Restoring CSP Satisfiability with MaxSAT*

Inês Lynce[1] and Joao Marques-Silva[2]

[1] IST/INESC-ID, Technical University of Lisbon, Portugal
ines@sat.inesc-id.pt
[2] CSI/CASL, University College Dublin, Ireland
jpms@ucd.ie

Abstract. The extraction of a Minimal Unsatisfiable Core (MUC) in a Constraint Satisfaction Problem (CSP) aims to identify a subset of constraints that make a CSP instance unsatisfiable. Recent work has addressed the identification of a Minimal Set of Unsatisfiable Tuples (MUST) in order to restore the CSP satisfiability with respect to that MUC. A two-step algorithm has been proposed: first, a MUC is identified, and second, a MUST in the MUC is identified. This paper proposes an integrated algorithm for restoring satisfiability in a CSP, making use of an unsatisfiability-based MaxSAT solver. The proposed approach encodes the CSP instance as a partial MaxSAT instance, in such a way that solving the MaxSAT instance corresponds to identifying the smallest set of tuples to be removed from the CSP instance to restore satisfiability. Experimental results illustrate the feasibility of the approach.

Key words: constraint satisfaction problems, minimal unsatisfiable cores, minimal set of unsatisfiable tuples, maximum satisfiability

1 Introduction

The identification of unsatisfiable problem instances poses a few questions, including knowing why the instance is unsatisfiable and how it can be repaired to become satisfiable. For example, configuring a product may not always result in a feasible configuration, but in that case the customer would be pleased to receive explanations to understand what made the configuration unfeasible or alternatively to receive some hints about how to make it feasible. Such feedback is in general expected to be as precise as possible, i.e. to identify a minimal reason of unsatisfiability and to minimize the impact of restoring satisfiability.

If one considers Boolean satisfiability (SAT), then these questions are answered identifying Minimal Unsatisfiable Subformulas (MUSes) and obtaining Maximum Satisfiability (MaxSAT) solutions. Similarly, in the context of the Constraint Satisfaction Problem (CSP) these questions are answered by identifying Minimal Unsatisfiable Cores (MUCs) and obtaining Maximum CSP (MaxCSP) solutions. However, as recently pointed out [7], in CSPs one may consider unsatisfiable sets of tuples instead of unsatisfiable sets of constraints. The first is

* Research partially funded by FCT project SHIPs (PTDC/EIA/64164/2006).

considered to be a finer-grained explanation as unsatisfiable sets of tuples may eventually not contain as many tuples as unsatisfiable sets of constraints.

Let us get back again to the configuration of a product. Suppose you have two configuration requirements for your new car: (1) it has to be a sports car and (2) you want it to have the most inexpensive type of seats. It turns out that the car model that you have selected does not allow such configuration. So you may be simply told that those two requirements are not compatible or, much better, that a sports car must have leather seats which is incompatible with having cloth seats. The latter explanation is certainly more precise. Also, it will be easier for you to repair the initial configuration either keeping all but one of the characteristics of a sports car (leather seats) or forgetting about inexpensive seats.

This paper addresses the problem of repairing an unsatisfiable CSP instance by removing the smallest number of tuples. This has in general less impact than removing constraints, and in the worst case the same impact. A MaxSAT encoding is used to solve the problem of repairing a CSP for which constraints are defined as conflicts, with no need of first extracting a MUC. Using an unsatisfiability-based MaxSAT solver has the advantage of identifying unsatisfiable sets of tuples while restoring satisfiability.

The paper is organized as follows. The next section introduces the preliminaries, alongside with illustrative examples. Next, the new approach for restoring satisfiability in a CSP using MaxSAT is described. Experimental results show that the proposed approach is feasible. Finally, the paper concludes and points out future research directions.

2 Preliminaries

2.1 CSPs, MUCs and MUSTs

In what follows we assume that CSP variables have finite domains and that the constraints over variables correspond to conflicts rather than supports.

