# Solving WCSP by Extraction of Minimal Unsatisfiable Cores

Nicolas Paris (student)
Christophe Lecoutre, Olivier Roussel, and Sébastien Tabary (supervisors)

CRIL - CNRS, UMR 8188,
Université Lille Nord de France, Artois
rue de l'université, 62307 Lens cedex, France
{paris,lecoutre,roussel,tabary}@cril.fr

**Abstract.** Usual techniques to solve WCSP are based on cost transfer operations coupled with a branch and bound algorithm. In this paper, we propose an incomplete approach based on the extraction and relaxation of Minimal Unsatisfiable Cores.

## 1 Introduction

The goal of the Constraint Satisfaction Problem (CSP) is to find a solution to a Constraint Network (CN). Sometimes, preferences among solutions need to be expressed. This is possible through the introduction of soft constraints. The framework called WCSP (Weighted CSP) allows us to handle such constraints: each soft constraint is defined by a cost function that associates a violation degree, called cost, with every possible instantiation of a subset of variables. The WCSP goal is to find a complete instantiation with a minimal combined cost of the soft constraints. Most of the current methods to solve Weighted CNs (WCNs) are based on branch and bound tree search combined with the use of soft local consistencies for estimating minimal costs of subproblems during search.

Because they must combine costs, algorithms establishing soft local consistencies are often more complex than their CSP counterparts. In this paper, we propose an original approach for WCSP that consists in solving a WCN by iteratively generating and solving classical CNs in order to benefit from efficient CSP solvers developed for more than two decades. The principle is to enumerate a sequence of CNs according to an increasing cost order related to the WCN. When a CN is proved to be unsatisfiable, a Minimal Unsatisfiable Core (MUC) is extracted in order to identify the soft constraints whose costs must be increased (relaxation), so as to finally obtain a solution. Note that our approach is inspired by MaxSAT techniques based on MUS extraction [1, 8].

## 2 Technical Background

### 2.1 Generalities

A *constraint network* (CN) $P$ involves a finite set of variables and a finite set of constraints. Each variable $x$ has an associated domain which contains the finite set of values

that can be assigned to $x$. Each constraint $c$ involves an ordered set of variables, denoted $scp(c)$ and called *scope* of $c$, and is defined by a relation containing the set of tuples allowed for the variables of $scp(c)$. A solution is a complete instantiation (i.e., the assignment of a value to each variable) such that all the constraints are satisfied. A CN is satisfiable iff it admits at least one solution. For more information on CNs, see [10].

An unsatisfiable core of a CN $P$ is an unsatisfiable subset of constraints of $P$. A core $C$ is a MUC iff each strict subset of constraints of $C$ is satisfiable. Several methods to extract MUCs of CNs are presented in [4–6]. Among the different versions presented in [5], we have chosen for our implementation the *dichotomic* version, called *dcMUC*.

A *Weighted Constraint Network* (WCN) $W$ involves a finite set of variables, a finite set of soft constraints denoted by $cons(W)$, and has an associated value $k > 0$ which is either a natural integer or $+\infty$. Each soft constraint $w \in cons(W)$ has a scope $scp(w)$ and is defined as a cost function from $l(scp(w))$ to $\{0, \ldots, k\}$, where $l(scp(w))$ is the Cartesian product of the domains of the variables involved in $w$ (labeling). An instantiation which is given the cost $k$ is *forbidden*. Otherwise, it is permitted with the corresponding cost. Costs are combined with the bounded addition $\oplus$ defined as: $\forall a, b \in \{0, \ldots, k\}, a \oplus b = min(k, a + b)$. The objective of WCSP is, for a given WCN, to find a complete instantiation with a minimal cost. Different variations of soft arc consistency for WCSP have been proposed during the last decade like AC* [7] and existential arc consistency (EDAC) [3]. All the algorithms proposed to enforce these different levels of consistency use cost transfer operations based on the concept of equivalence-preserving transformations. For more information on WCNs, see [9].

