The Smart Table Constraint

Jean-Baptiste Mairy[1], Yves Deville[1], and Christophe Lecoutre[2]

[1] ICTEAM, Université catholique de Louvain, Belgium
{jean-baptiste.mairy , yves.deville}@uclouvain.be
[2] CRIL-CNRS UMR 8188, Université d'Artois, F-62307 Lens, France
lecoutre@cril.fr

Abstract. Table Constraints are very useful for modeling combinatorial problems in Constraint Programming (CP). They are a universal mechanism for representing constraints, but unfortunately the size of their tables can grow exponentially with their arities. In this paper, we propose to authorize entries in tables to contain simple arithmetic constraints, replacing classical tuples of values by so-called smart tuples. Smart table constraints can thus be viewed as logical combinations of those simple arithmetic constraints. This new form of tuples allows us to encode compactly many constraints, including a dozen of well-known global constraints. We show that, under a very reasonable assumption about the acyclicity of smart tuples, a Generalized Arc Consistency algorithm of low time complexity can be devised. Our experimental results demonstrate that the smart table constraint is a highly promising general purpose tool for CP.

Table constraints explicitly express the allowed combinations of values for the variables they involve as sets of tuples, which are called tables. Table constraints can theoretically encode any kind of constraints and are amongst the most useful ones in Constraint Programming (CP). Indeed, they are often required when modeling combinatorial problems in many application fields. The design of filtering algorithms for such constraints has generated a lot of research effort, see [2, 19, 16, 8, 27, 15, 14, 21, 24]. The biggest problem with table constraints are their size. Several approaches have been proposed to reduce this size. Two of them modify the definition of classical tuples: compressed tuples [12, 25, 30] and short supports applied to table constraints [10]. Compressed tuples allow tuples entries to contain sets. A compressed tuple thus represents all the tuples in the cartesian product of the sets. Short supports applied to table constraints allow variables to be left out of the short tuple. Left-out variables can take any values from their domains. In this paper, we propose to generalize both compressed tuples and short supports in table constraints by authorizing tuples to contain simple arithmetic constraints. We call such tuples *smart tuples*, and tables containing smart tuples *smart tables*. For instance, the following set of tuples $\{(1,2,1), (1,3,1), (2,2,2), (2,3,2), (3,2,3), (3,3,3)\}$ on variables $\{x_1, x_2, x_3\}$ with domains $\{1,2,3\}$ can be represented by a smart table containing only one smart tuple:

x_1	x_2	x_3
$= x_3$	≥ 2	$*$

or in an equivalent form by $(x_1 = x_3, x_2 \geq 2)$. A symbol $*$ in the tabular form of a smart tuple means that, if not occurring anywhere else, the corresponding variable is not constrained at all by the tuple (which is not the case here).

As a motivating example of the interest of smart tables, let us consider a car configuration problem. We assume that the cars to configure have 2 colors (one for the body, $colB$, and the other for the roof, $colR$), a model number $modNum$, an option pack $optPack$ and an onboard computer $comp$. A configuration rule might state that, for a particular model number a and some fancy body color set S, an option pack less than a certain pack b implies that the onboard computer cannot be the most powerful one, c, and that the roof color has to be the same as the body color. This configuration constraint can be written as:

$$modNum = a \wedge colB \in S \wedge optPack < b \Rightarrow comp \neq c \wedge colR = colB$$

The encoding of this constraint with a smart table consists of four smart tuples: $(modNum \neq a), (colB \notin S), (optPack \geq b)$ and $(comp \neq c, colR = colB)$, which gives under tabular form:

$modNum$	$colB$	$colR$	$optPack$	$comp$
$\neq a$	$*$	$*$	$*$	$*$
$*$	$\notin S$	$*$	$*$	$*$
$*$	$*$	$*$	$\geq b$	$*$
$*$	$*$	$= colB$	$*$	$\neq c$

Encoding this constraint with classical tuples is exponentially larger, and even using compressed tuples or short supports results in a table that is strictly longer because none of these techniques can be used to encode compactly the relation existing between $colB$ and $colR$ (they require, for this case, one distinct tuple for each possible color). On the other hand, using reification (decomposition by adding auxiliary variables) of the configuration rule does not guarantee the same level of pruning as the smart table encoding since there is a cycle to handle.

