Propagating Soft Table Constraints

Christophe Lecoutre, Nicolas Paris, Olivier Roussel, Sébastien Tabary

▶ **To cite this version:**

Christophe Lecoutre, Nicolas Paris, Olivier Roussel, Sébastien Tabary. Propagating Soft Table Constraints. 18th International Conference on Principles and Practice of Constraint Programming (CP'12), 2012, Québec, Canada. pp.390-405. hal-00865618

HAL Id: hal-00865618
https://hal.science/hal-00865618v1

Submitted on 24 Sep 2013

HAL is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L'archive ouverte pluridisciplinaire **HAL**, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d'enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.

Propagating Soft Table Constraints

Christophe Lecoutre, Nicolas Paris, Olivier Roussel, and Sébastien Tabary

CRIL - CNRS, UMR 8188,
Univ Lille Nord de France, Artois
F-62307 Lens, France
{lecoutre,paris,roussel,tabary}@cril.fr

Abstract. WCSP is a framework that has attracted a lot of attention during the last decade. In particular, many filtering approaches have been developed on the concept of equivalence-preserving transformations (cost transfer operations), using the definition of soft local consistencies such as, for example, node consistency, arc consistency, full directional arc consistency, and existential directional arc consistency. Almost all algorithms related to these properties have been introduced for binary weighted constraint networks, and most of the conducted experiments typically include networks with binary and ternary constraints only. In this paper, we focus on extensional soft constraints (of large arity), so-called soft table constraints. We propose an algorithm to enforce a soft version of generalized arc consistency (GAC) on such constraints, by combining both the techniques of cost transfer and simple tabular reduction, the latter dynamically maintaining the list of allowed tuples in constraint tables. On various series of problem instances containing soft table constraints of large arity, we show the practical interest of our approach.

1 Introduction

The Weighted Constraint Satisfaction Problem (WCSP) is an optimization framework used to handle soft constraints, which has been successfully applied in many applications of Artificial Intelligence and Operations Research. 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. Using the (bounded) addition $\oplus$, these costs can be combined in order to obtain the overall cost of any complete instantiation. Finding a complete instantiation with a minimal cost is known to be NP-hard.

Several properties, such as Node Consistency (NC) and Arc Consistency (AC), introduced in the early 70's for the Constraint Satisfaction Problem (CSP), have been studied later in the context of WCSP. Since then, more and more sophisticated developments about the best form of soft arc consistency have been proposed over the years: full directional arc consistency (FDAC) [3], existential directional arc consistency (EDAC) [6], virtual arc consistency (VAC) and optimal soft arc consistency (OSAC) [4], among others. Cost transfer, which is the general principle behind the algorithms enforcing such properties, preserves the semantics of the soft constraints network while concentrating costs on domain values (unary constraints) and a global constant cost (nullary constraint). Quite interestingly, cost transfer algorithms have been shown to be particularly efficient to solve real-world problem instances, especially when soft constraints are binary or ternary (see `http://costfunction.org` for many such problems).

For soft constraints of large arity, cost transfer becomes a serious issue because the risk of combinatorial explosion has to be controlled. A first solution is to postpone cost transfer operations until the number of unassigned variables (in constraint scopes) is sufficiently low. However, it may dramatically damage the filtering capability of the algorithms, in particular at the beginning of search. A second solution is to adapt soft local consistency algorithms to certain families of global soft constraints. This is the approach followed in [14, 15] where the concept of projection-safe soft constraint is introduced. A third solution is to decompose soft constraints (cost functions) into soft constraints of smaller arity [7]. Decomposition of global soft constraints can also be envisioned [1]. Unluckily, not all soft constraints can be decomposed.

To enforce the property, known as Generalized Arc Consistency (GAC) on soft table constraints, i.e., soft constraints defined extensionally by listing tuples and their costs, we propose to combine two techniques, namely, Simple Tabular Reduction (STR) [16] and cost transfer. Basically, whenever some domain values are deleted during propagation or search, all tuples that become invalid are removed from constraint tables. This allows us to identify values that are no longer consistent with respect to GAC. Interestingly, because all valid tuples of tables are iterated over, it is easy and cheap to compute minimum costs of values. This is particularly useful for performing efficiently projection operations that are required to establish GAC.

The paper is organized as follows. In the first section, we present the technical background about WCSP. Next, we present the data structures and the algorithms used in our approach GAC^w-WSTR. We prove the correctness and discuss the complexity of our method. Finally, we introduce some new series of benchmarks and show the practical interest of our approach.

2 Technical Background

A *weighted constraint network* (WCN) P is a triplet $(\mathscr{X}, \mathscr{C}, k)$ where $\mathscr{X}$ is a finite set of n variables, $\mathscr{C}$ is a finite set of e soft (or weighted) constraints, and $k > 0$ is either a natural integer or ∞. Each variable x has a (current) domain, denoted by $dom(x)$, which is the finite set of values that can be (currently) assigned to x; the initial domain of x is denoted by $dom^{init}(x)$. d will denote the greatest domain size. An *instantiation* I of a set $X = \{x_1, \ldots, x_p\}$ of p variables is a set $\{(x_1, a_1), \ldots, (x_p, a_p)\}$ such that $\forall i \in 1..p, a_i \in dom^{init}(x_i)$; each a_i is denoted by $I[x_i]$. I is *valid* on P iff $\forall (x, a) \in I, a \in dom(x)$. Each soft constraint $c_S \in \mathscr{C}$ involves an ordered set S of variables, called its *scope*, and is defined as a cost function from $l(S)$ to $\{0, \ldots, k\}$ where $l(S)$ is the set of possible instantiations of S. When an instantiation $I \in l(S)$ is given the cost k, i.e., $c_S(I) = k$, it is said *forbidden*. Otherwise, it is permitted with the corresponding cost (0 being completely satisfactory). Costs are combined with the specific operator $\oplus$ defined as: $\forall \alpha, \beta \in \{0, \ldots, k\}, \alpha \oplus \beta = min(k, \alpha + \beta)$. The partial inverse of $\oplus$ is $\ominus$ defined by: if $0 \leq \beta \leq \alpha < k$, $\alpha \ominus \beta = \alpha - \beta$ and if $0 \leq \beta < k$, $k \ominus \beta = k$. A *unary* (resp., *binary*) constraint involves 1 (resp., 2) variable(s), and a *non-binary* one strictly more than 2 variables. For any constraint c_S, every pair (x, a) such that $x \in S \wedge a \in dom(x)$ is called a *value of* c_S.