### 2.2 Layers and Fronts

In this paper, we focus on extensional soft constraints, but the method can be easily extended to other kinds of constraints. These are soft table constraints where some tuples are explicitly listed with their costs in a table, and a default cost gives the cost of all implicit tuples (i.e., those not present in the table). All tuples of a soft table constraint having the same cost can be grouped to form a subset called *layer* (see Figure 1). A particular layer corresponds to the default cost: this layer contains no tuple, but implicitly represents all tuples that do not explicitly appear in the table. Within a constraint, layers are increasingly ordered according to their costs.

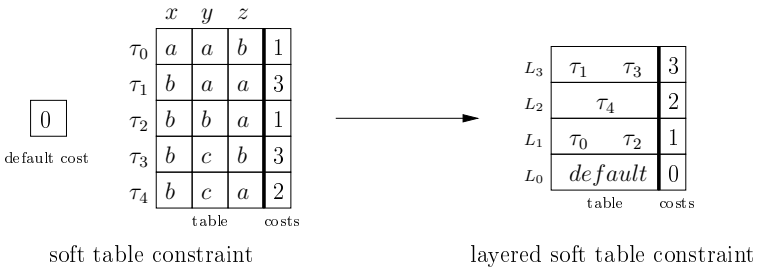


**Fig. 1.** Tuples of soft table constraints are grouped into layers.

A *front* $f$ is a function that maps each constraint of a WCN to one of its layers. A front represents a kind of border between the layers that will be considered (allowed) at a given moment, and the layers that will be discarded (forbidden). We shall use an array notation for the fronts. So, $f[w]$ will represent the layer associated with the constraint $w$ in the front $f$. The cost of a front $f$, $cost(f)$, is obtained by summing up all costs of the constraint layers corresponding to the front $f$. A front $f'$ is the direct successor of a front $f$ iff there exists a constraint $w_i$ such that $f'[w_i] = f[w_i] + 1$ and $\forall j \neq i, f'[w_j] = f[w_j]$. Because the layers of any constraint are increasingly ordered following their costs, we deduce that $cost(f) < cost(f')$.

## 3 General Principle

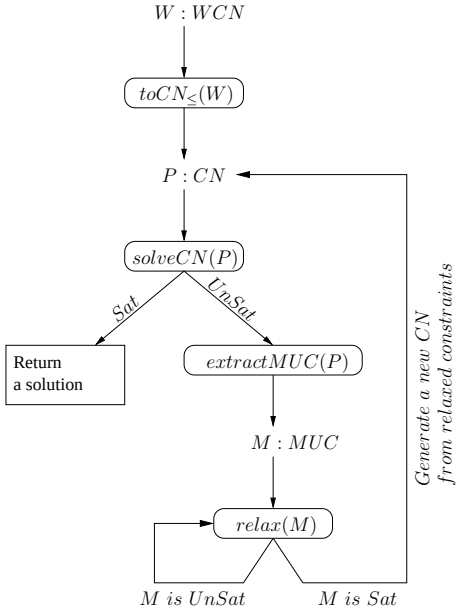


**Fig. 2.** Principle of the iterative relaxation of MUCs.

Figure 2 illustrates the main idea of our approach for solving weighted constraint networks. First, from a WCN $W$, an initial constraint network $P$ is built as follows: for each constraint of $W$, the tuples occurring in the layer of minimal cost are considered as allowed in $P$, while other tuples are considered as forbidden (function $toCN_{\leq}$). This process is similar to that proposed in [2]. Next, the CN $P$ is solved using a constraint solver (through a call to the function $solveCN$). If $P$ is satisfiable, then a solution is returned. Otherwise a MUC is extracted from $P$ (through a call to the function $extractMUC$) and then exploited. Indeed, some constraints of the MUC will be successively relaxed until satisfiability of the MUC is restored (through a call to the function

$relax$). Some constraints belonging to the MUC will be relaxed (some layers will be switched from the "forbidden" status to the "allowed" one). When constraints corresponding to the MUC are relaxed so as to become satisfiable, the process loops until the global satisfiability of $P$ is reached. In this case, a solution is returned.