Importantly, smart table constraints can be viewed as a disjunction of conjunctions of basic arithmetic constraints. Indeed, each smart tuple contains a conjunction of basic arithmetic constraints and the table is a disjunction of such tuples, since the variables can satisfy any of the smart tuples. Filtering of logical combinations of constraints has already been studied in the literature [1, 3, 4, 29, 28, 11, 18, 17, 9]. However, the particular form of our smart tuples leads to a filtering algorithm with a low polynomial time complexity. More precisely, we show how Simple Tabular Reduction (STR) [27, 15] can be adapted for smart table constraints to produce an efficient filtering procedure to enforce Generalized Arc Consistency. Smart table constraints can be viewed as a subset of the logic algebra defined in [1]: we impose a particular form on the logical combinations (disjunction of conjunction, conjunctions forming acyclic networks) and we restrict the constraints that can be combined to be simple arithmetic constraints. The rules for the filtering are however directly derived from [1]. While restricting the expressive power with respect to [1], smart tuples greatly increase the expressive power of classical tuples.

More importantly, those restrictions allows the introduction of a propagator computing efficiently the GAC maximal inconsistent sets (which is not always the case for the general logic algebra from [1]). The novelty in our approach lies in the introduction of such concrete propagator for such a subset of the logic algebra. Those restrictions also allows the filtering of smart tuples to be much simpler than the filtering for general conjunctions defined in [3, 11, 18]. The pruning achieved by the disjunction is equivalent to the pruning of constructive disjunction [29, 28]. The propagation of a whole table constraint can even be seen as the propagation of a large constructive disjunction. The ability to leave variables out of the constraints in the smart tuple makes their filtering as efficient as the improved constructive disjunction filtering defined in [18]. In [4], the authors propose a filtering for constructive disjunction based on indexicals as well as a stronger filtering, considering disjunctions together with other constraints. In this paper, we do not investigate propagating more than one table constraint at a time. The filtering for both conjunctions and disjunctions proposed in this paper is stronger than the light filtering proposed in [9], thanks to the hypothesis made on the form of the smart tuples.

1 Defining Smart Table Constraints

A Constraint Satisfaction Problem (CSP) P is composed of an ordered set of *variables* $X = \{x_1, \ldots, x_n\}$, where each variable x has a domain of possible values denoted by $dom(x)$, and a set of *constraints* $C = \{c_1, \ldots, c_e\}$, where each constraint c corresponds to a relation denoted by $rel(c)$ on a subset of variables of X ; this subset is called the *scope* of c and denoted by $scp(c)$. Each constraint c defines the possible combinations of values satisfying c in $rel(c)$. The *arity* of a constraint c is $\#\text{scp}(c)$, i.e., the number of variables involved in c. The largest arity is denoted by r, while the size of the largest domain is denoted by d.

A *literal* is a variable value pair (x, a) such that $x \in X$. A literal of a constraint c is a literal (x, a) such that $x \in scp(c)$. A literal (x, a) is *valid* iff $a \in dom(x)$. A *tuple on* an (ordered) subset of variables $Y = \{y_1, \ldots, y_p\} \subseteq X$ is a sequence of literals $((y_1, a_1), \ldots, (y_p, a_p))$, one for each variable $y \in Y$. When there is no ambiguity about Y, we simply write $(a_1, \ldots, a_p)$. A tuple is *valid* iff all its literals are valid. A tuple τ is *allowed* by a constraint c iff $\tau \in rel(c)$. A tuple τ *satisfies* a set of constraints C' iff for every constraint $c' \in C'$, $\tau[scp(c')]$ is allowed by c', where $\tau[Y]$ denotes the restriction of τ on literals referring to variables in Y. The set of solutions of a CSP $P = (X, C)$ is denoted by $sols(P)$; these are the valid tuples on X that satisfy C.