For any instantiation I and any set of variables X, let $I_{\downarrow X} = \{(x,a) \mid (x,a) \in I \wedge x \in X\}$ be the projection of I on X. If c_S is a soft constraint and I is an instantiation of a set $X \supseteq S$, then $c_S(I)$ will be considered to be equal to $c_S(I_{\downarrow S})$ (in other words, projections will be implicit). For a WCN P and a complete instantiation I of P, the cost of I is $\bigoplus_{c_S \in \mathscr{C}} c_S(I)$. The usual (NP-hard) task of Weighted Constraint Satisfaction Problem (WCSP) [12] is, for a given WCN, to find a complete instantiation with a minimal cost.

Many forms of soft arc consistency have been proposed during the last decade (e.g., see [4]). We now briefly introduce some of them. Without any loss of generality, the existence of a nullary constraint $c_\emptyset$ (a constant) as well as the presence of a unary constraint c_x for every variable x is assumed. A variable x is node-consistent (NC) iff $\forall a \in dom(x), c_\emptyset \oplus c_x(a) < k$ and $\exists b \in dom(x) \mid c_x(b) = 0$. Some other consistencies introduced for WCSP are AC* [9, 12], FDAC [3], EDAC [6], VAC and OSAC [4]. Algorithms enforcing such properties are based on equivalence-preserving transformations (EPT) that allow safe moves of costs among constraints: the cost of any complete instantiation is preserved. Two basic cost transfer operations are called project and unaryProject (see e.g., [4]). The former projects a given cost from a non-unary soft constraint to a unary constraint; for example, it is possible to project on $c_x(a)$ the *minimum* cost of a value (x,a) on a soft constraint c_S, which is $min_{I \in l(S) \wedge I[x]=a} c_S(I)$. The latter projects a given cost from a unary constraint to the nullary constraint $c_\emptyset$. We shall note $\phi(P)$ the enforcement of property ϕ (e.g., AC, EDAC, ...) on the (W)CN P. For non-binary soft constraints, generalized arc consistency (GAC), a well-known CSP property, has also been adapted to WCSP [5, 4]. We first need to introduce the notion of extended cost. The *extended* cost of an instantiation $I \in l(S)$ on a soft constraint c_S, includes the cost of I on c_S as well as the nullary cost $c_\emptyset$ and the unary costs for I of the variables in S. It is defined by $ecost(c_S, I) = c_\emptyset + \sum_{x \in S} c_x(I[x]) + c_S(I)$; we shall say that I is *allowed* on c_S iff $ecost(c_S, I) < k$.

Definition 1. *A soft constraint c_S is GAC-consistent iff:*

- $\forall I \in l(S)$, $c_S(I) = k$ *if* $ecost(c_S, I) = k$.
- *for every value* (x,a) *of* c_S, $\exists I \in l(S) \mid I[x] = a \wedge c_S(I) = 0$.

Below, we propose an alternative to this definition of GAC for WCSP, and call it weak GAC (GAC^w for short).

Definition 2. *A value (x,a) of a soft constraint c_S is GAC^w-consistent on c_S iff $\exists I \in l(S) \mid I[x] = a \wedge c_S(I) = 0 \wedge ecost(c_S, I) < k$. A soft constraint c_S is GAC^w-consistent iff every value of c_S is GAC^w-consistent.*

GAC is stronger than GAC^w because it identifies instantiations of constraint scopes that are inconsistent. However, when domains are the only point of interest, we can observe that the set of values deleted when enforcing GAC on a soft constraint c_S is exactly the set of values deleted when enforcing GAC^w on c_S.

Finally, one alternative approach to cost transfer methods is the algorithm PFC-MRDAC [8, 11, 10]. This is a classical branch and bound algorithm that computes lower bounds at each node of the search tree and that is used in our experimentation.

3 GAC^w-WSTR

The algorithm we propose, called GAC^w-WSTR, can be applied to any soft table constraint c_S whose default cost is either 0 or k. Such constraints occur quite frequently in practice. For example, among the 31 packages of WCSP instances listed on `http://costfunction.org`, 19 packages contain instances where the default cost of all soft table constraints is 0, and 5 packages contain instances where a various proportion of soft table constraints have a default cost equal to 0. This means that our approach can be applied on more than 61% of the packages currently available on this website. In this section, we first describe the data structures, then we introduce the algorithm GAC^w-WSTR, and finally we study its properties (correctness and complexity).

3.1 Data structures

A soft table constraint c_S is a constraint defined by a list $table[c_S]$ of t tuples[1] (built over S), a list $costs[c_S]$ of t integers, and an integer $default[c_S]$. The ith tuple in $table[c_S]$ is given as cost the ith value in $costs[c_S]$. Any *implicit* tuple, i.e., any tuple that is not present in $table[c_S]$, is given as (default) cost the value $default[c_S]$.

An important feature (inherited from STR) of the algorithm we propose is the cheap restoration of its structures when backtracking occurs. The principle is to split each constraint table into different sets such that each tuple is a member of exactly one set. One of these sets contains all tuples that are currently valid: tuples in this set constitute the content of the *current table*. For simplicity, data structures related to backtracking are not detailed in this paper (see [13]).