Definition 1. *(CSP) A CSP instance is a triple* (X, D, C) *where* $X = \{x_1, \ldots, x_n\}$ *is a set of* n *variables,* $D = \{d(x_1), \ldots, d(x_n)\}$ *is a set of* n *domains, where each domain* $d(x)$ *corresponds to the domain of variable* $x \in X$*, and* $C = \{c_1, \ldots, c_m\}$ *is a set of* m *constraints. Each constraint* $c \in C$ *is a pair* $c = (S, R)$ *where* S *is a sequence of variables of* X*, called the constraint scope, and* R *is a* $|S|$*-ary relation over* D*, called the constraint relation.*

An assignment to a CSP instance is a set $\{(v_1, a_1), \ldots, (v_k, a_k)\}$ where $\{v_1, \ldots, v_k\} \subseteq X$ is a set of variables and $a_i \in d(v_i)$ for all i with $1 \leq i \leq k$. In case $k = n$ the assignment is said to be complete. Assuming that a constraint relation specifies *disallowed* assignments, i.e. the conflicts, a solution to a CSP instance is a complete assignment such that no assignment to a constraint scope is a member of the corresponding constraint relation. A CSP instance for which there is a solution is said to be *satisfiable*; otherwise it is said to be *unsatisfiable*.

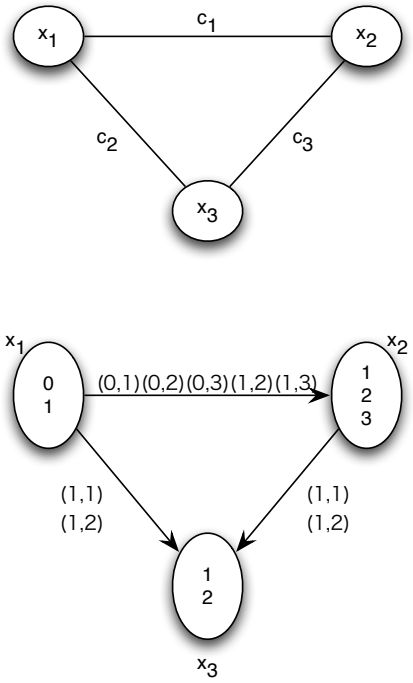

Fig. 1. Graphical representation of the CSP instance from Example 1

Example 1. (CSP) Let P be the CSP instance graphically represented in Figure 1. Formally, $P = (X, D, C)$, with $X = \{x_1, x_2, x_3\}$, $D = \{d(x_1) = \{0, 1\}, d(x_2) = \{1, 2, 3\}, d(x_3) = \{1, 2\}\}$ and $C = \{c_1, c_2, c_3\} = \{((x_1, x_2), \{(0, 1), (0, 2), (0, 3), (1, 2), (1, 3)\}), ((x_1,\ x_3), \{(1, 1), (1, 2)\}), ((x_2, x_3), \{(1, 1), (1, 2)\})\}$. Figure 1 represents P taking into account its variables and constraint scopes (top) and its variable domains and constraint relations (bottom). The arrows denote the order of the variable values in the conflict tuples.

We should note that the constraints of the previous example could be alternatively defined in terms of *allowed* assignments, i.e. the supports. In some cases such approach has the advantage of requiring less tuples. For example, c_1 would become $((x_1, x_2), \{(1, 1)\})$ instead. But in other cases it may be the contrary. For example, c_2 would become $((x_2, x_3), \{(2, 1), (2, 2), (3, 1)\,(3, 2)\})$.

Definition 2. *(UC) An unsatisfiable core (UC) of an unsatisfiable CSP $P = (X, D, C)$ is a CSP instance $P' = (X', D', C')$ such that (1) P' is unsatisfiable and (2) $X' \subseteq X$, $D' \subseteq D$ is the set of domains of variables in X' and $C' \subseteq C$ is the set of constraints with their scopes in X'.*

An *unsatisfiable core* (UC) of an unsatisfiable CSP instance is a CSP instance that is a subset of the former one with respect to its variables, domains and constraints, and is still unsatisfiable. An unsatisfiable CSP instance has at least one UC corresponding to itself.

An unsatisfiable core is expected to be minimal as it should be as precise as possible when identifying the causes of unfeasibility, thus extracting minimal unsatisfiable cores (MUCs). An UC has at least one MUC, and in case it is unique then the MUC is equal to the UC. Removing one constraint per MUC restores satisfiability. A smaller number of constraints suffices when constraints are shared by different MUCs.