A *table constraint* is a constraint whose semantics is defined in extension by listing the set of allowed (or forbidden) tuples. These tuples are *classical*. In this paper, we introduce smart table constraints. A *smart table constraint* sc is defined semantically from a set of smart tuples, called *smart table* and denoted by $table(sc)$. A *smart tuple* σ is a set of tuple constraints, where a *tuple constraint* can take four possible forms:

1. <var> <op> a
2. <var> $\in S$, <var> $\notin S$
3. <var> <op> <var>
4. <var> <op> <var> + b

where <var> is a variable in the scope of the smart table constraint, a and b some constants, S a set of constants and <op> an operator in the set $\{<, \leq, =, \neq, \geq, >\}$.

The semantics of smart table constraints is simple and natural: a classical tuple τ is allowed by a smart table constraint sc iff there exists at least one smart tuple $\sigma \in table(sc)$ such that τ satisfies σ. Note that when a variable $x \in \text{scp}(sc)$ is not involved in any tuple constraint of a smart tuple $\sigma \in table(sc)$ then x can take any value in its domain; such a variable is said to be *unrestricted* on σ and the set of unrestricted variables on σ is denoted by $unres(\sigma)$. Note also that any classical tuple $(a_1, \ldots, a_r)$ on a set of variables $\{x_1, \ldots, x_r\}$ can be re-written as the smart tuple $\{x_1 = a_1, \ldots, x_r = a_r\}$.

As seen in the introduction, smart tuples can help modeling constraints in a compact and natural way, when disjunction is needed. Smart table constraints can also be used to encode some global constraints. The encodings of *Lex*, *Max* and *Element* are smart table constraint versions of the ones proposed in [1]. In the examples below, tuple constraints are written directly inside the tables to ease reading. A tuple constraint of the form x_i <op> a (resp. x_i <op> $x_j + b$) is written as <op> a (resp. <op> $x_j + b$) in the column of the table corresponding to x_i. The following global constraints illustrate the modeling power of the smart table constraint. Their equivalent with classical tuples are exponentially larger. For instance, in the table for element, each smart tuple corresponds to d^m classical tuples. For compressed tuples, if only one variable is the target of all the tuple constraints, each smart tuple can be translated as d compressed tuples. This is the case for all the global constraint presented below except for *Lex*. For this constraint, the smart table is $O(d^m)$ times smaller than the table using compressed tuples. Short supports applied to table constraint can only encode efficiently unrestricted variables, making the encoding of each smart tuple $O(d^m)$ tuples with short supports for *Lex*, *Max* and *AtMost1*. Global constraints are of course not the sole purpose of the smart table constraints but being able to encode efficiently those constraints has many advantages.

Lex$([x_1, \ldots, x_m], [y_1, \ldots, y_m])$***:*** $\bar{\boldsymbol{x}} > \bar{\boldsymbol{y}}$

x_1	x_2	$\ldots$	x_m	y_1	y_2	$\ldots$	y_m
$> y_1$	$*$	$\ldots$	$*$	$*$	$*$	$\ldots$	$*$
$= y_1$	$> y_2$	$\ldots$	$*$	$*$	$*$	$\ldots$	$*$
$\ldots$	$\ldots$	$\ldots$	$\ldots$	$\ldots$	$\ldots$	$\ldots$	$\ldots$
$= y_1$	$= y_2$	$\ldots$	$> y_m$	$*$	$*$	$\ldots$	$*$

Max $([x_1, x_2, \ldots, x_m], M)$: $max(\bar{\mathbf{x}}) = M$

x_1	x_2	$\ldots$	x_m	M
$*$	$\leq x_1$	$\ldots$	$\leq x_1$	$= x_1$
$\leq x_2$	$*$	$\ldots$	$\leq x_2$	$= x_2$
$\ldots$	$\ldots$	$\ldots$	$\ldots$	$\ldots$
$\leq x_m$	$\leq x_m$	$\ldots$	$*$	$= x_m$

Element$(I, [x_1, x_2, \ldots, x_m], R)$*:* $X[I] = R$

I	x_1	x_2	$\ldots$	x_m	R
$= 1$	$*$	$*$	$\ldots$	$*$	$= x_1$
$= 2$	$*$	$*$	$\ldots$	$*$	$= x_2$
$\ldots$	$\ldots$	$\ldots$	$\ldots$	$\ldots$	$\ldots$
$= m$	$*$	$*$	$\ldots$	$*$	$= x_m$