For a (soft) constraint table c_S, the following arrays provide access to the disjoint sets of valid and invalid tuples within $table[c_S]$:

- $position[c_S]$ is an array of size t that provides indirect access to the tuples of $table[c_S]$. At any given time, the values in $position[c_S]$ are a permutation of $\{1, 2, \ldots, t\}$. The ith tuple of c_S is $table[c_S][position[c_S][i]]$, and its cost is given by $costs[c_S][position[c_S][i]]$.
- $currentLimit[c_S]$ is the position of the last current tuple in $table[c_S]$. The current table of c_S is composed of exactly $currentLimit[c_S]$ tuples. The values in $position[c_S]$ at indices ranging from 1 to $currentLimit[c_S]$ are positions of the current tuples of c_S.

The top half of Figure 1 illustrates the use of our data structures for a given constraint. The array $table$ is composed of 7 tuples (ranging in lexicographic order from τ_0 to τ_6). For each tuple, an associated cost is given by the array $costs$. The array $position$ provides an indirect access to the tuples. The last valid tuple of the table is marked by a pointer: $currentLimit$. Initially all tuples of the table are valid and the current table is composed of exactly $currentLimit$ tuples. In our example the value of $c_\emptyset$ is set to 0 and the upper-bound k is set to 5. The data structure c_1 represents the unary cost of each value (x, a). The data structure $position''$ represents the state of $position$ after applying our algorithm. Changes in this data structure will be explained in Section 3.2.

[1] A tuple can be seen as an instantiation over the variables of the scope of a constraint.

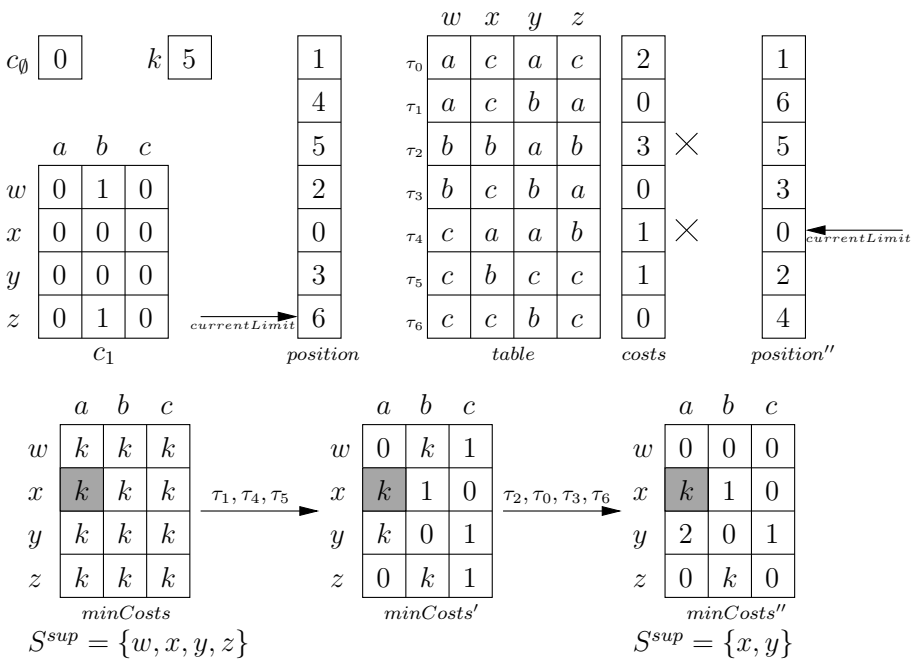

Fig. 1. Example of our data structures and their evolution after removing a from $dom(x)$

As in [13], we also introduce two sets of variables, called S^{sup} and S^{val}. On the one hand, as soon as all values in the domain of a variable have been detected GAC^w-consistent, it is futile to continue to seek supports for values of this variable. We therefore introduce the set S^{sup} of uninstantiated variables (from the scope of the constraint) whose domains contain each at least one value for which a support has not yet been found. All main operations in our algorithm will only handle variables in S^{sup}. To update S^{sup}, we use an array to count the number $nbGacValues[c_S][x]$ of GAC^w-consistent values identified for each variable x.

On the other hand, at the end of an invocation of GAC^w-WSTR for a constraint c_S, we know that for every variable $x \in S$, every tuple τ such that $\tau[x] \notin dom(x)$ has been removed from the current table of c_S. If there is no backtrack and $dom(x)$ does not change between this invocation and the next invocation, then at the time of the next invocation it is certainly true that $\tau[x] \in dom(x)$ for every tuple τ in the current table of c_S. In this case, there is no need to check whether $\tau[x] \in dom(x)$; efficiency is gained by omitting this check. We implement this optimization by means of set S^{val}, which is the set of uninstantiated variables whose domain has been reduced since the previous invocation of GAC^w-WSTR. To set up S^{val}, we need to record the domain size of each variable $x \in S$ right after the execution of GAC^w-WSTR on c_S: this value is recorded in $lastSize[c_S][x]$.

For enforcing GAC^w on a given constraint c_S, we need to compute minimum costs of values on c_S. This can be achieved at low cost while traversing the current table of c_S.

We just need an array $minCosts[c_S]$ to record those minimum costs; $minCosts[c_S][x][a]$ will denote the minimum cost of (x, a) on c_S Finally, when the default cost of c_S is 0, it is useful to count the number $nbTuples[c_S][x][a]$ of valid explicit tuples for each value (x, a), so as to determine whether a valid implicit tuple may exist.

To conclude this section, we briefly discuss how transfers of tuple costs can be implemented. Actually, to keep unchanged our table representation (i.e., keeping the same list of explicit tuples), we need to adopt the solution proposed in [5, 12], which has a reasonable $O(|S|d)$ complexity. The principle is to keep original values in $costs[c_S]$ while recording in an auxiliary structure called $deltas[c_S]$ the cumulated cost of all projections performed with respect to each value. The current cost of a given tuple τ (at position $index$ in our representation) is then computed as follows: $c_S(\tau) = costs[c_S][index] \ominus_{x \in S} deltas[c_S][x][\tau[x]]$.

