MUST: Provide a Finer-Grained Explanation of Unsatisfiability

Éric Grégoire, Bertrand Mazure, and Cédric Piette

CRIL-CNRS & IRCICA
Université d'Artois
rue Jean Souvraz SP18
F-62307 Lens Cedex France
{gregoire,mazure,piette}@cril.fr

Abstract. In this paper, a new form of explanation and recovery technique for the unsatisfiability of discrete CSPs is introduced. Whereas most approaches amount to providing users with a minimal number of constraints that should be dropped in order to recover satisfiability, a finer-grained alternative technique is introduced. It allows the user to reason both at the constraints and tuples levels by exhibiting both problematic constraints and tuples of values that would allow satisfiability to be recovered if they were not forbidden. To this end, the Minimal Set of Unsatisfiable Tuples (MUST) concept is introduced. Its formal relationships with Minimal Unsatisfiable Cores (MUCs) are investigated. Interestingly, a concept of shared forbidden tuples is derived. Allowing any such tuple makes the corresponding MUC become satisfiable. From a practical point of view, a two-step approach to the explanation and recovery of unsatisfiable CSPs is proposed. First, a recent approach proposed by Hemery *et al.*'s is used to locate a MUC. Second, a specific SAT encoding of a MUC allows MUSTs to be computed by taking advantage of the best current technique to locate Minimally Unsatisfiable Sub-formulas (MUSes) of Boolean formulas. Interestingly enough, shared tuples coincide with protected clauses, which are one of the keys to the efficiency of this SAT-related technique. Finally, the feasibility of the approach is illustrated through extensive experimental results.

Keywords: CSP, constraint networks, explanation, unsatisfiability, MUC, MUS, MUST.

1 Introduction

In this paper, we are concerned with unsatisfiable finite CSPs, namely finite Constraint Satisfaction Problems for which no solution exists. Recent approaches to explain such a form of unsatisfiability have been defined at the constraints level. For example, Hemery *et al.* [1] have proposed an approach called `DC(wcore)` to detect Minimally Unsatisfiable Cores (MUCs) of CSPs, i.e. unsatisfiable subsets of constraints of the initial CSP that are such that dropping any one the constraints allows the resulting subset to become satisfiable. In this paper, a finer-grained alternative technique is introduced. Indeed, dropping constraints can be too much destructive. Finer-grained information

could be provided to the user, allowing him (her) not only to pinpoint the causes of unsatisfiability at the constraints level, but also detect the tuples of values that are forbidden by the constraints and that would lead to satisfiability if they were allowed by those constraints.

In this respect, the contribution of this paper is twofold. On the one hand, a *Minimally Unsatisfiable Set of Tuples* (MUST) concept is introduced: it is aimed at encompassing the aforementioned notion of tuples that can allow satisfiability to be regained. The formal relationships between MUSTs and MUCs are investigated. Interestingly, a concept of shared forbidden tuples is derived. Allowing any shared tuple makes the corresponding MUC become satisfiable. On the other hand, a two-step approach to the explanation and recovery of unsatisfiable CSPs is proposed. A specific SAT encoding of a MUC allows MUSTs to be computed by taking advantage of the best current technique to locate Minimally Unsatisfiable Sub-formulas (MUSes) of Boolean formulas. Interestingly enough, shared tuples coincide with protected clauses [2], which are one of the keys to the efficiency of this SAT-related technique.

Accordingly, the paper is organized as follows. First, some basic definitions about CSPs and MUCs are provided. In section 3, MUSTs are introduced and linked to MUCs in section 4. Next, an original two-step approach to explain and recover from unsatisfiable CSPs is described. In section 5, a SAT-related approach to compute MUSTs and shared forbidden tuples of a MUC is introduced. The feasibility of the approach is illustrated through extensive experimental results in section 6. Section 7 compares the contribution presented in this paper with the current existing works. In the conclusion, some interesting paths for future research are described.

2 Background: CSPs and MUCs

In this section, the reader is provided with basic concepts about CSPs and MUCs.

Definition 1. *A finite Constraint Satisfaction Problem (in short, CSP) is a pair* $P = \langle V, C\rangle$ *where*

1. V *is a finite set of* n *variables* $\{v_1, \ldots, v_n\}$ *s.t. each variable* $v_i \in V$ *has an associated finite instantiation domain, denoted* $dom(v_i)$*, which contains the set of possible values for* v_i*,*
2. C *is a finite set of* m *constraints* $\{c_1, \ldots, c_m\}$ *s.t. each constraint* $c_j \in C$ *involves a subset of variables of* V*, called scope and denoted* $Var(c_j)$*, and has an associated relation* $R(c_j)$*, which contains the set of tuples allowed for the variables of its scope.*

Definition 2. *Solving a CSP* $P = \langle V, C\rangle$ *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 satisfiable else* P *is called unsatisfiable.*

In this paper, it will prove useful to adopt an alternative but equivalent definition for CSPs, expressed in terms of *forbidden* tuples of values.

