

# A New Heuristic-based albeit Complete Method to Extract MUCs from Unsatisfiable CSPs

Éric Grégoire Bertrand Mazure Cédric Piette Lakdhar Saïs  
CRIL CNRS & IRCICA  
Université d'Artois  
Rue Jean Souvraz SP18  
F-62307 Lens Cedex France  
{gregoire,mazure,piette,sais}@cril.univ-artois.fr

## Abstract

*When a Constraint Satisfaction Problem (CSP) admits no solution, most current solvers express that the whole search space has been explored unsuccessfully but do not exhibit which constraints are actually contradicting one another and make the problem infeasible. In this paper, we improve a recent heuristic-based approach to compute infeasible minimal subparts of CSPs, also called Minimally Unsatisfiable Cores (MUCs). 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 improve the performance of the initial technique, which was the most efficient one until now.*

**Keywords:** CSP, constraint satisfaction problems, MUC, heuristic, artificial intelligence

## 1. Introduction

Constraint Satisfaction Problems (CSPs) form a very active domain of research and application of Artificial Intelligence techniques, that has found its way in so many problem domains (see e.g. [1], [6], [7]). 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.

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 CSPs. A MUC is a set of infeasible constraints that is minimal in the sense that dropping any of its member makes the set of remaining 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 complex 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 are actually conflicting 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 a set of clauses of a SAT instance is minimal unsatisfiable is D<sup>P</sup>-Complete, which belongs to the second level of the polynomial hierarchy [19].

So far, there have been only a few research results about extracting MUCs from CSPs, or MUSes (Minimally Unsatisfiable Subformulas) in the Boolean case. Currently, the most efficient technique for computing MUSes is based on a heuristic that exploits the number of times a clause has been falsified during a failed local search for satisfiability [11]. In the CSP framework, there have been many works about the identification of (minimal) conflict sets of constraints (e.g. [20]) that are recorded during the search in order to perform various forms of intelligent backtracking, like dynamic backtracking [10] [17] or conflict-based backjumping [21]. In [14] a non-intrusive method was proposed to detect them. However, there have been few research works about the problem of extracting MUCs itself. A method to find all MUCs from a given set of constraints has been presented in [12] and in [9], 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 [18] and in [15], where an explanation that is based on the user's preferences is extracted. Also, the PaLM framework [16] provides cores that are not guaranteed to be minimal.

Very recently a novel approach has been presented in [13], called DC(wcore), to compute MUCs. DC(wcore) appears to be the most efficient one for most CSPs classes. In particular, DC(wcore) appears to improve a previous method introduced in [2] to extract a MUC, that was proposed in the specific context of model-based diagnosis. It also proves more competitive than the use of the QuickXplain [14] method to compute MUCs. In this paper, we show that our variant technique improves DC(wcore) very often.

The paper is organized as follows. In the following section, the reader is provided with the necessary background about CSPs. In Section 2, Hemery and co-authors' DC(wcore) technique is briefly recalled. In Section 3, the main improvement that we introduce in order to enhance that technique is presented. In Section 4, extensive experimental results are described, showing the value of our proposed enhancement. In Section 5, we conclude by presenting some interesting paths for future research.

## 2. CSPs: technical background

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

### Definition 2.1.

A CSP is pair  $P = (V, C)$  where  
 (i)  $V$  is a finite set of  $n$  variables s.t. each variable  $x \in V$  has an associated 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 has an associated relation  $rel(c)$ , which contains the set of tuples allowed for the variables of its scope.

### 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 met. When  $P$  admits at least one solution,  $P$  is said *satisfiable*, else  $P$  is said *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  $C$  be a set of 7 constraints. In Figure 1, the CSP  $P = (V, C)$  is represented as a non-oriented graph, where each variable is a node and each constraint is an edge.  $P$  is unsatisfiable. Indeed, no assignment of values for all variables of  $V$  allows all constraints of  $C$  to be satisfied.

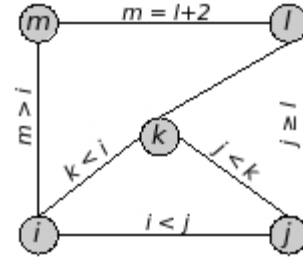


Figure 1. CSP: an example

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

### Definition 2.2.

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

1.  $P'$  is unsatisfiable
2.  $V' \subseteq V$  and  $C' \subseteq C$

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

1.  $P'$  is a core of  $P$
2. 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 2: no values for  $i, j$  and  $k$  can be found such that each value of a variable is smaller than the value of its neighbor. One constraint has to be removed if we want to fix this part of the problem.