Definition 3. *(MUC) A minimal unsatisfiable core (MUC) of an unsatisfiable CSP instance P is an UC of P, say P', such that removing any of the constraints of P' makes the resulting CSP instance satisfiable.*

Providing the user with explanations of unsatisfiability of CSPs has been first addressed with QuickXplain [8]. Moreover, the extraction of MUCs in CSPs has been recently made feasible [5]. The proposed algorithm performs successive runs of a complete backtracking search, using weights, in order to isolate an inconsistent subset of constraints.

Another approach for explaining unfeasibility consists in dealing with the tuples of constraint relations (in the sequel called tuples) rather than constraints defined as a pair with their scopes and relations [7]. This approach has clear advantages. Instead of identifying a set of constraints as the causes of unsatisfiability, a set of tuples is identified. Consequently, restoring satisfiability when considering tuples implies removing tuples rather than constraints, which may have a minor impact on the encoding. This assumes that, when making the encoding of a CSP instance, errors have been introduced when encoding a few tuples rather than when encoding a few constraints (including all its tuples).

Definition 4. *(MUST) A minimal unsatisfiable set of tuples (MUST) of an unsatisfiable CSP $P = (X, D, C)$, with $C = \{(S_1, R_1), \ldots, (S_m, R_m)\}$, is a CSP $P' = (X', D', C')$ such that (1) P' is unsatisfiable, (2) $\forall (S', R') \in C' \; \exists (S, R) \in C : S' = S \wedge R' \subseteq R$, $X' \subseteq X$ is the set of variables in S' and $D' \subseteq D$ is the set of domains of variables in X' and (3) removing any tuple from R' s.t. $(S', R') \in C'$ makes the resulting CSP instance satisfiable.*

It is worth mentioning that tuples in a MUST do not necessarily belong to constraints in a MUC. An illustrative example is given bellow.

Example 2. (MUC and MUST) Figure 2 illustrates MUCs and MUSTs of the CSP instance P introduced in Example 1. P has two MUCs: one with c_1 and c_2 and another one with c_1 and c_3. An intuitive explanation is the following: c_1 implies the assignment $\{(x_1, 1), (x_2, 1)\}$, whereas c_2 *forbids* the assignment $\{(x_1, 1)\}$ and c_3 *forbids* the assignment $\{(x_2, 1)\}$. Any of these MUCs contains seven tuples. Several MUSTs can be identified in P but only two of them are illustrated in the figure (bottom). One of them (left) corresponds to the smallest MUST and contains only five tuples. The point is that not all tuples from

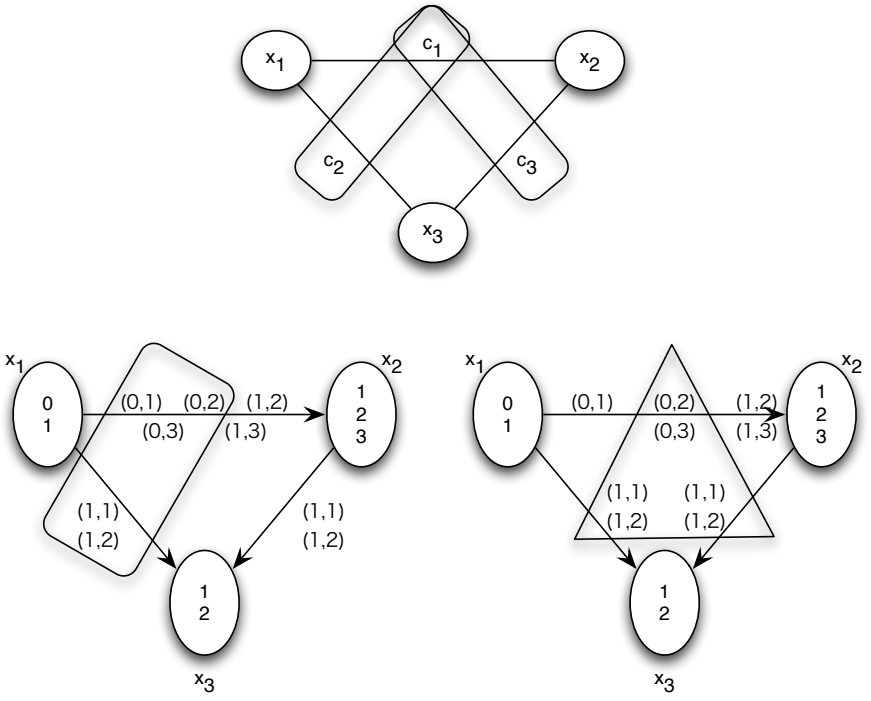

Fig. 2. MUCs and MUSTs of the CSP instance from Example 1

constraint c_1 are required to make the instance unsatisfiable when considering also c_2. The other MUST (right) contains six tuples with the particularity of covering all the constraints.