## 4 Algorithm

The function $cons(W)$ returns the set of soft constraints of the WCN $W$. Fronts are exploited by the function $toCN_{\leq}(W, f)$ which builds a CN by selecting as allowed tuples for a soft constraint $w$ the tuples of the layers below $f[w]$, i.e., with an index less than or equal to $f[w]$ ; these layers are said allowed. In other words, for each hard constraint $c$ built from a soft constraint $w$ of $W$, the tuples allowed in $c$ are those having a cost less than or equal to the cost of the layer $f[w]$. The function $solveCN(P)$ solves a CN given in parameter and returns either a solution or $\bot$ if the CN is unsatisfiable. The function $extractMUC(P)$ returns a MUC from an unsatisfiable CN $P$ given in parameter. The function $restrict(W, M)$ returns a WCN containing only the constraints of the WCN $W$ corresponding to the constraints present in the MUC $M$.

From a WCN $W$, Function incompleteSearch returns a solution for a CN derived from $W$. It starts with a first front which retains only the least cost layer of each constraint (lines 1-2). From a WCN $W$ and a front $f$, Function $toCN_{\leq}$ builds a constraint network from both the WCN and the front $f$ previously selected (line 4). This network is then solved. If a solution is found, then it is returned (line 7). If the network has been proved unsatisfiable, it is necessary to relax some constraints of the WCN. To achieve this, the algorithm extracts a WCN $W'$ from a computed MUC (lines 9-10). The front $f$ is then updated by Function $relax$ (line 11). This procedure will relax one (or several constraints) of $W'$ in order to break the MUC associated with $W'$. This process loops until a solution is found for the constraint network associated with $W$.

Function $relax(W, f)$ updates the front $f$ in order to allow new layers. The idea is to relax constraints of the MUC in a greedy way such as to make the MUC satisfiable. A local priority queue is used and initialized with the front $f$. While this queue is not empty and the MUC not broken, we extract the front $f$ having the lowest cost. Successors of $f$ are generated by Function $relax$ which generates new fronts by incrementing $f[w]$ for one of the constraints $w$. Note that in the different calls to the function $relax$, it is sufficient to work only on a subset of the constraints belonging to the WCN $W'$ associated with the MUC ($restrict(W, M)$). Moreover, because only constraints belonging to the WCN associated with the MUC are considered at a given time, the CNs generated are typically much smaller than the original WCN. One can expect that checking the satisfiability will be quite simple (hence fast) in most cases.

This approach doesn't guarantee the identification of an optimum solution. The reason is that all the relaxations of MUCs are not considered. Indeed, we only identify the first relaxation which restores satisfiablity of a MUC.

**Function** incompleteSearch($W$: WCN)

```
1  foreach w ∈ cons(W) do
2      f[w] ← 0
3  repeat
4      P ← toCN≤(W, f)
5      sol ← solveCN(P)
6      if sol ≠ ⊥ then
7          return sol
8      else
9          M ← extractMUC(P)
10         W' ← restrict(W, M)
11         f ← relax(W', f)
12 until sol ≠ ⊥
```

## 5 Experimental Results

In order to show the viability of our approach, we have performed some experiments using a computer with processors Intel(R) Core(TM) i7-2820QM 2.30GHz. Our greedy approach, noted $GMR$, has been compared with two complete approaches enforcing EDAC, proposed by both our solver $AbsCon$ and the solver $ToulBar2$. A time-out of 600 seconds was set per instance. The total CPU time necessary to solve each instance is given as well as the upper bound found (UB). If the execution of the algorithm is not yet finished before the time-out, the bound found at the end of the 600 seconds is given. Table 1 provides the results obtained for some instances $spot5$ and $celar$.