3.2 Algorithm

Whereas STR for crisp table constraints just requires simple iterations, for soft table constraints, we have to handle several potential iterations over table tuples (due to cost transfer operations). This is what we show now with Algorithm 10 that enforces GAC^w on any soft table constraint c_S whose default cost is either 0 or k. The first instruction is a call to Function initialize, Algorithm 5, which initializes both sets S^{sup} and S^{val}. More precisely, S^{sup} is initialized to contain future variables only, which is exactly $S \setminus past(P)$; the set $past(P)$ denotes the set of variables of the WCN P explicitly instantiated by the search algorithm. The set S^{val} contains the future variables whose domains have been changed since the last call to the algorithm (for the same constraint c_S). At line 6 of Algorithm 5, we have $|dom(x)|$ which is the size of the current domain of x, and $lastSize[c_S][x]$ which is the size of the domain of x, the last time the specific constraint c_S was processed; initially we have $lastSize[c_S][x] = -1$ for every pair composed of a constraint c_S and a variable x in S. Additionally, S^{val} also contains the last assigned variable, denoted by $lastPast(P)$ here, if it belongs to the scope of the constraint c_S. Indeed, after any variable assignment $x = a$, some tuples may become invalid due to the removal of values from $dom(x)$. The last assigned variable is the only instantiated variable for which validity operations must be performed.

First, let us assume that $default[c_S] = k$. In this case, a call to Function traverse-k, Algorithm 7, is performed at line 4 of Algorithm 10. Lines 1-4 of Algorithm 7 allow us to initialize the arrays $nbGacValues[c_S]$ and $minCosts[c_S]$: initially, no value has been proved to be GAC^w-consistent and no tuple with a cost lower than k has been found. Then the loop at lines $6 - 24$ successively processes all current tuples of the table of c_S. At line 10 of Algorithm 7, a validity check is performed on tuple τ when $cnt > 2$ (the operator 'or else' uses a short-circuit evaluation). This means that validity checks are only performed during the two first traversals of the table (i.e., two first calls to traverse-k) because after the second traversal, no other value can be deleted. The validity check is performed by Function isValidTuple, Algorithm 3, that deals only with variables in S^{val}. At line 11 of Algorithm 7, the extended cost of τ is computed (see Algorithm 4) and compared with k when $cnt > 1$. This means that such a computation is only performed during the first traversal of the table because the extended cost of any tuple on c_S remains constant after that traversal (projections do not modify extended

costs as shown by Lemma 1). If tuple τ (whose cost is γ) is both valid and allowed, the array $minCosts[c_S]$ is subject to potential update (line 16). Besides, when the cost γ of τ is 0, we know that we have just found a support for (x, a) on c_S, so we can increment $nbGacValues[c_S][x]$ (line 18), and discard x from S^{sup} (line 20) in case (x, a) was the last value in $dom(x)$ without any proved support. In constant time at line 23 a tuple τ that is either invalid or forbidden is removed from the current table: actually it is moved to the end of the current table before the value of $currentLimit[c_S]$ is decremented.

The next instruction of Algorithm 10 is a call to Function pruneVars (Algorithm 6). This function allows us to remove all values proved to be inconsistent wrt GAC^w: they are the values (x, a) such that $minCosts[c_S][x][a] = k$. When at least one value is removed from the domain of a variable x, x is added to the set Y^{val} (line 6). Besides, after processing x, the value of $lastSize[c_S][x]$ is updated to record the new domain size (line 9). In Y^{sup}, we only keep variables for which at least one support must be sought (line 8). Finally, the sets Y^{sup} and Y^{val} become the new S^{sup} and S^{val}. Note that S^{sup} and S^{val} are not handled exactly as in STR2 [13], the difference residing mainly in Algorithm 6.

The main loop of Algorithm 10 aims at making successive projections in order to exhibit a support for each remaining value of c_S. All variables in S^{sup} require such operations. Each such variable is picked in turn at line 9, and projections (potentially followed by a unary projection) are performed at lines 10-16. If S^{sup} still contains at least one variable, the counter cnt is incremented, and traverse-k is called again. This new call permits to update the array $minCosts[c_S]$ as well as the set S^{sup}.

Now, let us assume that $default[c_S] = 0$. In that case, Function traverse-0 (Algorithm 9) is called instead of Function traverse-k, at lines 6 and 22 of Algorithm 10. The main difference between traverse-0 and traverse-k is that forbidden tuples must be kept in order to be able to count the number of valid tuples in the current table. Counting is managed at lines 5 and 16. Once the current table has been iterated over, we need to look for the existence of a valid implicit tuple for each value (lines 27-31). Function allowedImplicitTuple determines whether there exists such a valid implicit tuple containing the value (x, a).

The bottom half of Figure 1 illustrates the evolution of the data structure $minCosts$ during a call to Algorithm 7. We suppose that the event $x \neq a$ has triggered a reconsideration of the constraint. Note that before calling Algorithm 7 the structure $minCosts$ has been initialized (with value k) using Algorithm 5 and the set S^{sup} contains all the unassigned variables involved in the scope of the constraint. First, tuple τ_1 is considered. This tuple is valid (all values of the tuple belong to the current domains) and the tuple is also allowed since the extended cost of the tuple is equal to 0 (which is less than $k = 5$). Then the data structure $minCosts$ is updated for each value of τ_1. Next τ_4 is considered. Due to $x \neq a$, τ_4 is no more valid. So τ_4 is swapped with the last valid tuple and the pointer $currentLimit$ is decremented. In the figure, a cross identifies the removed tuple. The minimal cost of (x, a) remains k. The structure $minCosts'$ depicts the state of $minCosts$ after considering tuples τ_1, τ_4 and τ_5. Next when considering τ_2, this tuple is identified as not allowed because its extended cost is equal to $k = 5$. Hence, τ_2 is removed. The structure $minCosts''$ represents the state of $minCosts$ after considering all tuples. Finally, as all values of w have a minimal cost equal to 0, it means