AtMost1$([x_1, \ldots, x_m], Y)$*:* $\#\{1 \leq i \leq m | x_i = Y\} \leq 1$

x_1	x_2	$\ldots$	x_m	Y
$*$	$\neq Y$	$\ldots$	$\neq Y$	$*$
$\neq Y$	$*$	$\ldots$	$\neq Y$	$*$
$\ldots$	$\ldots$	$\ldots$	$\ldots$	$\ldots$
$\neq Y$	$\neq Y$	$\ldots$	$*$	$*$

NotAllEqual$(x_1, \ldots, x_m)$*:* $\exists 1 \leq i, j \leq m : x_i \neq x_j$

x_1	x_2	x_3	$\ldots$	x_m
$*$	$\neq x_1$	$*$	$\ldots$	$*$
$*$	$*$	$\neq x_1$	$\ldots$	$*$
$\ldots$	$\ldots$	$\ldots$	$\ldots$	$\ldots$
$*$	$*$	$*$	$\ldots$	$\neq x_1$

Diffn$([x_1, \ldots, x_m], [i_1, \ldots, i_m], [y_1, \ldots, y_m], [j_1, \ldots, j_m])$*:*
no overlap between orthotopes defined in $\mathbb{R}^m$ from points $\bar{\mathbf{x}}$ and $\bar{\mathbf{y}}$ with lengths along axes of $\bar{\mathbf{i}}$ and $\bar{\mathbf{j}}$ respectively.

x_1	x_2	$\ldots$	x_m	y_1	y_2	$\ldots$	y_m
$*$	$*$	$\ldots$	$*$	$\geq x_1 + i_1$	$*$	$\ldots$	$*$
$\geq y_1 + j_1$	$*$	$\ldots$	$*$	$*$	$*$	$\ldots$	$*$
$*$	$*$	$\ldots$	$*$	$*$	$\geq x_2 + i_2$	$\ldots$	$*$
$*$	$\geq y_2 + j_2$	$\ldots$	$*$	$*$	$*$	$\ldots$	$*$
$\ldots$	$\ldots$	$\ldots$	$\ldots$	$\ldots$	$\ldots$	$\ldots$	$\ldots$
$*$	$*$	$\ldots$	$*$	$*$	$*$	$\ldots$	$\geq x_m + i_m$
$*$	$*$	$\ldots$	$\geq y_m + j_m$	$*$	$*$	$\ldots$	$*$

2 Filtering Smart Table Constraints

This section presents a filtering algorithm to establish GAC on smart table constraints. GAC is a property that relies on the concept of support. A *support* of a constraint c is a tuple on $scp(c)$ which is both valid and allowed by c. A support for a literal (x, a) of c is a support of c containing (x, a).

Definition 1. *A constraint c is Generalized Arc Consistent (GAC) iff all the literals of the constraint have a support in c. A CSP is GAC iff all its constraints are GAC.*

In general, identifying the set of supports of a constraint allows us to enforce GAC. Actually, for any smart table constraint sc, each smart tuple σ corresponds to a small CSP $P_\sigma = (X_\sigma, C_\sigma)$, with $X_\sigma = \text{scp}(sc)$ and $C_\sigma = \sigma$. The classical tuples that are supports of sc from σ are exactly the solutions in $sols(P_\sigma)$. Hence, the full set of supports of sc is equal to $\bigcup_{\sigma \in table(sc)} sols(P_\sigma)$. This is similar to the way set of supports are computed for constructive disjunction.

Our objective is to efficiently identify and remove valid literals of sc without any support. It may seem costly to compute $sols(P_\sigma)$ for every smart tuple σ. Obtaining the set of supports for an arbitrary logical combination of constraints is NP-hard [1]. However, we impose that the constraint graph of any CSP P_σ that is associated with a smart tuple σ, is acyclic and P_σ is a conjunction. This restriction allows an efficient processing of the smart tuples when used for filtering.

Property 1. Let σ be a smart tuple of a smart table constraint, P'_σ, the GAC closure of P_σ, is globally consistent, i.e., each literal of P'_σ appears in at least one solution of P_σ.

This property is derived from [20] and the acyclic nature of the constraint graphs defined by smart tuples. This means that the set of literals appearing in $sols(P_\sigma)$ can be obtained by simply applying GAC on P_σ.