Satisfiability of a CSP can be restored by removing one tuple of each MUST. Similarly to MUCs, a smaller number of tuples may be removed when tuples are shared by different MUSTs. For the running example, removing a tuple representing either the conflict between assignments $x_1 = 0$ and $x_2 = 2$ or assignments $x_1 = 0$ and $x_2 = 3$ suffices to restore satisfiability.

2.2 SAT, MUSes and MaxSAT

We assume the reader is familiar with the SAT problem that consists in deciding whether there exists a satisfying assignment to a set of Boolean variables such that a given CNF formula is satisfied. A CNF formula is a conjunction of clauses, where a clause is a disjunction of literals and a literal is a Boolean variable or its complement (a positive or a negative literal, respectively).

A large body of research in SAT has been devoted to explaining unfeasibility of CNF formulas in the last decade. These explanations have many different

applications and are of the utmost importance in model checking using SAT solvers. For each iteration, if the call to the SAT solver returns no solution, then an explanation is extracted to be incorporated in the next iteration. Clearly, this explanation should be as precise as possible. Explanations including irrelevant clauses are expected to have a negative impact in the subsequent iterations.

The design of tools to identify unsatisfiable subformulas (USes) in SAT formulas has been early addressed by Bruni [2] who used an heuristic approach to identify a subset of clauses that are still unsatisfiable. Later on, Zhang [18] and Goldberg [4] proposed a similar idea of extracting unsatisfiability proofs based on the resolution steps taken for deriving the empty clause. These resolution steps are implicitly behind the conflict clauses added to the formula as a result of the analysis of conflicts during the search. The identification of all the minimal unsatisfiable subformulas (MUSes) in a systematic way was first developed by Liffiton [11, 12] and later used for identifying the smallest MUS in an efficient way [10]. The identification of MUSes has also shown to be competitive recurring to a local search procedure as a preprocessing step to reduce the number of clauses that may be part of the MUS [6].

The MaxSAT problem consists in finding the largest number of clauses that can be satisfied in a CNF formula. The partial MaxSAT problem distinguishes between hard and soft clauses: hard clauses must be satisfied by any solution, while the number of soft clauses satisfied by a solution must be maximized. Partial MaxSAT can be seen as a compromise between SAT and MaxSAT where the hard clauses are treated as a SAT problem and the soft clauses are treated as a MaxSAT problem.

3 Restoring Satisfiability with MaxSAT

The goal of restoring satisfiability of a CSP instance with a minimal impact can be achieved by identifying the smallest size set of tuples to be removed from the CSP constraint relations. Removing these tuples guarantees that the instance is no longer unsatisfiable, whereas adding any of the removed tuples makes the instance unsatisfiable.

With the aim of restoring CSPs satisfiability, recent work by Grégoire *et al.* [7] introduced a two-step algorithm: (1) identify a MUC and (2) identify a MUST in that MUC. This algorithm is called MUSTER and is outlined in Algorithm 1. The extraction of a MUC and the extraction of a MUST is performed by using tools developed in the past which are publicly available.

This paper suggests restoring CSP satisfiability with MaxSAT, which guarantees restoring satisfiability. This is a clear advantage with respect to MUSTER, which only removes one MUST. The proposed algorithm involves three steps: (1) encoding the problem into partial MaxSAT, (2) solving the partial MaxSAT instance and (3) identifying the tuples to be removed from the CSP instance given the partial MaxSAT solution. The pseudo code for this procedure is illustrated in Algorithm 2. For each partial MaxSAT instance P' encoding a CSP instance P, there is a mapping between each MaxSAT variable and each domain value of