Algorithm 1: project(c_S: soft constraint, x: variable, a: value, α: integer)

$c_x(a) \leftarrow c_x(a) \oplus \alpha$
$deltas[c_S][x][a] \leftarrow deltas[c_S][x][a] \oplus \alpha$;

Algorithm 2: unaryProject(x: variable, α: integer)

foreach *value* $a \in dom(x)$ **do**
    $c_x(a) \leftarrow c_x(a) \ominus \alpha$
$c_\emptyset \leftarrow c_\emptyset \oplus \alpha$

Algorithm 3: isValidTuple(c_S: soft constraint, τ: tuple): Boolean

foreach *variable* $x \in S^{val}$ **do**
    if $\tau[x] \notin dom(x)$ **then**
        return *false*
return *true*

Algorithm 4: ecost(c_S: soft constraint, γ: integer, τ: tuple): integer

return $c_\emptyset \bigoplus_{x \in S} c_x(\tau[x]) \oplus \gamma$

Algorithm 5: initialize(c_S: soft constraint)

$S^{sup} \leftarrow \emptyset$; $S^{val} \leftarrow \emptyset$
if $lastPast(P) \in S$ **then**
    $S^{val} \leftarrow S^{val} \cup \{lastPast(P)\}$
foreach *variable* $x \in S \mid x \notin past(P)$ **do**
    $S^{sup} \leftarrow S^{sup} \cup \{x\}$
    if $|dom(x)| \neq lastSize[c_S][x]$ **then**
        $S^{val} \leftarrow S^{val} \cup \{x\}$

Algorithm 6: pruneVars(c_S: soft constraint)

$Y^{sup} \leftarrow \emptyset$, $Y^{val} \leftarrow \emptyset$
foreach *variable* $x \in S^{sup}$ **do**
    foreach $a \in dom(x)$ **do**
        if $minCosts[c_S][x][a] = k$ **then**
            remove a from $dom(x)$
            add x to Y^{val}
        else if $minCosts[c_S][x][a] > 0$ **then**
            add x to Y^{sup}
    $lastSize[c_S][x] \leftarrow |dom(x)|$
$S^{val} \leftarrow Y^{val}$; $S^{sup} \leftarrow Y^{sup}$

Algorithm 7: traverse-k(c_S: soft constraint, cnt: integer)

```
1  foreach variable x ∈ S^sup do
2      nbGacValues[c_S][x] ← 0
3      foreach a ∈ dom(x) do
4          minCosts[c_S][x][a] ← k
5  i ← 1
6  while i ≤ currentLimit[c_S] do
7      index ← position[c_S][i]
8      τ ← table[c_S][index]                    // current tuple
9      γ ← costs[c_S][index] ⊖_{x∈S} deltas[c_S][x][τ[x]]      // tuple cost
10     valid ← cnt > 2 or else isValidTuple(c_S, τ)
11     allowed ← cnt > 1 or else ecost(c_S, γ, τ) < k
12     if valid ∧ allowed then
13         foreach variable x ∈ S^sup do
14             a ← τ[x]
15             if γ < minCosts[c_S][x][a] then
16                 minCosts[c_S][x][a] ← γ
17                 if γ = 0 then
18                     nbGacValues[c_S][x] + +
19                     if nbGacValues[c_S][x] = |dom(x)| then
20                         S^sup ← S^sup \ {x}
21         i ← i + 1
22     else
23         swap(position[c_S], i, currentLimit[c_S])
24         currentLimit[c_S] − −
```

that all values have at least a support in this constraint. The variable w can then be safely removed from S^{sup}. One can apply a similar reasoning for the variable z. Actually, the value (z, b) will be removed when Algorithm 6 is called (since its minimal cost is equal to k), and since all the other values have a support then z can be safely removed from S^{sup}. Note that if τ_2 had not been removed (by omitting to compute its extended cost), the minimal cost of (z, b) would have been 3 (instead of k). Consequently this value would not have been removed. After the execution of Algorithms 7 and 6, the set S^{sup} only contains variables x and y.

3.3 Properties

Lemma 1. *Let c_S be a soft constraint, and (x, a) be a value of c_S. The extended cost of every tuple $\tau \in l(S)$ remains constant, whatever the operation* project(c_S, x, a, α) *or* unaryProject(x, α) *is performed (Proof omitted)*

Under our assumptions, a preliminary observation is that we do not have to keep track of the effect of projections project(c_S, x, a, α) on the default cost. Indeed, if

Algorithm 8: allowedImplicitTuple(c_S: soft constraint, x: variable, a: value)

```
foreach τ ∈ l(S) | τ[x] = a do
    if ¬binarySearch(τ, table[c_S]) then
        return true
return false
```