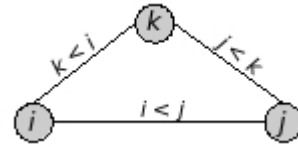


Figure 2. One MUC from the previous example

Solving a CSP is a NP-complete problem. There exist many complete and incomplete techniques to address it. Most "efficient" complete techniques rely on a progressive exploration of the search space, where at each step the set of currently instantiated variables is incremented and some filtering consistency checks are performed. Specifically, many current techniques make use of constraint propagation algorithms that propagate the values of the currently instantiated variables and filter the remaining domain of possible values for the other variables in such a way that the constraints could be satisfied. When one such domain becomes empty, this means that the lastly instantiated variable conducts the constraint to be infeasible. Hence, the algorithm needs to backtrack in order to consider another possible value for this lastly instantiated variable. One family of constraint

propagation algorithms is called MAC (Maintaining Arc Consistency) [22]. In the following we consider a complete CSP solver based on the MAC implementation by [5].

### 3. The DC(wcore) technique by Hemery et al.

Basically, the DC(wcore) technique by [13] is based on two successive steps, namely wcore and DC(wcore). 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 yield an actual MUC from this core. Our contribution is mainly an improvement of the first step. Accordingly, we shall focus only on this latter one; the second step remains unchanged (although we shall see that it can often be performed faster since it will often be operating on smaller cores, i.e. better approximations of MUCs).

The wcore procedure is based on the following findings. First, it is well-known [2] that when the unsatisfiability of a CSP instance is proved by a forward-checking based algorithm, this one can deliver a core of the CSP. It is formed of all *active* constraints, namely constraints whose satisfaction required at least one value from the acceptable domain of one of its variables to be removed during the filtering steps. Second, wcore makes use of the powerful heuristic *dom/wdeg* [4] to select the next variable to be assigned. This heuristic allows one to select the next variable to be instantiated among the variables that occurred most often in the constraints that were among the most often ones that were shown infeasible during the previous filtering steps.

Accordingly, the first step of DC(wcore) consists of a loop where calls to a complete MAC-based solver are iterated on a CSP instance while the number of active constraints decreases. Importantly, the aforementioned *dom/wdeg* heuristic that is used to dynamically guide the ordering of the variables to be instantiated is based on counters that are preserved from one call of MAC to the next one. These counters record the number of times a constraint has been found infeasible during the filtering steps.

A core is delivered when the last call to the MAC-based solver leads to a larger or equal number of active constraints. This core is the previous set of active constraints.

### 4. The MAC-based solver backtracks too early

The power of wcore relies on the efficiency of the MAC-based solver, which, among other things, increment the counters of the constraints that are infeasible at some filtering step. Accordingly, the solver resumes its exploration by focusing on “difficult” constraints first, thanks to the use of the *dom/wdeg* heuristic.

However, the goal of the MAC-based solver is to show in the most efficient manner that a CSP is unsatisfiable or exhibits at least one solution. We believe that such a solver could be modified when the goal is to get the smallest possible cores. More precisely, when the solver has shown that one constraint is infeasible due to the propagation of the value of the lastly instantiated variable, the solver backtracks. We believe that such a backtrack occurs too early. Other constraints are also perhaps violated in the same circumstances. It could prove useful to record all those violations too. This could be recorded through the counters associated to the constraints that will be used further on by the *dom/wdeg* ordering heuristic. Hence, we changed the 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 constraints are checked for feasibility under a given partial assignment of the set of variables. In this respect, the initial MAC-based solver collects a less systematic information about possible MUCs. We call our procedure full-wcore (weighting *all* falsified constraints) as a reference to the wcore (weigh core) name.

As our extensive experimental studies show, this simple modification of the algorithm improves the performance of both wcore and DC(wcore).

## 5. Experimental results

In order to validate our hypotheses, we have conducted extensive experimentations on various CSP benchmarks (e.g. [3]). In the following, we provide the reader with a sample of typical results. Full-wcore and DC(full-wcore) have been implemented in C and our tests were performed on Pentium IV 3GHz, under Linux Fedora Core 4. As the DC(wcore) technique from [13] was implemented in Java, we have re-implemented it in C in order to conduct a fair comparison.