Obtaining the GAC closure on each of the P_σ and taking their union at the end to have the set of supported literals corresponds exactly to an application of the filtering rules defined in [1], when seeing the smart table constraint as a logical combination of basic arithmetic constraints. The acyclic nature of the conjunctions in the smart tuples guarantees that the set of supported literals computed by this procedure is the set of GAC literals for the logical combination of arithmetic constraints by Theorem 3 in [1]. Hence, this procedure is correct and computes the GAC literals for the smart table constraint. Moreover, the complexity of filtering P_σ can also benefit from the form of the smart tuples, as expressed below.

Property 2. The GAC closure of an acyclic binary CSP can be obtained in $O(e \cdot F)$, where filtering an individual constraint if $O(F)$.

The procedure for obtaining the GAC closure of an acyclic binary CSP $P = (X, C)$ is the following. The CSP forms a forest (possibly, with only one tree), and each tree of the forest can be filtered independently since no variable is shared between trees. For each tree T, revising constraints in turn from the deepest ones to the shallowest ones, and then the other way around, achieves GAC on T. Each constraint in C is thus revised two times (no fixed point needed). Revising a constraint c consists in removing

the literals that have no support on c. We call this procedure GAC_tree. GAC_tree, as well as properties 1 and 2, are not original to this work, but they justify the filtering procedure of the smart table constraints.

Applying GAC_tree to a smart tuple σ of a constraint sc requires decomposing σ according to its connected components; the result of this decomposition will be denoted by forest(σ). More precisely, for each subset $cc \subseteq \sigma$ that represents a connected component, there is an associated tree T in forest(σ) that defines an independent sub-CSP (X_T, C_T) with $X_T = vars(cc)$ and $C_T = cc$. We shall refer to such sub-CSPs with tree shape as *treeCSPs*. An additional *void* tree T defining a trivial sub-CSP (X_T, C_T) with $X_T = unres(\sigma)$ and $C_T = \emptyset$ is introduced if $unres(\sigma) \neq \emptyset$. This guarantees that $sols(P_\sigma) = \Pi_{T \in trees(\sigma)} sols(T)$, which results from the independence of the trees w.r.t each other.

The filtering algorithm proposed for smart table constraints, called smartSTR, works with the decompositions into treeCSPs instead of working directly with the smart tuples. It is inspired from STR (Simple Tabular Reduction) [27, 15]. STR works by scanning constraint tables, going through each tuple sequentially. The validity of each row is checked. When a row is not valid, it is removed from the table. Otherwise, all the literals of the row are marked as having a support. After scanning the whole table, all the literals for which no support has been found are removed. The difference between STR and smartSTR is the way validity checks and the collection of supported literals are performed. A smart tuple σ is valid iff P_σ admits at least one solution. A smart tuple σ is thus valid iff each treeCSP in forest(σ) admits at least one solution. The literals supported by σ are the literals in $sols(P_\sigma)$ (obtained with GAC_tree), computed as the union of the supported literal sets of each individual treeCSP in forest(σ).

Algorithm 1 presents the pseudo-code of smartSTR. It uses a data structure sl that contains all the literals without any found support (sl stands for support-less). Line 3 initializes sl with all valid literals (no support has been found yet). Then the algorithm loops over all the smart tuples of the constraint (line 4). The test at line 5 checks the validity of the current smart tuple by testing the validity of all its treeCSPs. If the smart tuple σ is valid, each of its independent treeCSP removes from sl the literals they support (loop at lines 6-7). The loop at line 8 empties the sets sl of all unrestricted variables on σ, as there is no restriction on those variables (actually, this corresponds to dealing with the void tree that is not in practice included in forest(σ)). If the smart tuple is invalid, it is removed from the table (line 9); the table of the constraint is represented using a sparse set, as in STR1 and STR2. After going through all the smart tuples of the constraint, smartSTR removes the literals that are still left without a support (loop at line 10).

As seen in Algorithm 1, each treeCSP is responsible to check its validity and to remove from sl the literals it supports. This is done through `isValid` and `collect` methods. Their specifications can be found in Interface 1, called TreeCSP. Note that a treeCSP involves a set of variables $vars$ and belongs to the forest of a smart tuple σ.