Algorithm 9: traverse-0(c_S: soft constraint, cnt: integer)

```
foreach variable x ∈ S^sup do
    nbGacValues[c_S][x] ← 0
    foreach a ∈ dom(x) do
        minCosts[c_S][x][a] ← k
        nbTuples[c_S][x][a] ← 0
i ← 1
while i ≤ currentLimit[c_S] do
    index ← position[c_S][i]
    τ ← table[c_S][index]                          // current tuple
    γ ← costs[c_S][index] ⊖_{x∈S} deltas[c_S][x][τ[x]]      // tuple cost
    valid ← cnt > 2 or else isValidTuple(c_S, τ)
    allowed ← ecost(c_S, γ, τ) < k
    if valid then
        foreach variable x ∈ S^sup do
            a ← τ[x]
            nbTuples[c_S][x][a] + +
            if allowed ∧ γ < minCosts[c_S][x][a] then
                minCosts[c_S][x][a] ← γ
                if γ = 0 then
                    nbGacValues[c_S][x] + +
                    if nbGacValues[c_S][x] = |dom(x)| then
                        S^sup ← S^sup \ {x}
        i ← i + 1
    else
        swap(position[c_S], i, currentLimit[c_S])
        currentLimit[c_S] − −
foreach variable x ∈ S^sup do
    nb ← |Π_{y∈S|y≠x} dom(y)|
    foreach a ∈ dom(x) do
        if nbTuples[c_S][x][a] ≠ nb ∧ minCosts[c_S][x][a] > 0 then
            if allowedImplicitTuple(c_S, x, a) then
                minCosts[c_S][x][a] ← 0
```

Algorithm 10: GAC^w-WSTR(c_S: soft constraint)

```
1  initialize(c_S)
2  cnt ← 1
3  if default[c_S] = k then
4  |  traverse-k(c_S, cnt)
5  else
6  |  traverse-0(c_S, cnt)                    // default[c_S] = 0
7  pruneVars(c_S)
8  while S^sup ≠ ∅ do
9  |  pick and delete x from S^sup
10 |  α ← +∞
11 |  foreach a ∈ dom(x) do
12 |  |  if minCosts[c_S][x][a] > 0 then
13 |  |  |  project(c_S, x, a, minCosts[c_S][x][a])
14 |  |  α ← min(α, c_x(a))
15 |  if α > 0 then
16 |  |  unaryProject(x, α)
17 |  if S^sup ≠ ∅ then
18 |  |  cnt++
19 |  |  if default[c_S] = k then
20 |  |  |  traverse-k(c_S, cnt)
21 |  |  else
22 |  |  |  traverse-0(c_S, cnt)              // default[c_S] = 0
```

$default[c_S] = k$, we have $k \ominus \alpha = k$ and if $default[c_S] = 0$ a projection is only possible when no implicit tuple exists with $x = a$.

Proposition 1. *Algorithm 10 enforces GAC^w on any soft table constraint c_S such that $default[c_S] = k$.*

Proof. Let (x, a) be a value of c_S (before calling Algorithm 10), and let the value $\alpha = minCosts[c_S][x][a]$ be obtained (for the minimum cost of (x, a) on c_S) just before executing line 7 of Algorithm 10. On the one hand, if $\alpha = k$ then it means that there is no explicit valid tuple τ in the current table such that $\tau[x] = a \wedge ecost(c_S, \tau) < k$ (because all explicit tuples have just been iterated over by Function traverse-k called at line 4). Besides, as the default cost is k, there is no implicit tuple τ such that $ecost(c_S, \tau) < k$. We can conclude that (x, a) is inconsistent w.r.t. GAC^w. This is the reason why when pruneVars is called at Line 7, this value (x, a) is removed (see Line 5 of Algorithm 6). On the other hand, if $\alpha < k$, it means that there exists a non-empty set X of valid tuples τ such that $\tau[x] = a \wedge ecost(c_S, \tau) < k$. Let us first consider the call to Function pruneVars at line 7. For every value (y, b) removed at line 5 of Algorithm 6, we have $minCosts[c_S][y][b] = k$, which implies that for every $\tau \in X$, we have $\tau[y] \neq b$ (otherwise $minCosts[c_S][y][b]$ would have been α). Consequently, all tuples in X remain valid and allowed after the execution of pruneVars. Those tuples, present in X,

will remain valid and allowed throughout the execution of the algorithm because after executing pruneVars, no more values can be deleted, and cost transfer operations do not modify extended costs (see Lemma 1). This guarantees that all values detected inconsistent by GAC^w are deleted during the call to pruneVars. Now, for (x, a), either $minCosts[c_S][x][a]$ incidentally becomes 0 by means of cost transfers concerning variables other than x, or $0 < minCosts[c_S][x][a] < k$ at the moment where x is picked at line 9 of Algorithm 10. When executing lines 11-14, all values of x are made GAC^w-consistent. So, this is the case for (x, a). We have just proved that every deleted value is inconsistent w.r.t. GAC^w, and that every remaining value is GAC^w-consistent. □

Proposition 2. *Algorithm 10 enforces GAC^w on any soft table constraint c_S such that $default[c_S] = 0$.*

The proof (omitted here) is similar to that of Proposition 1, with the additional consideration of implicit valid tuples. Notice that Algorithm 10 enforces both GAC^w and NC on any soft table constraint whose default cost is either k or 0.

We now discuss the complexity of GAC^w-WSTR for a given constraint c_S. With $r = |S|$ being the arity of c_S, the space complexity is $O(tr)$ for structures $table[c_S]$ and $costs[c_S]$, $O(t)$ for $position[c_S]$, $O(r)$ for S^{sup}, S^{val}, $lastSize[c_S]$ and $nbGac\ Values[c_S]$, $O(rd)$ for $minCosts[c_S]$, $nbTuples[c_S]$ and $deltas[c_S]$. Overall, the worst-case space complexity is $O(tr+rd)$. The time complexity is $O(r)$ for initialize, $O(rd+tr)$ for traverse-k and $O(rd)$ for pruneVars. Importantly, the number of turns of the main loop starting at line 8 of Algorithm 10 is at most r because a variable can never be put two times in S^{sup}; the complexity for one iteration is $O(d)$ for lines 10-16 augmented with that of traverse-k or traverse-0. Overall, the worst-case time complexity of GAC^w-WSTR when $default[c_S] = k$ is $O(r^2(d + t))$. On the other hand, the time complexity of allowedImplicitTuple is $O(rt \log(t))$ because the loop at line 1 of Algorithm 8 is executed at most t times. Indeed, each call to binarySearch is $O(r \log(t))$ and the loop is stopped as soon as a valid tuple cannot be found in the table. traverse-0 is $O(rd+tr)$ for lines 1-26 and $O(r^2dt \log(t))$. Finally, the worst-case time complexity of GAC^w-WSTR when $default[c_S] = 0$ is $O(r^3dt \log(t))$.