In Table 1, wcore and full-wcore are compared. Let us recall here that both of them are intended to find one core that is not guaranteed to be minimal. We list the number of variables (#V), constraints (#C) of the CSP instances and provide for the discovered cores the number of constraints that they contain, together with the CPU time spent in seconds. The results are reported as (*number of constraint, time*). The time-out was fixed to 2000 seconds.

As the results show, exploring all the constraints at the filtering step even after an infeasible constraint has been discovered helps us in reducing the size of the extracted core. As a typical example, full-wcore delivered a core made of 490 constraints in 778 seconds for the “geo50.20.d4.75.94” instance whereas wcore delivered a 496-constraints core in 784 seconds.

Exploring all constraints instead of backtracking as soon as an infeasible constraint has been discovered can *a priori* slow down the whole computation process. However, it appears that in practice the global computation time is decreased, mainly because better

choices of branching variables can be performed as the *dom/wdeg* heuristic is oriented towards problematic constraints in an improved way.

<i>Instance</i>	<i>#V</i>	<i>#C</i>	<i>wcore</i>	<i>full-wcore</i>
composed-25-1-2-0	33	224	(214 , 0.05)	(211, 0.05)
composed-25-1-2-6	33	224	(198 , 0.04)	(192 , 0.05)
composed-25-1-2-7	33	224	(202 , 0.08)	(200 , 0.08)
composed-25-1-2-9	33	224	(194 , 0.06)	(191 , 0.06)
graph14_f28	916	4638	(3487 , 26.45)	(42 , 6.15)
graph2_f25	400	2245	(1581 , 19.98)	(1558 , 15.31)
geo50.20.d4.75.41	50	469	(457 , 329.07)	(449 , 333.80)
geo50.20.d4.75.52	50	483	(453 , 311.22)	(446 , 316.58)
geo50.20.d4.75.57	50	555	(546 , 498.61)	(546 , 373.40)
geo50.20.d4.75.94	50	496	(496 , 784.56)	(490 , 778.75)
scen1_f9	916	5548	(695 , 3.20)	(691 , 3.35)
scen2_f25	200	1235	(584 , 13.90)	(276 , 15.91)
scen3_f11	400	2760	(659 , 21.94)	(424 , 25.18)
ehi-85-297-11	297	4100	(158 , 0.47)	(157 , 0.47)
ehi-85-297-13	297	4102	(193 , 0.54)	(188 , 0.56)
ehi-85-297-16	297	4105	(206 , 0.67)	(195 , 0.75)
lsq_dg_4	16	88	(63 , 0.01)	(56 , 0.01)
qcp-15-120-100-15	225	3150	(70 , 0.09)	(68 , 0.09)
qcp-15-120-90-15	225	3150	(813 , 5.04)	(813 , 4.48)

Table 1. *wcore* vs. *full-wcore*

In Table 2, the complete *DC(wcore)* technique is compared with a combination of our technique with the *DC* fine-tune second step, namely *DC(full-wcore)*.

Again, better results are obtained most often. This can be understood easily: a better approximation of a MUC reduces the required time for *DC* to converge on an exact MUC. Notice that, on some benchmarks (e.g. *lsq\_dg\_4* and *qcp-15-120-100-15*), different MUCs are obtained with both methods. On this last instance, it is worth noting that both *full-wcore* and *wcore* have extracted an exact MUC, directly.

<i>Instance</i>	<i>#V</i>	<i>#C</i>	<i>DC(wcore)</i>	<i>DC(full-wcore)</i>
qa_3	10	40	(15 , 0.90)	(15 , 0.90)
bh-4-4-0001	112	1262	(50 , 11.17)	(50 , 10.50)

<i>Instance</i>	<i>#V</i>	<i>#C</i>	<i>DC(wcore)</i>	<i>DC(full-wcore)</i>
bh-4-4-0002	112	1262	(50 , 10.72)	(50 , 9.97)
lsq_dg_4	16	88	(33 , 0.57)	(24 , 1.14)
qcp-15-120-100-15	225	3150	(70 , 3.07)	(68 , 2.90)
qcp-15-120-90-15	225	3150	(813 , 421.14)	(813 , 420.83)

Table 2. *DC(wcore)* vs. *DC(full-wcore)*

## 6. 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, we have improved the currently most efficient technique to address this problem. The key point was to allow the MAC-based solver to check all constraints for infeasibility during the standard filtering process even after a first infeasible constraint has been discovered.

This result opens many research and applicative perspectives. First, our 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. Then, it should be noted that our 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  $O(C^{n/2})$  where  $n$  is the number of constraints of the CSP). Clearly, our technique 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 our technique 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 our approach 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.