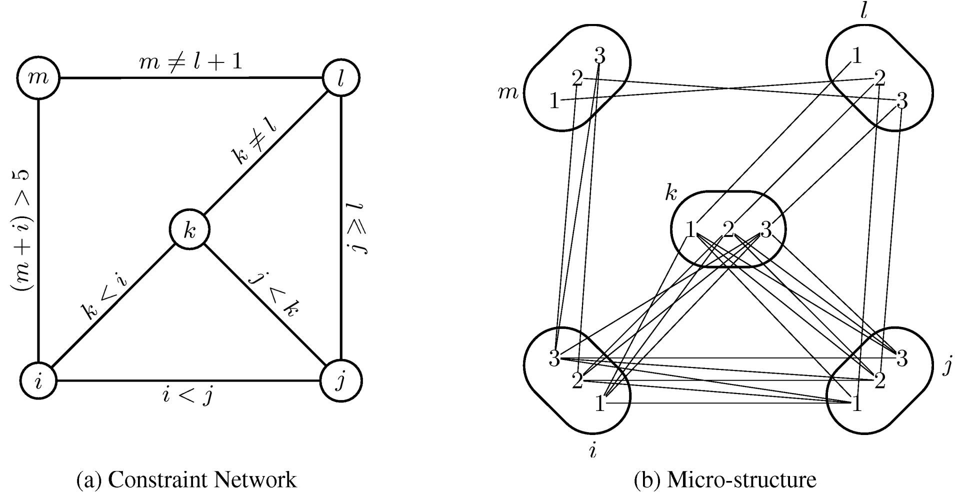

Fig. 1: Graph-representations of Example 1

Definition 3. *Let $\langle V, C\rangle$ be a CSP and let $c \in C$ s.t. $Var(c) = \{v_c^1, \ldots, v_c^l\}$. A forbidden tuple of values is a member of $dom(v_c^1) \times \cdots \times dom(v_c^l)$ s.t. $R(c)$ is not satisfied. The set of forbidden tuples of values for a constraint c is denoted $T(c)$.*

Accordingly, CSPs can be redefined as follows.

Definition 4. *A finite CSP is a pair $P = \langle V,C\rangle$ where*

1. *V is a finite set of n variables $\{v_1, \ldots, v_n\}$ s.t. each variable $v_i \in V$ has an associated finite instantiation domain, denoted $dom(v_i)$, which contains the set of possible values for v_i,*
2. *C is a finite set of m constraints $\{c_1, \ldots, c_m\}$ s.t. each constraint $c_j \in C$ involves a subset of variables of V, called scope and denoted $Var(c_j)$, and is given a relation $T(c_j)$, which contains the set of tuples forbidden for the variables of its scope.*

Accordingly, P will also be denoted $\langle V, \{(Var(c_1), T(c_1)), (Var(c_2), T(c_2)), \ldots, (Var(c_n), T(c_n))\}\rangle$.

Definition 5. *A constraint $c \in C$ of the CSP $P = \langle\{v_1, \ldots, v_n\}, C\rangle$ is falsified by an assignment $A \in dom(v_1) \times \cdots \times dom(v_n)$ iff the projection of A on $Var(c)$ is included in $T(c)$.*

In the following, *(forbidden) tuple* will be a shorthand for *forbidden tuple of values*, and binary CSPs will be considered only, namely CSPs where constraints involve two variables. Using binary CSPs do not restrict the impact of our works, since it is well-known that every discrete CSP can be reduced into a binary one, in polynomial time.

Example 1. Let V be $\{i,j,k,l,m\}$ where each variable has the same domain $\{1,2,3\}$. Let C be a set of 7 constraints. In Figure 1a, the CSP $P = \langle V, C\rangle$ is represented by

a so-called *constraint network*, namely a non-oriented graph, where each variable is a node and each constraint is an edge, labelled with its corresponding relation. It is also useful to represent a CSP by its *micro-structure*, which is a graph where the values of each variable are listed, and edges between values represent forbidden tuples. The micro-structure of this example is depicted in Figure 1b.

When a CSP is infeasible, it exhibits at least one *Minimally Unsatisfiable Core*, or MUC. A MUC is a subpart of a CSP that is unsatisfiable and that does not contain any proper subpart that is also unsatisfiable.

Definition 6. *Let* $P = \langle V, C \rangle$ *and* $P' = \langle V', C' \rangle$ *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. In the previous example, P is unsatisfiable. Indeed, P contains the MUC $P' = \langle V, \{i < j, j < k, k < i\} \rangle$: no assignment of values for i, j and k can be found such that these three constraints are satisfied, and dropping any constraint leads to satisfiability.

Computing one MUC for an unsatisfiable CSP is an NP-hard problem. More precisely, checking whether a constraint belongs to a MUC or not is in Σ_2^p [3]. Moreover, the number of MUCs inside a CSP can be exponential in the worst-case; it is in $\mathcal{O}(C_m^{m/2})$, where m is the number of constraints in the CSP. It should be noted that MUCs can share non-empty intersections. Several techniques have been proposed in the literature to compute MUCs, the `DC(wcore)` approach introduced recently by Hemery *et al.* [1] is claimed by its authors to be the most efficient one, most often.

3 MUSTs

Restoring the satisfiability of a CSP can be achieved through restoring the satisfiability of each of its MUCs. A natural way to break the unsatisfiability of a MUC is to drop any of its constraints. However, dropping some constraints as a whole can appear too much destructive. On the contrary, we might prefer to *weaken* one or several constraints, instead of removing them. One way to do that is to provide the user with some forbidden tuples that should be allowed in order to recover satisfiability.

Let us introduce the following MUST concept and study to which extent it can be a first good step in that direction. A MUST (*Minimally Unsatisfiable Set of Tuples*) of an unsatisfiable CSP P is an unsatisfiable CSP P' that is such that the sets of forbidden tuples of its constraints are subsets of the corresponding sets w.r.t. P and such that allowing any of the tuples of P' will render P' satisfiable.