4 Benchmarks

We have performed a first experimentation using a new series of Crossword instances called *crossoft*, which can be naturally represented by soft table constraints. Given a grid and a dictionary, the goal is to fill the grid with words present in the dictionary. To generate those instances, we used three series of grids (Herald, Puzzle, Vg) and one dictionary, called OGD, that contains common nouns (with a cost of 0) and proper nouns (with a cost r, where r is the length of the word). Penalties are inspired from the profits associated with words as described on the french web site `http://ledefi.pagesperso-orange.fr`.

We have performed a second experimentation using random WCSP instances. We have generated different classes of instances by considering the CSP model RB [17]. With some well-chosen parameters, Theorem 2 in [17] holds: an asymptotic phase transition is guaranteed at a precise threshold point. CSP instances from model RB were

translated into WCSP instances by associating a random cost (between 1 and k) with each forbidden CSP tuple, and considering a default cost for implicit tuples equal to 0. This guarantees the hardness of the generated random WCSP instances. Using $k = 10$, we generated 5 series of 10 WCSP instances of arity 3; rb-r-n-d-e-t-s is an instance of arity r with n variables, domain size d and e r-ary constraints of tightness t (generated with seed s). A second set was obtained by translating random CSP instances (with arity equal to 10) into WCSP. Such a translation was also used for the series *renault-mod*.

Next, we have performed experimental trials with a new series of instances called *poker* based on the version *Texas hold 'em* of poker. The goal is to fill an empty 5×5 board with cards taken from the initial set of cards so as to obtain globally the best hands in each row and column. The model used to generate the instances is the following: there is a variable per cell representing a card picked in the initial set of cards. In *Poker-n* the initial set of cards contains n cards of each suit and only combinations of at least 2 cards are considered. Of course, putting the same card several times on the grid is forbidden. The cost of each hand is given below:

Royal Flush	Straight Flush	Four of a Kind	Full House	Flush	Straight	Three of a Kind	Two Pairs	Pair	High Card
0	1	2	3	4	5	6	7	8	9

Finally, we have experimented our approach on real-world series from `http://costfunction.org/en/benchmark`. We have used the *ergo* and the *linkage* series which are structured WCSP instances with constraints of arity larger than 3.

5 Experimental Results

In order to show the practical interest of our approach to filter soft table constraints of large arity, we have conducted an experimentation (with our solver AbsCon) using a cluster of Xeon 3.0GHz with 1GiB of RAM under Linux. We have implemented a version of PFC-MRDAC, where minimum costs (required by the algorithm to compute lower bounds) are obtained by calling Functions traverse-0 and traverse-k. At each step of the search, only one call to either Function traverse-0 or Function traverse-k is necessary for each soft table constraint because PFC-MRDAC does not exploit cost transfer operations (due to lack of space, we cannot give full details). This version will be called PFC-MRDAC-WSTR, whereas the classical version will be called here PFC-MRDAC-GEN. We have also implemented the algorithm GAC^w-WSTR and embedded it in a backtrack search algorithm that maintains GAC^w. This search algorithm is also able to maintain AC* and FDAC, instead of GAC^w. Note that PFC-MRDAC-GEN, "maintaining AC*" and "maintaining FDAC" iterate over all valid tuples in order to compute lower bounds or minimum costs. To control such iterations (that are exponential with respect to the arity of constraints), we have pragmatically tuned a parameter that delays the application of the algorithm until enough variables are assigned. We have

conducted an experimentation on the benchmarks described in the previous section. A time-out of $1,200$ seconds was set per instance. The variable ordering heuristic was $wdeg/dom$ [2] and the value ordering heuristic selected the value with minimal cost.

The overall results are given in Table 1. Each line of this table corresponds to a series of instances: *crossoft-ogd*, *rand-3*, *ergo*,... The total number of instances for each series is given in the second column of the table. For each series, we provide the number of solved instances (optimum proved) by each method within 20 minutes. For series *crossoft*, the algorithms PFC-MRDAC-WSTR and "maintaining GAC^w-WSTR" solve more instances than the generic algorithms. We obtain the same kind of results with *poker*. Unsurprisingly, the STR approaches are not so efficient on RB series (*rand-3*), which can be explained by the low arity of the constraints (which are ternary) involved in these instances. On random problems with high arity (involving 10 variables) results are clearly better: generic algorithms can not solve any of these instances. Finally, for *ergo* and *linkage* series, results are not so significant. Indeed these instances have either constraints with low arity or variable domains with very few values (for example, the maximum domain size is 2 for the instance *cpcs422b*). When the size of variable domains is small, Cartesian products of domains grow slowly with the constraint arity, and so generic algorithms iterating over valid tuples can still be competitive.

		PFC-MRDAC-		Maintaining-		
$Series$	$\#Inst$	WSTR	GEN	GAC^w-WSTR	AC*	FDAC
crossoft-herald	50	33	10	**47**	11	11
crossoft-puzzle	22	**22**	9	**22**	18	18
crossoft-vg	64	**14**	6	**14**	7	7
poker	18	**10**	2	**10**	5	5
rand-3 (rb)	48	20	29	20	**32**	30
rand-10	20	**20**	0	**20**	0	0
ergo	19	13	10	15	15	**17**
linkage	30	0	0	0	1	**9**
renault-mod	50	**50**	32	**50**	**50**	47

Table 1. Number of solved instances per series (a time-out of 1,200 seconds was set per instance).

