, 2008.
5. L. Bordeaux, Y. Hamadi, and L. Zhang. Propositional satisfiability and constraint programming: A comparative survey. *ACM Computing Surveys*, 38(4):12, 2006.
6. F. Boussemart, F. Hémery, C. Lecoutre, and L. Saïs. Boosting systematic search by weighting constraints. In *Proceedings of the 16th European Conference on Artificial Intelligence (ECAI'04)*, pages 146–150, 2004.
7. B. Cabon, S. de Givry, L. Lobjois, T. Schiex, and J. Warners. Radio link frequency assignment. *Constraints*, 4(1):79–89, 1999.
8. J.W. Chinneck. *Feasibility and Viability. In: Advances in Sensitivity Analysis and Parametric Programming*, volume 6, chapter 14. Kluwer Academic Publishers, Boston (USA), 1997.
9. A. Chmeiss and L. Saïs. Constraint satisfaction problems: Backtrack search revisited. In *Proceedings of the 16th IEEE International Conference on Tools with Artificial Intelligence (ICTAI'04)*, pages 252–257, 2004.
10. M. Garcia de la Banda, P. J. Stuckey, and J. Wazny. Finding all minimal unsatisfiable subsets. In *Proceedings of the Fifth ACM-SIGPLAN International Conference on Principles and Practice of Declarative Programming (PPDL'03)*, pages 32–43, 2003.
11. N. de Siqueira and J.F. Puget. Explanation-based generalisation of failures. In *Proceedings of the Eighth European Conference on Artificial Intelligence (ECAI'88)*, pages 339–344, 1988.
12. R. Dechter. *Constraint Processing*. Morgan Kaufmann Publishers Inc., 2003.
13. M. L. Ginsberg. Dynamic backtracking. *Journal of Artificial Intelligence Research*, 1:25–46, 1993.
14. É. Grégoire, B. Mazure, and C. Piette. Extracting MUSes. In *Proceedings of the 17th European Conference on Artificial Intelligence (ECAI'06)*, pages 387–391, Trento (Italy), 2006.
15. É. Grégoire, B. Mazure, and C. Piette. Boosting a complete technique to find MSS and MUS thanks to a local search oracle. In *Proceedings of the 20th International Joint Conference on Artificial Intelligence (IJCAI'07)*, volume 2, pages 2300–2305, 2007.
16. B. Han and S. Lee. Deriving minimal conflict sets by CS-Trees with mark set in diagnosis from first principles. In *IEEE Transactions on Systems, Man, and Cybernetics*, volume 29, pages 281–286, 1999.

17. F. Hemery, C. Lecoutre, L. Saïs, and F. Boussemart. Extracting MUCs from constraint networks. In *Proceedings of the 17th European Conference on Artificial Intelligence (ECAI'06)*, pages 113–117, 2006.
18. U. Junker. QuickXplain: Conflict detection for arbitrary constraint propagation algorithms. In *IJCAI'01 Workshop on Modelling and Solving problems with constraints (CONS-1)*, 2001.
19. U. Junker. QuickXplain: Preferred explanations and relaxations for over-constrained problems. In *Proceedings of the 19th National Conference on Artificial Intelligence (AAAI'04)*, pages 167–172, 2004.
20. N. Jussien and V. Barichard. The PaLM system: explanation-based constraint programming. In *Proceedings of TRICS: Techniques foR Implementing Constraint programming Systems, a post-conference workshop of CP'00*, pages 118–133, 2000.
21. N. Jussien, R. Debruyne, and P. Boizumault. Maintaining arc-consistency within dynamic backtracking. In *Principles and Practice of Constraint Programming*, pages 249–261, 2000.
22. F. Laburthe and The OCRE Project Team. Choco: implementing a CP kernel. In *Proceedings of TRICS: Techniques foR Implementing Constraint programming Systems, a post-conference workshop of CP'00*, 2000. http://www.choco-constraints.net.
23. C. Lecoutre, F. Boussemart, and F. Hemery. Backjump-based techniques versus conflict-directed heuristics. In *Proceedings of the 16th IEEE International Conference on Tools with Artificial Intelligence (ICTAI'04)*, pages 549–557, Boca Raton (USA), 2004.
24. A.K. Mackworth. Consistency in networks of relations. *Artificial Intelligence*, 8(1):99–118, 1977.
25. J. Mauss and M. M. Tatar. Computing minimal conflicts for rich constraint languages. In *Proceedings of the 15th European Conference on Artificial Intelligence (ECAI'02)*, pages 151–155, 2002.
26. C. H. Papadimitriou and D. Wolfe. The complexity of facets resolved. *Journal of Computer and System Sciences*, 37(1):2–13, 1988.
27. T. Petit, C. Bessière, and J.C. Régin. A general conflict-set based framework for partial constraint satisfaction. In *Proceedings of SOFT'03: Workshop on Soft Constraints held with CP'03*, 2003.
28. P. Prosser. Hybrid algorithms for the constraint satisfaction problems. In *Computational Intelligence*, volume 9(3), pages 268–299, 1993.
29. D. Sabin and E.C. Freuder. Contradicting conventional wisdom in constraint satisfaction. In *Proceedings of the Second International Workshop on Principles and Practice of Constraint Programming, PPCP'94*, pages 10–20, 1994.
30. T. Schiex and G. Verfaillie. Stubbornness: an enhancement scheme for backjumping and nogood recording. In *Proceedings of the 11th European Conference on Artificial Intelligence (ECAI'04)*, pages 165–169, Amsterdam (Netherlands), 1994.