| | | AbsCon | | ToulBar2 |
|---|---|---|---|---|
| $Instances$ | | GMR | EDAC | EDAC |
| $spot5/spot5$-404 | CPU | 4.99 | $> 600$ | 217 |
| | UB | 118 | **114** | **114** |
| $spot5/spot5$-412 | CPU | 18.8 | $> 600$ | $> 600$ |
| | UB | **33, 403** | 43, 390 | 37, 399 |
| $spot5/spot5$-505 | CPU | 12 | $> 600$ | $> 600$ |
| | UB | **22, 266** | 28, 258 | 25, 268 |
| $celar/graph$-05 | CPU | 16.6 | $> 600$ | 0.62 |
| | UB | **221** | 4, 645 | **221** |
| $celar/scen$-06-20 | CPU | 68.5 | $> 600$ | 67.9 |
| | UB | 3, 402 | 8, 594 | **3, 163** |
| $celar/scen$-07 | CPU | 209.9 | $> 600$ | $> 600$ |
| | UB | **426, 423** | $31,230K$ | 505, 731 |

**Table 1.** CPU time in seconds and bound found before the time-out for instances *spot5* and *celar*.

We observe that $GMR$ provides better or equal bounds than those obtained by the two complete approaches, except for the instance $spot5$-404 and $scen$-06-20. Beyond the fact that $GMR$ is not complete, this can be explained by the fact that these instances are relatively easy and cost transfer algorithms reach a better UB before the time-out. However, it is interesting to note that the bound found by $GMR$ is not so different from the one obtained by complete approaches, and this within an acceptable time. On the other instances, we observe that our approach provides a better or equal UB than those of complete approaches in a time shorter than the time-out. Indeed, on the $graph$-05 instance, even if it is not so fast than $ToulBar2$, $GMR$ is able to find the optimum UB.

## 6 Conclusion

In this paper, we have proposed an original greedy approach for solving weighted constraint networks through successive resolutions of hard constraint networks. We identify a minimal unsatisfiable core for each unsatisfiable constraint network, in order to focus the cost increase on the sole constraints of the MUC. We have shown that our algorithm obtains results which are comparable with other state of the art approaches. Currently, we are working on a complete version of our greedy approach.

## Acknowledgments

This work has been supported by both CNRS and OSEO within the ISI project 'Pajero'.

## References

1. C. Ansotegui, M.L. Bonet, and J. Levy. A new algorithm for weighted partial MaxSAT. In *Proceedings of AAAI'10*, pages 3–8, 2010.
2. M. Cooper, S. de Givry, M. Sanchez, T. Schiex, and M. Zytnicki. Virtual arc consistency for weighted CSP. In *Proceedings of AAAI'08*, pages 253–258, 2008.
3. S. de Givry, F. Heras, M. Zytnicki, and J. Larrosa. Existential arc consistency: Getting closer to full arc consistency in weighted CSPs. In *Proceedings of IJCAI'05*, pages 84–89, 2005.
4. J.L. de Siqueira and J.F. Puget. Explanation-based generalisation of failures. In *Proceedings of ECAI'88*, pages 339–344, 1988.
5. F. Hemery, C. Lecoutre, L. Sais, and F. Boussemart. Extracting MUCs from constraint networks. In *Proceedings of ECAI'06*, pages 113–117, 2006.
6. U. Junker. QuickXplain: preferred explanations and relaxations for over-constrained problems. In *Proceedings of AAAI'04*, pages 167–172, 2004.
7. J. Larrosa and T. Schiex. Solving weighted CSP by maintaining arc consistency. *Artificial Intelligence*, 159(1-2):1–26, 2004.
8. J.P. Marques-Silva and J. Planes. Algorithms for maximum satisfiability using unsatisfiable cores. In *DATE'08*, pages 408–413, 2008.
9. P. Meseguer, F. Rossi, and T. Schiex. Soft constraints. In *Handbook of Constraint Programming*, chapter 9, pages 281–328. Elsevier, 2006.
10. F. Rossi, P. van Beek, and T. Walsh, editors. *Handbook of Constraint Programming*. Elsevier, 2006.