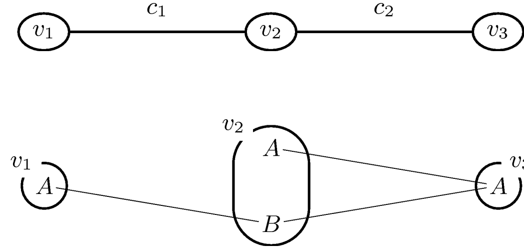

Fig. 2: Graphical representations of Example 3

Definition 7. *Let* $P = \langle V, \{(Var(c_1), T(c_1)), \ldots, (Var(c_m), T(c_m))\}\rangle$ *be an unsatisfiable CSP. The CSP* $P' = \langle V, \{(Var(c_1), T'(c_1)), \ldots, (Var(c_m), T'(c_m))\}\}\rangle$ *is a* MUST (Minimally Unsatisfiable Set of Tuples) *of* P *if and only if :*

1. P' *is unsatisfiable*
2. $\forall i$ *s.t.* $1 \leq i \leq m$, $T'(c_i) \subseteq T(c_i)$
3. $\forall i$ *s.t.* $1 \leq i \leq m$, $\forall\, T''(c_i) \subset T'(c_i)$,
   $\langle V, \{(Var(c_1), T'(c_1)), ..., (Var(c_i), T''(c_i)), ..., (Var(c_m), T'(c_m))\}\rangle$ *is satisfiable*

Hence, a MUST can be interpreted as a tentative way to explain infeasibility at a lower level of abstraction than the constraints one does. At this point, it is important to note that this definition for a MUST of a CSP P does not require that allowing one of the forbidden tuples of the MUST will make P become satisfiable. Indeed, it is easy to prove that an unsatisfiable CSP can exhibit an exponential number of MUSTs in the worst case and that their set-theoretic intersection can be empty. Thus, the removal of *one* forbidden tuple might not be enough to regain satisfiability. Furthermore, as the following example shows, tuples contained in a MUST might even not take part into the actual cause of the infeasibility of the CSP.

Example 3.

Let $P = \langle V, C\rangle$ s.t. $V = \{v_1, v_2, v_3\}$, with $dom(v_1) = dom(v_3) = \{A\}, dom(v_2) = \{A, B\}$, and $C = \{c_1 = (\{v_1, v_2\}, \{(A, B)\}), c_2 = (\{v_2, v_3\}, \{(A, A), (B, A)\})\}$. In Figure 2, both the graph and the micro-structure of this CSP are given. Clearly, P is unsatisfiable and exhibits only one MUC, made of the c_2 constraint, since this one prevents any assignment from being valid between v_2 and v_3. On the contrary, considering a lower level of abstraction, we see that P exhibits two MUSTs, namely:

- $P_{M_1} = \langle V, \{(\{v_1, v_2\}, \{(A, B)\}), \{(\{v_2, v_3\}, \{(A, A)\})\}\}\rangle$
- $P_{M_2} = \langle V, \{(\{v_2, v_3\}, \{(A, A), (B, A)\})\}\rangle$

P_{M_1} is a MUST that contains tuples from both constraints. It does not correspond to any MUC of P. P_{M_2} is a MUST that is also a MUC of P. Moreover, P_{M_1} contains the

only forbidden tuple linking v_1 and v_2, which does not participate to the unsatisfiability of P.

Although these results might sound negative, in the next section it is shown that MUSTs form an adequate concept to explain unsatisfiability at the tuples level, provided that MUSTs are considered within MUCs.

4 MUSTs within MUCs

It is well-known that any unsatisfiable CSP exhibits at least one MUC (which can be the CSP itself). Unsurprisingly, any unsatisfiable CSP also exhibits at least one MUST. Indeed, MUCs are unsatisfiable CSPs, which ensures that at least one MUST can be extracted from any MUC of a CSP.

Proposition 1. *At least one MUST can be extracted from any unsatisfiable CSP.*

Proof. *Let $P = \langle V, C\rangle$ be an unsatisfiable CSP. Assume that P does not contain any MUST. Thus, P itself is not a MUST: hence there exists a forbidden tuple of P s.t. allowing it gives rise to another unsatisfiable CSP containing no MUST, namely $\exists c \in C, \exists t \in T(c)$ such that $P^1 = \langle V, (C \backslash c) \cup (Var(c),\ T(c) \backslash t)\rangle$ is also unsatisfiable, and does not exhibit any MUST. Iterating this reasoning, it is easily proved by induction that this would lead to the existence of a CSP $P' = \langle V, \emptyset\rangle$ that should be unsatisfiable, whereas such a CSP is clearly satisfiable.* $\square$

Moreover, stronger relations link MUCs and MUSTs, as shown by the following proposition.

Proposition 2. *Let P be a MUC that contains m constraints. There exists at least m tuples s.t. allowing any one of them makes P regain satisfiability. These tuples belong to all MUSTs of P.*

Proof. *Since $P = \langle V, C\rangle$ is a MUC, whenever any of its m constraints c_i is dropped, the resulting CSP is satisfiable. Let A be an assignment that satisfies $P' = \langle V, C \backslash c_i\rangle$. c_i is violated by A: indeed, in the opposite case, P would be satisfiable and would not be a MUC. The projection of A on $Var(c_i)$ is included in $T(c_i)$, according to definition 5. Thus, removing this forbidden tuple is sufficient to make P feasible. The same argument can be used for any of the m constraints of P. Thus, there exists at least m tuples such that omitting one of them makes the MUC regain feasibility. Clearly, these tuples necessarily belong to all the sources of unsatisfiability, and consequently to every MUST of the MUC. Hence, the set-theoretic intersection of all MUSTs of P contains at least m tuples.* $\square$

In this respect, some tuples allow the unsatisfiability of the MUC to be “broken”, simply by allowing any of them. These tuples necessarily belong to all sources of unsatisfiability of the MUC, and consequently to every MUST of the MUC. Accordingly, it is thus not possible to discover two MUSTs of a MUC with an empty set-theoretic intersection. Tuples belonging to every MUST of a MUC P will be called *shared tuples* of P.