Algorithm 1 Identifying a MUST in a CSP

MUSTER($CSP\ P$)
1 $P' \leftarrow$ Extract MUC from P
2 $P'_{SAT} \leftarrow$ Encode CSP P' into SAT
3 $\triangleright$ $M_{P'-P'_{SAT}}$ contains mapping SAT/CSP
4 $Sol_{P'_{SAT}} \leftarrow$ Extract MUS from P'_{SAT}
5 $P'' \leftarrow$ CSP with recovered constraint tuples from $Sol_{P'_{SAT}}$ using $M_{P'-P'_{SAT}}$
6 **return** P''

Algorithm 2 Restoring CSP satisfiability with MaxSAT

RESTORECSPWITHMAXSAT($CSP\ P$)
1 $P_{PMaxSAT} \leftarrow$ Encode P as a partial MaxSAT instance
2 $\triangleright$ $M_{P-P_{PMaxSAT}}$ contains mapping SAT/CSP
3 $Sol_{P_{PMaxSAT}} \leftarrow PMaxSATsolver(P_{PMaxSAT})$
4 $P' \leftarrow$ CSP with recovered constraint tuples from $Sol_{P_{PMaxSAT}}$ using $M_{P-P_{PMaxSAT}}$
5 **return** P'

a CSP variable. A partial MaxSAT solver is then called to solve P'. Its solution is used to identify the set of constraint tuples of P to be removed. In this section, we will explain how to encode a CSP instance into a partial MaxSAT instance to restore satisfiability.

3.1 A MaxSAT Encoding

We recall that given a CSP instance $P = (X, D, C)$ for which the constraints represent conflicts, it is encoded into a SAT instance P' as follows [3]:

- For each variable $x \in X$ and each value in the corresponding domain $d(x) \in D$ is created a Boolean variable.
- For each set of Boolean variables $\{b_1, \ldots, b_{|d(x)|}\}$ associated with the same CSP variable $x \in X$ is created one clause $(b_1 \vee \ldots \vee b_{|d(x)|})$ to ensure that each CSP variable is assigned at least one value and a set of binary clauses $(\neg b_i \vee \neg b_j)$ with $1 \leq i < j \leq |d(x)|$ to ensure that no more than one value is assigned to a CSP variable. In what follows, these clauses will be called *at least one clauses* and *at most one clauses*, respectively.
- For each tuple $t \in R$ of each constraint $c = (S, R) \in C$ is created a clause with $|S|$ negative literals to ensure that the corresponding values are disallowed. In what follows, these clauses will be called *conflict clauses*.

[3] Details about encoding CSP into SAT and vice versa can be found in [17].

We should note that there exist alternative encodings for guaranteeing that exactly one value is assigned to each CSP variable (see for example [13, 16]).

Example 3. (CSP to SAT) Consider a CSP instance $P = (X, D, C) = (\{x_1, x_2\}, \{d(x_1) = \{1, 2\}, d(x_2) = \{1, 3\}\}, \{((x_1, x_2), \{(1, 1), (2, 3)\})\})$. Let us consider that a Boolean variable b_{ij} corresponds to the domain value $j \in d(x_i) \in D$ of variable $x_i \in X$. The corresponding SAT instance has therefore the following set of clauses: $\{(x_{11} \vee x_{12}), (x_{21} \vee x_{23}), (\neg x_{11} \vee \neg x_{12}), (\neg x_{21} \vee \neg x_{23}), (\neg x_{11} \vee \neg x_{21}), (\neg x_{12} \vee \neg x_{23})\}$. The first two clauses correspond to *at least one clauses*, the next two clauses correspond to *at most one clauses* and the last two clauses correspond to *conflict clauses*.

The CSP to SAT encoding used by MUSTER does not include the *at most one clauses*. Provided that every *at least one clauses* belong to the MUST, computing one MUST amounts to compute one MUS in the CNF formula.

Consider an unsatisfiable CSP instance $P = (X, D, C)$ with constraints representing conflicts and for which the minimum set of tuples to be removed in order to restore satisfiability is to be found. This problem is encoded into a partial MaxSAT instance P' as follows:

- Each *at least one clause* produced by an encoding of P into SAT is a *hard clause* of P'.
- Each *conflict clause* produced by an encoding of P into SAT is a *soft clause* of P'.

Remark 1. When using the MaxSAT encoding to restore satisfiability of a CSP instance, there is no need of adding *at most one clauses* to guarantee a one-to-one correspondence between Boolean variables and CSP variable values.