## Acknowledgements

This paper has been supported in part by the EC under a FEDER grant and the *Conseil Régional du Nord/Pas-de-Calais*.

## References

- [1] Apt K., *Principles of Constraint Programming*, Cambridge University Press, 2003.
- [2] Baker R.R., Dikker F., Tempelman F. and Wognum P.M., « Diagnosing and solving over-determined constraint satisfaction problems », *Proceedings of the 13th International Joint Conference on Artificial Intelligence (IJCAI'93)*, 276-281, 1993.
- [3] <http://cpai.ucc.ie/05/Benchmarks.html>
- [4] Boussemart F., Hemery F., Lecoutre C. and Saïs L., « Boosting systematic search by weighting constraints », *Proceedings of the 16th European Conference on Artificial Intelligence (ECAI'04)*, 146-150, 2004.
- [5] Chmeiss A. and Saïs L., « Constraint Satisfaction Problems: Backtrack Search Revisited », *Proceedings of the 16th IEEE International Conference on Tools with Artificial Intelligence (ICTAI'04)*, 252-257, 2004.
- [6] *Proceedings of the twelfth International Conference on Principles and Practice of Constraint Programming (CP'06)*, Springer, 2006.
- [7] Dechter R., *Constraint Processing*, Morgan Kaufmann, 2003.
- [8] de Siqueira J.L. and Puget J-F., « Explanation-based generalisation of failures », *Proceedings of the 8th European Conference on Artificial Intelligence (ECAI'88)*, 339-344, 1988.
- [9] Garcia de la Banda M., Stuckey P.J. and Wazny J., « Finding all minimal unsatisfiable subsets », *Proceedings of the Fifth ACM-SIGPLAN International Conference on Principles and Practice of Declarative Programming (PPDP'03)*, 2003.
- [10] Ginsberg M., « Dynamic backtracking », *Journal of Artificial Intelligence Research*, vol. 1, 25-46, 1993.
- [11] Grégoire É., Mazure B. and Piette C., « Extracting MUSes », *Proceedings of the 17th European Conference on Artificial Intelligence (ECAI'06)*, August 2006.
- [12] Han B. and Lee S.-J., « Deriving minimal conflicts sets by CS-trees with mark set in diagnosis from first principles », *IEEE Transactions on Systems, Man, and Cybernetics*, 29(2), 281-286, 1999.
- [13] Hemery F., Lecoutre C., Saïs L. and Boussemart F., « Extracting MUCs from Constraint Networks », *Proceedings of the 17th European Conference on Artificial Intelligence*, August 2006.
- [14] Junker U., « QuickXplain: conflict detection for arbitrary constraint propagation algorithms », *Proceedings of the 17th International Joint Conference on Artificial Intelligence (IJCAI'01) Workshop on modelling and solving problems with constraints*, 2001.
- [15] Junker U., « QuickXplain: preferred explanations for over-constrained problems », *Proceedings of the Nineteenth National Conference on Artificial Intelligence (AAAI'04)*, 167-172, 2004.
- [16] Jussien N. and Barichard D., « The PaLM system: explanation-based constraint programming », *Proceedings of TRICS'00 : workshop held with CP'00*, 118-133, 2000.
- [17] Jussien N., Debruyne R. and Boismault P., « Maintaining arc-consistency within dynamic backtracking », *Proceedings of the 6th International Conference on Principles and Practice of Constraint Programming (CP'00)*, 249-261, 2000.
- [18] Mauss J. and Tatar M., « Computing minimal conflicts for rich constraint languages », *Proceedings of the 16th European Conference on Artificial Intelligence (ECAI'02)*, 151-155, 2002.
- [19] Papadimitriou C.H. and Wolfe D., « The complexity of facets resolved », *Journal of Computer and System Sciences*, 37, 2-13, 1988.
- [20] Petit T., Bessière C. and Régim J.C., « A general conflict-set based framework for partial constraint satisfaction », *Proceedings of SOFT'03: Workshop on Soft Constraints held with CP'03*, 2003.
- [21] Prosser P., « Hybrid algorithms for the constraint satisfaction problems », *Computational Intelligence*, 9(3), 268-299, 1993.
- [22] Sabin D. and Freuder E., « Contradicting conventional wisdom in constraint satisfaction », *Proceedings of the Second Workshop on Principles and Practice of Constraint Programming (CP'94)*, 10-20, 1994.
- [23] Tsang E., *Foundations of Constraint Satisfaction*, Academic Press, 1993.