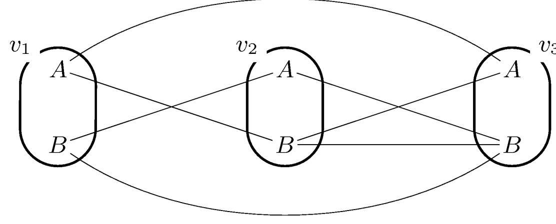

Fig. 3: Micro-structure of Example 4

Definition 8. *Let P be a MUC. The shared tuples of P are the forbidden tuples belonging to all MUSTs of P.*

Allowing any shared tuple allows the corresponding MUC to be broken. Moreover, computing a MUST of a MUC P delivers a super-set of the set of shared tuples.

Example 4. Let a CSP $P = \langle\{v_1, v_2, v_3\}, \{c_1, c_2, c_3\}\rangle$ s.t.

1. $\forall i \in \{1, 2, 3\}, dom(v_i) = \{A, B\}$
2. $c_1 = (\{v_1, v_2\}, \{(A, B), (B, A)\})$
3. $c_2 = (\{v_2, v_3\}, \{(A, B), (B, A), (B, B)\})$
4. $c_3 = (\{v_1, v_3\}, \{(A, A), (B, B)\})$

The micro-structure of P is given in Figure 3. Let us note that P is a MUC since P is unsatisfiable and dropping any of its constraints yields a satisfiable CSP. P exhibits two MUSTs; their micro-structures are given in Figure 4. By considering the set-theoretic intersection of those MUSTs, one can obtain the shared tuples of P. The shared tuples are represented using boldface edges in Figure 5, while the other tuples are represented using dotted lines. Clearly, allowing one shared tuple allows us to restore the satisfiability of both MUSTs and their corresponding MUC. However, allowing any other forbidden tuple of these MUSTs does not guarantee the satisfiability of the initial MUC to be restored. This example also shows us that the CSP formed with the shared tuples of a MUC is not necessarily unsatisfiable.

These last results plead for a two-step policy for the explanation of unsatisfiability in terms of MUCs, MUSTs and shared tuples. Indeed, looking for MUSTs in the general case does not seem the most promising approach since MUSTs can coincide with no MUC at all. On the contrary, an interesting approach would require to search for MUCs as a first step. MUCs provide explanations of unsatisfiability that are expressed in terms of a minimal number of constraints. The user can drop one such constraint to break the unsatisfiability of the MUC. Alternatively, he (she) could rather search for MUSTs corresponding to the discovered MUC. More precisely, if he (she) manages to discover shared tuples, the user would be provided with a set of forbidden tuples that is such that allowing just one such tuple is *enough* to break the unsatisfiability of the MUC. In such a way, the user would be given the ability to weaken problematic constraints instead of dropping one whole constraint. Such a policy is to be iterated until all (remaining) MUCs of the resulting CSP have been addressed.

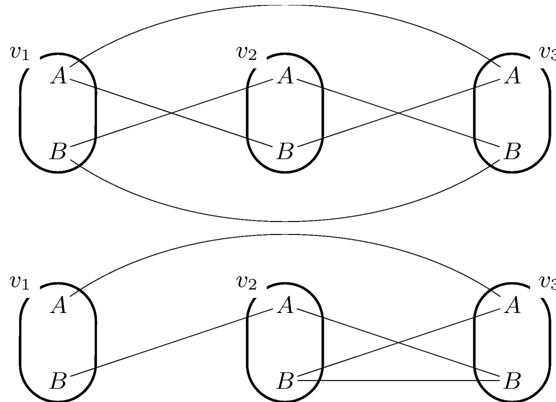

Fig. 4: Micro-structure of the two MUSTs of Example 4

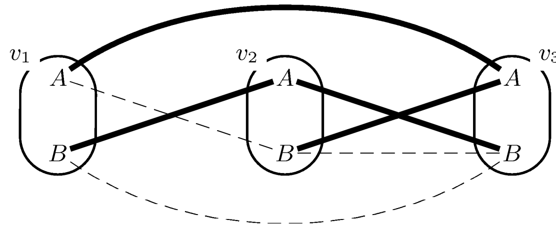

Fig. 5: Shared tuples of Example 4

Several algorithms have been proposed to compute one MUC. Thus, the next issue that is to be addressed is how both MUSTs and shared tuples could be computed within a MUC. In the following, it is shown that, modulo a specific SAT encoding, shared tuples exactly coincide with so-called protected clauses [2], which play a central role in the efficiency of the currently most efficient technique to compute MUCs in the Boolean case, namely MUSes (Minimally Unsatisfiable Sub-formulas). In this respect, a Boolean translation of MUCs that allows us to benefit from the efficiency of this powerful computational technique will be provided.

5 Using `OMUS` to compute MUSTs and shared tuples

Computing a MUST could be performed through several traditional techniques from the CSP and the operational research domains, such as the destructive, additive or dichotomic minimization procedures [4]. However, those approaches would deliver MUSTs, only. The computation of shared tuples would require either a linear number of additional step-by-step tests of satisfiability, or to consider all solutions of each relaxation obtained by removing one constraint from the computed MUC. On the contrary, an approach allowing both a MUST and shared tuples to be computed at the same time is introduced in this section.