Proof. We assume that the CSP instance is unsatisfiable. The hard clauses guarantee that at least one value is assigned to each CSP variable. The soft clauses guarantee the smallest number of violated constraints. Having more than one value assigned to a CSP variable, which means that any of those values can be selected to be in the solution, cannot reduce the number of violated constraints.

Proposition 1. *A solution to the MaxSAT instance P' corresponds to the minimum set of tuples to be removed from P in order to restore satisfiability.*

Proof. The hard clauses guarantee that any solution to the partial MaxSAT instance corresponds to a CSP complete assignment. Each soft clause represents one tuple in a constraint. Hence, a solution to the partial MaxSAT instance satisfies the largest number of tuples, which is equivalent to unsatisfying the smallest number of tuples.

As an additional remark, observe that MaxSAT has been used in the past to solve the MaxCSP problem [1], thus allowing to know which constraints have to be removed to restore satisfiability of a CSP. Although the motivation of this use of MaxSAT clearly differs from the one proposed here, the encoding is

similar. Observe that a given assignment to the variables in a constraint scope can correspond to at most one tuple of the constraint relation, in which case we say that that that conflict tuple is *violated*. Hence, the number of violated constraints corresponds to the number of violated conflict tuples.

3.2 Unsatisfiability-Based MaxSAT Solvers

Unsatisfiability-based MaxSAT solvers [3, 15, 14] represent an approach for solving MaxSAT problem instances, which contrasts with the traditional MaxSAT solvers based on branch and bound algorithms [9]. Unsatisfiability-based MaxSAT solvers iteratively identify Unsatisfiable Subformulas (USs), not necessarily minimal, which are relaxed after being identified. Clauses are relaxed by adding one relaxation variable per clause. The use of cardinality constraints on the variables used to relax clauses in USs guarantees that a minimum number of clauses is relaxed. Hence, the MaxSAT solution is computed. The identification of clauses in USs is iterated until the resulting formula is satisfiable. In this case, one clause from each identified US is relaxed. A number of variants of MaxSAT algorithms have been proposed [3, 15, 14], which differ on the actual organization of the algorithm, and the way cardinality constraints are encoded to clausal form.

The use of unsatisfiability-based MaxSAT solvers has the advantage of having a solver which identifies unsatisfiable cores while running. Hence, in case a solution is not found within the allowed CPU time, there is still relevant information for restoring satisfiability.

In this paper we will be using the unsatisfiability-based MSUnCore [15, 14] version of April 2009. For this reason, we will use the name MSUnCore to denote the MaxSAT-based algorithm for restoring CSPs satisfiability.

3.3 Muster *vs* MSUnCore

A few interesting remarks are made bellow in order to stress that MSUnCore should be considered as an alternative approach to Muster.

We should first clarify how the Muster algorithm could be instrumented to restore CSPs satisfiability. First, a MUC is identified in a CSP. Second, MUSTs in the MUC are identified to restore satisfiability, i.e. the required tuples are removed from the MUC such that it becomes satisfiable. Now these same tuples are removed from the CSP and the whole process is repeated.

Remark 2. The solution provided by MSUnCore removes a number of tuples that is equal or smaller than the iterated application of Muster.

Proof. Muster first identifies a MUC and then MUSTs within the MUC. This does not guarantee removing tuples shared by MUSTs belonging to different MUCs. (Figure 2 illustrates this issue: unless Muster is lucky enough to remove tuple $(0, 2)$ or tuple $(0, 3)$, Muster will remove 2 tuples in two iterations, while MSUnCore will remove only one tuple.)

Remark 3. The number of Unsatisfiable Sets of Tuples (USTs) identified by MSUnCore during the search provides a lower bound for the number of tuples to be removed to restore satisfiability.

Proof. Unsatisfiability-based MaxSAT solvers (of which MSUnCore is a concrete example), iteratively identify and relax unsatisfiable subformulas (USs). Moreover, for each iteration for which a US is found, the number of clauses to relax is increased by 1. Consequently, at each step of the MSUnCore algorithm, the number of identified USs represents a lower bound on the MaxSAT solution, and so represents a lower bound on the number of tuples to be removed to restore satisfiability.