From now on, the treeCSPs that are composed of only one constraint will be called *branches*. In the code presented below, we have specific classes for *unary* branches (containing a tuple constraint of the form <var> <op> a, or <var> $\in S$), and *binary* branches (containing a tuple constraint of the form <var> <op> <var> , or <var>

```
smartSTR(SmartTableConstraint sc):
    // post: the constraint sc is GAC
    forall x ∈ scp(sc): sl(sc)[x] ← dom(x)
    forall σ ∈ table(sc):
        if (∧_{T∈forest(σ)} T.isValid()):
            forall T ∈ forest(σ):
                T.collect(sl(sc))
            forall x ∈ unres(σ): sl(sc)[x] ← ∅
        else: remove σ from table(sc)
    forall x ∈ scp(sc):
        dom(x) ← dom(x) \ sl(sc)[x]
```

Algorithm 1: smartSTR

```
interface TreeCSP
  fields: Variable[] vars
  isValid()
    // post: returns true iff the treeCSP is valid
  collect(Set{Value}[] sl)
    // pre:  the smart tuple σ, such that the treeCSP
    //       is in forest(σ), is valid
    // post: ∀x ∈ vars, ∀a ∈ dom(x), (x, a) has a
    //       support in the treeCSP ⇒ a ∉ sl[x]
```

Interface 1: Interface for treeCSPs

<op> <var> + b). There is one unary and binary branch class for each value of <op> . We also introduce one class for *simple* trees (trees of height 1 consisting of multiple branches all sharing the same root variable) and another one for *general* trees (trees of height > 1).

Algorithm 2 presents the classes introduced for unary branches with operations $=$, and $<$. The pseudo-code for the other operators are very similar. The additional method `filterX`, not contained in Interface 1, is responsible to filter the (pseudo) domain Dx given as argument. It is used by simple and general trees, where GAC has to be enforced on several branches. Dx is used to avoid filtering directly dom(x), because the effective filtering can only be done when all smart tuples have been processed.

Algorithms 3 and 4 present the classes introduced for binary branches with operations $=$ and $<$, respectively. Again, the pseudocodes for the other operators are very similar. In those pseudocodes, $S \oplus b$, where S is a set and b a value, represents the addition of the constant to all the values in the set. They all implement the method `filterX`, as unary branches do, but with two parameters (Dx and Dy). Dx is the copy of the domain of x to filter and Dy is a domain for y to use to filter Dx. The second parameter is needed during the execution of GAC_tree to use an already filtered copy

```
1  class UnaryBranchEq:TreeCSP /* x = a */
2      fields: Variable x, Value a
3      isValid(): return a ∈ dom(x)
4      collect(sl): sl[x] ← sl[x] \ {a}
5      filterX(Dx): Dx ← Dx ∩ {a}
6
```

```
1  class UnaryBranchLt:TreeCSP /* x < a */
2      fields: Variable x, Value a
3      isValid(): return min(dom(x)) < a
4      collect(sl):
5          sl[x] ← sl[x] \ {b ∈ dom(x) : b < a}
6      filterX(Dx): Dx ← Dx \ {b ∈ Dx : b ≥ a}
7
```

Algorithm 2: Classes for unary branches $=$ and $<$

of the domain of y to filter the copy of dom(x). Again, the filtering of the real domains of the variables can only occur after all the smart tuples have been processed. Those classes also implement a `filterY` method which is the counterpart of `filterX` for y. They implement a method `collectY`, used by simple trees to collect values, but only for the second involved variable y with respect to a (pseudo) domain Dx, given as a parameter, for the first involved variable x. It is called after Dx, which is initially a copy of dom(x), has been filtered through the entire simple or general tree.

```
1  class BinaryBranchEq:TreeCSP /* x = y + b */
2      fields: Variable x, y
3              Value b
4      isValid(): return dom(x) ∩ dom(y) ⊕ b ≠ ∅
5      collect(sl):
6          I ← dom(x) ∩ dom(y) ⊕ b
7          sl[x] ← sl[x] \ I
8          sl[y] ← sl[y] \ I
9      collectY(sl, Dx):
10         I ← Dx ∩ dom(y) ⊕ b
11         sl[y] ← sl[y] \ I
12     filterX(Dx,Dy): Dx ← Dx ∩ Dy ⊕ b
13     filterY(Dx,Dy): Dy ← Dx ∩ Dy ⊕ b
14
```