When one MUC has been obtained trough e.g. Hemery *et al.*'s `DC(wcore)` technique [1], it is translated inside the Boolean framework in such a way that the computation of MUSTs and shared tuples of the MUC is achieved through the computation of MUSes, namely minimally unsatisfiable sets of clauses of a CNF formula. The selected translation schema is a form of *direct encoding* [5] that consists in encoding each domain value of each variable by a different Boolean variable C_{v_i}. Accordingly, the number of variables in the Boolean framework is given by the sum of the sizes of the domains of the variables of the MUC. Let $P = \langle V, C \rangle$ be a MUC, the following clauses are then created.

1. *at-Least-one clauses* ensure that at least one possible value for each variable v_i is selected in a solution
   $C_{v_1} \vee C_{v_2} \vee ... \vee C_{v_m}$ $\forall v \in V$ with $dom(v) = \{v_1, v_2, ..., v_m\}$
2. *at-Most-one clauses* ensure that at most one value is selected for each variable
   $\neg C_{v_a} \vee \neg C_{v_b}$ $\forall v \in V$ $\forall (v_a, v_b) \in dom(v) \times dom(v)$
3. *Conflict clauses* encode forbidden tuples
   $\neg C_{v_i} \vee \neg C_{v_j}$ $\forall c \in C$ $\forall (v_i, v_j) \in T(c)$

This form of encoding has been adopted because it allows a forbidden tuple to be translated into a unique clause. Moreover, the "at-Most-one clauses" are actually not added to the generated formula. Indeed, those clauses have been proved optional [6]; moreover, omitting them enables us to ensure that minimality is preserved in both frameworks. Thus, each MUS of the generated Boolean formula corresponds to a MUST of the infeasible CSP. Computing one MUST amounts to computing one MUS in this Boolean framework, provided that every *at-Least-one* clauses belong to the MUS.

More precisely, we make use of the `OMUS` technique by Grégoire *et al.* [2], which is currently one of the most efficient complete techniques to discover one MUS inside an unsatisfiable SAT instance. One key to the efficiency of this approach is the concept of *protected clauses*. Interestingly enough, protected clauses appear to encode shared tuples.

The `OMUS` technique is based on the so-called critical clause concept, which is a clause that is falsified under a given assignment of values and that is such that satisfying it by a minimal change of the assignment will always conduct at least another clause to be falsified in its turn. Roughly, the `OMUS` technique is a two-step approach. First, a superset of a MUS is computed by iterating the computation of the number of times each clause is critical during local search runs and by dropping clauses with the lowest scores. Second, a fine-tune process allows an exact MUS to be delivered. Interestingly, each time there is just one clause that is not satisfied w.r.t. some variables assignment during the local search run, it is marked and never be dropped from the formula. These clauses are called *protected*, and belong to all MUSes of the Boolean formula. According to this encoding, protected clauses coincide with shared tuples of the translated MUC. These tuples are thus delivered together with the MUST, without any computing overhead. Furthermore, the first step of this local-search-based algorithm sometimes delivers an unsatisfiable set of protected clauses, which forms a MUS, and the second step of the algorithm is avoided.

Algorithm 1: `direct_encode`

Input: a CSP: $\langle V, C \rangle$
Output: a set of clauses Σ viewed as a CNF formula

1 **begin**
2 $\quad \Sigma \longleftarrow \emptyset$;
3 $\quad$ **foreach** $v_i \in V$ **do**
4 $\quad\quad \Sigma \longleftarrow \Sigma \bigcup \{ \bigvee_{\forall j \in dom(v_i)} x_{ij} \}$; /* "At-least-one" clauses */
5 $\quad$ **foreach** $c \in C$ **do**
6 $\quad\quad$ **foreach** $t \in T(c)$ **do**
7 $\quad\quad\quad \Sigma = \Sigma \bigcup \{ \bigvee_{\substack{\forall (v_i \in Var(c) \text{ and } j \in dom(v_i)) \\ \text{s.t. } j \text{ is the forbidden value of } v_i \text{ in } t}} \neg x_{ij} \}$;
8 $\quad$ **return** Σ ;
9 **end**

Algorithm 2: `mus2must`

Input: a MUS: Σ and a MUC: $\langle V, \{(Var(c_1), T(c_1)), ..., (Var(c_m), T(c_m))\} \rangle$
Output: a MUST: $\langle V, \{(Var(c_1), T'(c_1)), ..., (Var(c_m), T'(c_m))\} \rangle$

1 **begin**
2 $\quad$ **foreach** $c \in C$ *s.t.* $C = \{(Var(c_1), T(c_1)), ..., (Var(c_m), T(c_m))\}$ **do**
3 $\quad\quad T'(c) \longleftarrow \emptyset$;
4 $\quad\quad$ **foreach** $t \in T(c)$ **do**
5 $\quad\quad\quad$ **if** *((* $\bigvee_{\substack{\forall (v_i \in Var(c) \text{ and } j \in dom(v_i)) \\ \text{s.t. } j \text{ is the forbidden value of } v_i \text{ in } t}} \neg x_{ij}) \in \Sigma$*)* **then**
6 $\quad\quad\quad\quad T'(c) \longleftarrow T'(c) \cup \{t\}$;
7 $\quad$ **return** $\langle V, \{(Var(c_1), T'(c_1)), ..., (Var(c_m), T'(c_m))\} \rangle$;
8 **end**

6 Experimental studies

In order to assess the practical value of these techniques, an algorithm called `MUSTER` (MUST-ExtRaction) has been implemented and run on various benchmarks from the last CSP competition [7]. This software makes thus use of (a `C` variant of) Hemery *et al.*'s `DC(wcore)` technique to extract a MUC from the considered CSP. Then, the MUC is converted into a Boolean clausal formula according to the aforementioned *direct-encoding* described in Algorithm 1. A MUS is then computed thanks to Grégoire *et al.*'s `OMUS` procedure [2]. As described above, there is a one-to-one correspondence between this MUS and a MUST from the CSP; this latter one is delivered, together with detected shared tuples, using a simple translation procedure described in Algorithm 2. The `MUSTER` method is summarized in Algorithm 3.