4 Experimental Evaluation

The experimental evaluation was performed over a set of instances from the First CSP Competition (`http://cpai.ucc.ie/05/CallForSolvers.html`). This set of instances corresponds to the set of instances used to test Muster [7].

Table 1 provides the characterization and results for each instance. Each instance is characterized in terms of the number of CSP variables (#vars) and constraints (#constr). In addition, the CPU time required by Muster to identify one MUC and one MUST in that MUC is given (1MUC+1MUST), as well as the CPU time required by an unsatisfiability-based MaxSAT solver (MSUnCore [14]) to identify the first UST (1UST), which is not necessarily minimal. Additional information gives the CPU time required by MSUnCore to restore satisfiability (AllMUSTs), which implies eliminating all MUSTs, as well as the total number of tuples to be removed to restore satisfiability (RemovedTuples). The CPU time was limited to 1000 seconds (TO means timeout). In case only a few USTs are identified, it is given a lower bound on the number of tuples to be removed to restore satisfiability.

The results included for Muster where taken from [7]. We should therefore note that the machine used for running MSUnCore is not significantly different from the one used for running Muster. Muster was run on a Pentium IV 3GHz under Linux Fedora Core 5, whereas MSUnCore was run on a Intel Xeon 5160 3GHz under RedHat Enterprise Linux WS4. The performance difference is therefore not significant for the conclusions to be drawn below.

First of all, the results illustrate the feasibility of the proposed approach. Many times MSUnCore is able to restore satisfiability faster than Muster is able to identify one MUST in a MUC. Also interesting is to note that most instances are restored in terms of satisfiability by removing very few tuples. This supports the claim that removing tuples has a very little impact. For the cases where MSUnCore is not able to completely restore satisfiability, it is able to identify a few USTs though, which gives a lower bound on the number of tuples to be removed. In any case, the identification of the first UST by MSUnCore is *always* faster than using Muster. Also, MSUnCore seems to require more time when the number of tuples to be removed is larger.

Table 1. Experimental results

Instance	CSP		Muster	MSUnCore		
Name	#vars	#constr	1MUC+1MUST	1UST	AllMUSTs	RemovedTuples
composed-25-1-2-0	224	4,440	23.80	0.01	0.04	1
composed-25-1-2-1	224	4,440	26.56	0.01	0.67	3
composed-25-1-25-8	247	4,555	15.10	0.01	0.18	2
composed-75-1-2-1	624	10,440	77.27	0.01	0.17	2
composed-75-1-2-2	624	10,440	81.18	0.01	0.19	2
composed-75-1-25-8	647	10,555	82.78	0.01	0.25	2
composed-75-1-80-6	702	10,830	69.65	0.01	0.16	2
composed-75-1-80-7	702	10,830	392.08	0.01	0.05	1
composed-75-1-80-9	702	10,830	95.17	0.01	0.18	2
qk_10_10_5_add	55	48,640	TO	74.62	92.89	1
qk_10_10_5_mul	105	49,140	2814.28	99.55	111.16	1
qk_8_8_5_add	38	19,624	548.03	6.04	13.44	1
qk_8_8_5_mul	78	19,944	531.90	10.58	19.36	1
graph2_f25	2,245	145,205	853.41	0.36	1.64	1
qa_3	40	800	8.64	0.01	0.06	1
dual_ehi-85-297-14	4,111	102,234	43.61	0.14	TO	$\geq$8
dual_ehi-85-297-15	4,133	102,433	29.88	0.15	TO	$\geq$8
dual_ehi-85-297-16	4,105	102,156	33.73	0.15	TO	$\geq$8
dual_ehi-85-297-17	4,102	102,112	47.04	0.15	TO	$\geq$8
dual_ehi-85-297-18	4,120	102,324	33.88	0.15	TO	$\geq$8
dual_ehi-90-315-21	4,388	108,890	38.16	0.28	TO	$\geq$8
dual_ehi-90-315-22	4,368	108,633	48.26	0.26	TO	$\geq$8
dual_ehi-90-315-23	4,375	108,766	15.09	0.25	TO	$\geq$8
dual_ehi-90-315-24	4,378	108,793	29.99	0.21	TO	$\geq$7
dual_ehi-90-315-25	4,398	108,974	34.30	0.20	TO	$\geq$8
scen6_w2	648	513,100	TO	2.51	TO	$\geq$7
scen6_w1_f2	319	274,860	TO	17.25	31.83	1
scen11_f10	4,103	738,719	602.53	1.99	TO	$\geq$4
scen11_f12	4,103	707,375	541.95	1.87	TO	$\geq$3

5 Conclusions and Future Work

This paper suggests the use of MaxSAT to restore CSP satisfiability by removing the smallest number of constraint tuples. This solution contrasts with a MaxCSP solution as tuples instead of constraints are removed. We argue that removing tuples is more adequate for most of the problems due to having a minor impact.