Table 2 focuses on some selected instances with the same comparison of algorithms. We provide an overview of the results in terms of CPU time (in seconds). On instances of series *crossoft* and *poker*, our approach (PFC-MRDAC-WSTR and GAC^w-WSTR) outperforms the generic ones whatever the envisioned solving approach (i.e., with or without cost transfer) is. Note that results for maintaining AC* and FDAC are quite close. Instances of these two problems have constraints with high arity and variables with rather large domains. Therefore, the STR technique is well-adapted. Note that for various instances, generic approaches can not find and prove optimum solutions before the time limit whereas STR-based algorithms solve them in a few seconds.

Instances	PFC-MRDAC-WSTR	PFC-MRDAC-GEN	Maintaining-GAC^w-WSTR	Maintaining-AC*	Maintaining-FDAC
crossoft-ogd-15-09	26.5	$> 1,200$	**25.2**	273	269
crossoft-ogd-23-01	$> 1,200$	$> 1,200$	**565**	$> 1,200$	$> 1,200$
crossoft-ogd-puzzle-18	**6.29**	$> 1,200$	6.66	$> 1,200$	$> 1,200$
crossoft-ogd-vg-5-6	**0.4**	155	0.77	31.5	32.3
poker-5	0.26	92.4	**0.24**	1.39	1.5
poker-6	**0.38**	463	0.39	6.58	6.99
poker-9	0.79	$> 1,200$	**0.63**	782	1022
poker-12	1.51	$> 1,200$	**0.89**	$> 1,200$	$> 1,200$
rb-3-12-12-30-0.630-0	4.13	1.2	3.61	**0.79**	0.86
rb-3-16-16-44-0.635-2	94.3	7.41	51.6	**2.31**	3.11
rb-3-20-20-60-0.632-0	614	34.4	830	24.3	**23.3**
pedigree1	$> 1,200$	$> 1,200$	890	819	**35.0**
barley	$> 1,200$	$> 1,200$	40.7	23	**20.3**
cpcs422b	**7.4**	113	8.61	58.3	111
link	68.6	$> 1,200$	5.41	**4.55**	6.5
rand-10-20-10-5-9	3.94	$> 1,200$	**2.39**	$> 1,200$	$> 1,200$
rand-10-20-10-5-10	5.27	$> 1,200$	**2.67**	$> 1,200$	$> 1,200$
renault-mod-12	1.74	680	**1.39**	6.01	14.4
renault-mod-14	2.49	$> 1,200$	**1.49**	6.83	14.9

Table 2. CPU time (in seconds) to prove optimality on various selected instances (a time-out of 1,200 seconds was set per instance).

6 Conclusion

In this paper, we have introduced a filtering algorithm that enforces a form of generalized arc consistency, called GAC^w, on soft table constraints. This algorithm combines simple tabular reduction and cost transfer operations. The experiments that we have conducted show the viability of our approach when soft table constraints have large arity, whereas usual generic soft consistency algorithms are not applicable to their full extent. The algorithm we propose can be applied to any soft table constraint with a default cost of either 0 or k, which represents a large proportion of practical instances. A direct perspective of this work is to generalize our approach to soft table constraints with any default cost.

Acknowledgments

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

References

1. D. Allouche, C. Bessiere, P. Boizumault, S. de Givry, P. Gutierrez, S. Loudni, J.-P. Métivier, and T. Schiex. Decomposing global cost functions. In *Proceedings of AAAI'12*, 2012.
2. F. Boussemart, F. Hemery, C. Lecoutre, and L. Sais. Boosting systematic search by weighting constraints. In *Proceedings of ECAI'04*, pages 146–150, 2004.
3. M. Cooper. Reduction operations in fuzzy or valued constraint satisfaction. *Fuzzy Sets and Systems*, 134(3):311–342, 2003.
4. M. Cooper, S. de Givry, M. Sanchez, T. Schiex, M. Zytnicki, and T. Werner. Soft arc consistency revisited. *Artificial Intelligence*, 174(7-8):449–478, 2010.
5. M.C. Cooper and T. Schiex. Arc consistency for soft constraints. *Artificial Intelligence*, 154(1-2):199–227, 2004.
6. 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.
7. A. Favier, S. de Givry, A. Legarra, and T. Schiex. Pairwise decomposition for combinatorial optimization in graphical models. In *Proceedings of IJCAI'11*, pages 2126–2132, 2011.
8. E.C. Freuder and R.J. Wallace. Partial constraint satisfaction. *Artificial Intelligence*, 58(1-3):21–70, 1992.
9. J. Larrosa. Node and arc consistency in weighted CSP. In *Proceedings of AAAI'02*, pages 48–53, 2002.
10. J. Larrosa and P. Meseguer. Partition-Based lower bound for Max-CSP. *Constraints*, 7:407–419, 2002.
11. J. Larrosa, P. Meseguer, and T. Schiex. Maintaining reversible DAC for Max-CSP. *Artificial Intelligence*, 107(1):149–163, 1999.
12. J. Larrosa and T. Schiex. Solving weighted CSP by maintaining arc consistency. *Artificial Intelligence*, 159(1-2):1–26, 2004.
13. C. Lecoutre. STR2: Optimized simple tabular reduction for table constraint. *Constraints*, 16(4):341–371, 2011.
14. J. Lee and K. Leung. Towards efficient consistency enforcement for global constraints in weighted constraint satisfaction. In *Proceedings of IJCAI'09*, pages 559–565, 2009.
15. J. Lee and K. Leung. A stronger consistency for soft global constraints in weighted constraint satisfaction. In *Proceedings of AAAI'10*, pages 121–127, 2010.
16. J.R. Ullmann. Partition search for non-binary constraint satisfaction. *Information Science*, 177:3639–3678, 2007.
17. K. Xu, F. Boussemart, F. Hemery, and C. Lecoutre. Random constraint satisfaction: easy generation of hard (satisfiable) instances. *Artificial Intelligence*, 171(8-9):514–534, 2007.