All experiments have been conducted on a Pentium IV, 3Ghz under Linux Fedora Core 5. A significant sample of results are given on Table 1, which contains 3 main columns, namely *Instance*, *Extracted MUC* and *Extracted MUST*. The first column provides information about the considered unsatisfiable CSP: namely, the benchmark

Algorithm 3: MUSTER

Input: a CSP: $\langle V, C\rangle$
Output: a MUST: $\langle V, \{(Var(c'_1), T''(c'_1)), ..., (Var(c'_m), T''(c'_m))\}\rangle$

```
1 begin
2   ⟨V,C'⟩ ⟵ DC(wcore)(⟨V,C⟩) ; /* ⟨V,C'⟩ is a MUC s.t. C' ⊆ C */
    /*          and C' = {(Var(c'_1),T(c'_1)),...,(Var(c'_m),T(c'_m))} */
3   Σ_CNF ⟵ direct_encode(⟨V,C'⟩) ;
4   Σ_MUS ⟵ OMUS(Σ_CNF) ;                     /* Σ_MUS ⊆ Σ_CNF */
5   ⟨V,{(Var(c'_1),T''(c'_1)),...,(Var(c'_m),T''(c'_m))}⟩ ⟵ mus2must(Σ_MUS , ⟨V,C'⟩);
    /*              ∀ 1 ≤ i ≤ m   T''(c'_i) ⊆ T(c'_i)              */
6   return ⟨V,{(Var(c'_1),T''(c'_1)),...,(Var(c'_m),T''(c'_m))}⟩ ;
7 end
```

name, the number of involved constraints and the number of forbidden tuples that the constraints represent. In the second one, the main information about the MUC computed by DC(wcore) is given: namely, the number of constraints of the MUC, the number of tuples that they represent, and the computing time in seconds that was spent. Finally, the third main column provides the main information about the computed MUST: namely, its number of tuples, the number of discovered shared tuples, and the computing time in seconds that was spent. A time-out was set to 3 hours of CPU time.

First, let us note that a MUST was extracted within a reasonable amount of time for most benchmarks, which represent hard to solve problems. For instance, a MUC made of 13 constraints is discovered for the composed-75-1-2-1 CSP, which contains 624 contraints. This MUC forbids 845 tuples. Actually, this set can be reduced to just 344 tuples, which form one MUST of the benchmark. Moreover, the user is provided with a set of 104 tuples such that if one of these latter tuples is allowed, then the unsatisfiable part of the CSP represented by the MUC is fixed. Similar results were obtained for e.g. scen11_f10, which is an instance of the famous Radio Link Frequency Assignment Problem (RLFAP). This latter benchmark involves almost 800,000 forbidden tuples. However, only some of these them really participate to the unsatisfiability of the CSP. Indeed, a MUC that contains less than 5,000 tuples is exhibited, and this one has been reduced into a MUST made of 3,077 tuples. In this MUST, allowing one tuple among the 2,728 discovered shared ones is enough to allow the MUC to regain feasibility.

Obviously enough, due to the high-level computational complexity of the addressed problem, we cannot expect our approach to solve all problems within a reasonable amount of time. For example, although a MUST was extracted for the Queen-Knight problem qk_8_8_5_add, MUSTER was not able to deliver any of its shared tuples, although we know that these latter tuples are contained in the 10149 tuples that form the computed MUST. Let us also note that the same MUC and MUST have been discovered for both qk_8_8_5_add and qk_8_8_5_mul problems. This is easily explained: those problems result from various combinations between the 8 queens problem and 5 knights one. Since the 5 knights problem is not feasible, a same explanation of unsatisfiability can be delivered for all combinations of this problem with other CSPs.