Future work includes adapting MSUnCore for identifying *minimal* unsatisfiable cores. This will have the advantage of identifying MUSTs instead of USTs, which are more relevant in case the search does not terminate in the given timeout.

References

1. J. Argelich, A. Cabiscol, I. Lynce, and F. Manyà. Modelling Max-CSP as partial Max-SAT. In *International Conference on Theory and Applications of Satisfiability Testing*, pages 1–14, 2008.
2. R. Bruni. On exact selection of minimally unsatisfiable subformulae. *Annals of Mathematics and Artificial Intelligence*, 43(1):35–50, 2005.
3. Z. Fu and S. Malik. On solving the partial MAX-SAT problem. In *International Conference on Theory and Applications of Satisfiability Testing*, pages 252–265, August 2006.
4. E. I. Goldberg and Y. Novikov. Verification of proofs of unsatisfiability for CNF formulas. In *Design, Automation and Test in Europe Conference*, pages 10886–10891, 2003.
5. É. Grégoire, B. Mazure, and C. Piette. Extracting MUSes. In *European Conference on Artificial Intelligence*, pages 387–391, 2006.
6. É. Grégoire, B. Mazure, and C. Piette. Local-search extraction of MUSes. *Constraints*, 12(3):325–344, 2007.
7. É. Grégoire, B. Mazure, and C. Piette. MUST: Provide a finer-grained explanation of unsatisfiability. In *International Conference on Principles and Practice of Constraint Programming*, pages 317–331, 2007.
8. U. Junker. QUICKXPLAIN: Preferred explanations and relaxations for over-constrained problems. In *AAAI National Conference on Artificial Intelligence*, pages 167–172, 2004.
9. C. M. Li and F. Manyà. MaxSAT, hard and soft constraints. In A. Biere, M. Heule, H. van Maaren, and T. Walsh, editors, *SAT Handbook*, pages 613–631. IOS Press, 2009.
10. M. Liffiton, M. Mneimneh, I. Lynce, Z. Andraus, J. Marques-Silva, and K. Sakallah. A branch and bound algorithm for extracting smallest minimal unsatisfiable subformulas. *Constraints, In Press*, 2009.
11. M. Liffiton and K. Sakallah. On finding all minimally unsatisfiable subformulas. In *International Conference on Theory and Applications of Satisfiability Testing*, pages 173–186, 2005.
12. M. Liffiton and K. Sakallah. Algorithms for computing minimal unsatisfiable subsets of constraints. *Journal of Automated Reasoning*, 40(1):1–33, 2008.
13. J. Marques-Silva and I. Lynce. Towards robust CNF encodings of cardinality constraints. In *International Conference on Principles and Practice of Constraint Programming*, pages 483–497, 2007.
14. J. Marques-Silva and V. Manquinho. Towards more effective unsatisfiability-based maximum satisfiability alg orithms. In *International Conference on Theory and Applications of Satisfiability Testing*, pages 225–230, 2008.
15. J. Marques-Silva and J. Planes. Algorithms for maximum satisfiability using unsatisfiable cores. In *Design, Automation and Test in Europe Conference*, pages 408–413, 2008.
16. C. Sinz. Towards an optimal CNF encoding of boolean cardinality constraints. In *International Conference on Principles and Practice of Constraint Programming*, pages 827–831, 2005.
17. T. Walsh. SAT v CSP. In *International Conference on Principles and Practice of Constraint Programming*, pages 441–456, 2000.
18. L. Zhang and S. Malik. Validating SAT solvers using an independent resolution-based checker: Practical implementations and other applications. In *Design, Automation and Test in Europe Conference*, pages 10880–10885, 2003.