Instance			Extracted MUC			Extracted MUST		
name	#con	#tuples	#con	#tuples	time (s)	#tuples	#st[1]	time (s)
composed-25-1-2-0	224	4,440	14	910	10.72	354	119	13.08
composed-25-1-2-1	224	4,440	15	975	9.09	339	59	17.47
composed-25-1-25-8	247	4,555	9	585	8.85	259	116	6.25
composed-75-1-2-1	624	10,440	13	845	66.42	344	104	10.85
composed-75-1-2-2	624	10,440	14	910	66.74	376	48	14.44
composed-75-1-25-8	647	10,555	16	1,040	59.09	461	51	23.69
composed-75-1-80-6	702	10,830	11	715	61.48	278	55	8.17
composed-75-1-80-7	702	10,830	16	1,040	379.85	420	75	17.23
composed-75-1-80-9	702	10,830	12	780	86.16	306	89	9.01
qk_10_10_5_add	55	48,640	5	47,120	19.68	24,855	0	3081.1
qk_10_10_5_mul	105	49,140	5	47,120	1.29	24,855	0	2812.99
qk_8_8_5_add	38	19,624	5	18,800	3.33	10,149	0	544.7
qk_8_8_5_mul	78	19,944	5	18,800	0.66	10,149	0	531.24
graph2_f25	2,245	145,205	43	4,498	427.36	2470	1,516	426.05
qa_3	40	800	15	583	0.32	203	152	8.32
dual_ehi-85-297-14	4,111	102,234	40	1,145	3.35	311	142	40.26
dual_ehi-85-297-15	4,133	102,433	35	1,083	4.03	310	172	25.85
dual_ehi-85-297-16	4,105	102,156	36	1,032	4.68	301	159	29.05
dual_ehi-85-297-17	4,102	102,112	43	1,239	4.83	348	172	42.21
dual_ehi-85-297-18	4,120	102,324	33	972	3.48	271	141	30.4
dual_ehi-90-315-21	4,388	108,890	37	1,120	3.16	354	129	35
dual_ehi-90-315-22	4,368	108,633	41	1,218	4.57	410	187	43.69
dual_ehi-90-315-23	4,375	108,766	29	835	2.86	251	131	12.23
dual_ehi-90-315-24	4,378	108,793	31	974	4.57	315	167	25.42
dual_ehi-90-315-25	4,398	108,974	38	1,106	3.89	375	179	30.41
scen6_w2	648	513,100	7	8,020	53.78	4,872	2,953	1107.39
scen6_w1_f2	319	274,860	21	21,146	488.64	-	-	*time out*
scen11_f10	4,103	738,719	16	4,588	164.27	3,077	2,728	438.26
scen11_f12	4,103	707,375	16	4,588	122.12	3,053	2,728	419.83

Table 1: Extracting a MUST

Finally, let us comment on the number of shared tuples. Proposition 2 ensures that a MUC made of m constraints contains at least m shared tuples. However, the number of shared tuples is often larger in practice. For example, a MUC made of 43 constraints is extracted from `graph2_f25`: so, at least 43 shared tuples could have been expected. Actually, 1,516 such tuples were delivered, enabling the user to just select one constraint among 43 ones, and then just select and allow one tuple among an average of 35 candidate ones, to regain feasibility.

7 Related works

The approach introduced in this paper can be interpreted as a refinement of explanation techniques that provide users with MUCs in case of unsatisfiability. There have been

[1] #st: #shared tuples

only a few research results about extracting MUCs from CSPs, or MUSes in the Boolean case. In this respect, the approach in this paper takes advantage of one of the most often efficient technique to compute MUCs [1] and of the currently most efficient technique compute MUSes [2], in order to deliver MUSTs and shared tuples.

In the CSP framework, there have been several other works about the identification of (minimal) conflict sets of constraints (e.g. [8]) that are recorded during the search in order to perform various forms of intelligent backtracking, like dynamic backtracking [9] [10] or conflict-based backjumping [11]. In [12] a non-intrusive method was proposed to detect them. However, there have been few 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 [13] and in [14], 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 [15] and in [16], where an explanation that is based on the user's preferences is extracted. Also, the `PaLM` framework [17], implemented in the constraint programming system `Choco` [18], is an explanation tool that can answer for instance the question: why is there no solution that contains the value v_i for some variable A? Moreover, in case of unsatisfiability, `PaLM` is able to provide a core, but this one is not guaranteed to be minimal. The `DC(wcore)` approach that is used in this paper appears to improve a previous method introduced in [19] 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` [12] method to compute MUCs.

In the Boolean framework, the problem of extracting a MUS from an unsatisfiable CNF formula has also received much attention. In [20], Bruni has proposed an approach that approximates MUSes by means of an adaptative search guided by clauses hardness. Zhang and Malik have described in [21] a way to extract MUSes by learning nogoods involved in the derivation of the empty clause by resolution. In [22], Lynce and Marques-Silva have proposed a complete and exhaustive technique to extract one smallest MUS of a SAT instance. Together with Mneimneh, Andraus and Sakallah [23], the same authors have also proposed an algorithm that makes use of iterative max-SAT solutions to compute such smallest unsatisfiable subsets of clauses. Oh and her co-authors have presented in [24] a Davis, Putnam, Logemann and Loveland DPLL-oriented approach that is based on a marked clause concept to allow one to approximate MUSes. Let us also mention a complete approach by Liffiton and Sakallah [25], recently improved by a non-standard use of local search [26], that attempts to compute the exhaustive set of MUSes of a propositional formula.

Finally, let us note that the problem of finding an *Irreductible Infeasible Subsystem* has also been the subject of specific research efforts in mathematical programming [4][27].

8 Conclusions and perspectives

These results open many interesting research perspectives.

An unsatisfiable CSP can exhibit several MUCs that can share non-empty set-theoretic intersections. Regaining feasibility through permitting shared tuples thus re-

quires the `MUSTER` process to be iterated until all remaining MUCs in the resulting CSP have been addressed. Clearly, the order according to which MUCs are addressed influences the tuples that are delivered to the users. In this respect, it could be fruitful to develop order-independent techniques that directly deliver sets of tuples that would make the CSP feasible if these latter tuples were allowed. In this respect, recent results by Grégoire *et al.* [28] that allows covers of MUSes to be computed, i.e. sets of MUSes that cover all basic infeasibility causes, could be exploited in that direction. The concept of shared tuples would have to be revised accordingly, and specific computational techniques would have to be devised in order to compute that.

Although it appears to be highly competitive in practice, our technique to compute shared tuples remains uncomplete since it does not guarantee that all shared tuples will be delivered. Another interesting path for future research would consist in developping complete techniques that guarantee, modulo a possible exponential blow-up, the computation of *all* shared tuples.

In this paper, basic formal results about MUSTs have been provided. Clearly, much more can be done in this respect. Variant definitions coud be provided, in particular variants that would not rely on the MUC concept to address infeasibility.

Also, several MUSTs can be inter-dependent within a CSP: studying the formal properties of their relationships is also a promising path for future research.

References

1. Hemery, F., Lecoutre, C., Saïs, L., Boussemart, F.: Extracting MUCs from constraint networks. In: Proceedings of the 17th European Conference on Artificial Intelligence (ECAI'06). (2006) 113–117.
2. Grégoire, É., Mazure, B., Piette, C.: Extracting MUSes. In: Proceedings of the 17th European Conference on Artificial Intelligence (ECAI'06). (2006) 387–391
3. Eiter, T., Gottlob, G.: On the complexity of propositional knowledge base revision, updates and counterfactual. Artificial Intelligence **57** (1992) 227–270
4. Chinneck, J. In: Chapter 14: Feasibility and Viability, In: Advances in Sensitivity Analysis and Parametric Programming. Volume 6. Kluwer Academic Publishers, Boston (USA) (1997)
5. de Kleer, J.: A comparison of ATMS and CSP techniques. In: Proceedings of the Eleventh International Joint Conference on Artificial Intelligence (IJCAI'89). (1989) 290–296
6. Walsh, T.: SAT v CSP. In: Proceedings of the 6th International Conference on Principles and Practice of Constraint Programming (CP'00). (2000) 441–456
7. CSPcomp: CSP competition http://cpai.ucc.ie/06/competition.html
8. Petit, T., Bessière, C., Régin, J.: A general conflict-set based framework for partial constraint satisfaction. In: Proceedings of SOFT'03: Workshop on Soft Constraints held with CP'03. (2003)
9. Ginsberg, M.L.: Dynamic backtracking. Journal of Artificial Intelligence Research **1** (1993) 25–46
10. Jussien, N., Debruyne, R., Boizumault, P.: Maintaining arc-consistency within dynamic backtracking. In: Principles and Practice of Constraint Programming. (2000) 249–261
11. Prosser, P.: Hybrid algorithms for the constraint satisfaction problems. In: Computational Intelligence. Volume 9(3). (1993) 268–299

12. Junker, U.: QuickXplain: Conflict detection for arbitrary constraint propagation algorithms. In: IJCAI'01 Workshop on Modelling and Solving problems with constraints (CONS-1). (2001)
13. Han, B., Lee, S.: 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. (1999) 281–286
14. de la Banda, M., Stuckey, P.J., Wazny, J.: Finding all minimal unsatisfiable subsets. In: Proceedings of the Fifth ACM-SIGPLAN International Conference on Principles and Practice of Declarative Programming (PPDL'03). (2003) 32–43
15. Mauss, J., Tatar, M.M.: Computing minimal conflicts for rich constraint languages. In: Proceedings of the 15th European Conference on Artificial Intelligence (ECAI'02). (2002) 151–155
16. Junker, U.: QuickXplain: Preferred explanations and relaxations for over-constrained problems. In: Proceedings of the 19th National Conference on Artificial Intelligence (AAAI'04). (2004) 167–172
17. Jussien, N., Barichard, V.: The PaLM system: explanation-based constraint programming. In: Proceedings of TRICS: Techniques foR Implementing Constraint programming Systems, a post-conference workshop of CP'00. (2000) 118–133
18. Laburthe, F., Team, T.O.P.: 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.
19. Bakker, R.R., Dikker, F., Tempelman, F., Wognum, P.M.: Diagnosing and solving overdetermined constraint satisfaction problems. In: Proceedings of the 13th International Joint Conference on Artificial Intelligence (IJCAI'93). Volume 1., Morgan Kaufmann, 1993 (1993) 276–281
20. Bruni, R.: Approximating minimal unsatisfiable subformulae by means of adaptive core search. Discrete Applied Mathematics **130**(2) (2003) 85–100
21. Zhang, L., Malik, S.: Extracting small unsatisfiable cores from unsatisfiable boolean formula. In: Sixth international conference on theory and applications of satisfiability testing (SAT'03). (2003)
22. Lynce, I., Marques-Silva, J.: On computing minimum unsatisfiable cores. In: International Conference on Theory and Applications of Satisfiability Testing. (2004)
23. Mneimneh, M.N., Lynce, I., Andraus, Z.S., Marques Silva, J.P., Sakallah, K.A.: A branch-and-bound algorithm for extracting smallest minimal unsatisfiable formulas. In: International Conference on Theory and Applications of Satisfiability Testing (SAT'05). (2005) 467–474
24. Oh, Y., Mneimneh, M., Andraus, Z., Sakallah, K., Markov, I.: AMUSE: a minimally-unsatisfiable subformula extractor. In: Proceedings of the 41th Design Automation Conference (DAC 2004). (2004) 518–523
25. Liffiton, M., Sakallah, K.: On finding all minimally unsatisfiable subformulas. In: Proceedings of SAT'05. (2005) 173–186
26. Grégoire, É., Mazure, B., Piette, C.: Boosting a complete technique to find MSSes and MUSes thanks to a local search oracle. In: Proceedings of the 20th International Joint Conference on Artificial Intelligence (IJCAI'07). Volume 2. (2007) 2300–2305
27. Atlihan, M., Schrage, L.: Generalized filtrering algorithms for infeasibility analysis. Computers and Operations Research (2007) to appear
28. Grégoire, É., Mazure, B., Piette, C.: Local-search extraction of MUSes. Constraints Journal: Special issue on Local Search in Constraint Satisfaction **12**(3) (2007) to appear