Algorithm 3: Classes for binary branche $=$

Algorithm 5 gives the pseudo-code for simple trees, where all involved branches share the same root variable x (see the assertion at line 4). Since we can change the

```
1   class BinaryBranchLt:TreeCSP /* x < y + b */
2       fields: Variable x, y
3               Value: b
4       isValid(): return min(dom(x)) < max(dom(y)) + b
5       collect(sl):
6           sl[x] ← sl[x] \ {a ∈ dom(x) : a < max(dom(y)) + b}
7           sl[y] ← sl[y] \ {c ∈ dom(y) : c > min(dom(x)) − b}
8       collectY(sl, Dx):
9           sl[y] ← sl[y] \ {c ∈ dom(y) : c > min(Dx) − b}
10      filterX(Dx,Dy):
11          Dx ← Dx \ {a ∈ Dx : a ≥ max(Dy) + b}
12      filterY(Dx,Dy):
13          Dy ← Dy \ {c ∈ Dy : c ≤ min(Dx) − b}
14
```

Algorithm 4: Classes for binary branches $<$

order of the variables in binary branches ($x_1 < x_2 \rightarrow x_2 > x_1$, etc.), this is not a requirement on the form of the smart tuples. This is enforced at the creation of the smart tuples trees. The validity test at line 5 starts by making a copy Dx of dom(x). Then, Dx is filtered through all branches (loop starting at line 7). The unary branches are treated at lines 8-9 and the binary ones, at lines 10-11. For the binary branches, `filterX` is called with the full domain of y as argument for the copy of y's domain. If Dx does not become empty, that means that the simple tree has at least one solution. The method `collect` at line 13 uses Dx (which has already been filtered by `isValid`). Since all values in Dx have a support in the simple tree, they are removed from sl$[x]$ (line 14). The loop at line 15 goes through every binary branch (i.e., with a scope containing 2 variables) to collect the supported values for the second involved variables (y) from the filtered domain Dx. The supported values for variables y are directly removed from sl instead of copying their domain and filtering it. Note that methods `isValid` and `collect` are adaptations of the two pass filtering GAC_tree. During the first pass, only the domain of x (actually, Dx) is filtered. Indeed, as it may change at each new processed branch, filtering the domains of variables y (actually, updating sl) is useless at that time. The validity test is not concerned by the second pass because if x still has values in its domain after the first pass, the simple tree is guaranteed to have at least one solution.

The class for general trees is given in Algorithm 6. This algorithm uses several fields. The array `allVars` contains all the variables appearing in the tree. The 2 dimensional array `branches` contains all the branches for each level of the tree, from 1 (branches containing the root variable) to `treeHeight`. The array `domCopy` contains the copies of the domains of the variables of the tree that are used during the procedure GAC_tree. For this algorithm, we will suppose that, for all the binary branches, the variable x is always the closest to the root. This is again enforced during the creation of the smart tuples trees. The assertion of line 5 thus checks that all the variables x have a corresponding variable as y at the level below (closer to the root). The `isValid` method

```
1   class SimpleTree:TreeCSP
2       fields: Variable x, TreeCSP[] branches,
3               Domain Dx
4       assert ∀T ∈ branches : T.x = x
5       isValid():
6           Dx ← dom(x)
7           forall T ∈ branches:
8               if #T.vars = 1
9                   T.filterX(Dx)
10              else
11                  T.filterX(Dx,dom(y))
12          return Dx ≠ ∅
13      collect(sl):
14          sl[x] ← sl[x] \ Dx
15          forall T ∈ branches : #T.vars = 2
16              T.collectY(sl,Dx)
17
```

Algorithm 5: Class for simple trees

(line 7) realizes the first pass of GAC_tree (using copies of the domains), filtering the domains of the different x variables from the leafs to the root. If the variable at the root (branches$[1][1].x$) of the tree still has values, it returns true. Its `collect` method (line 17) then achieves the second pass by filtering the (copies of the) domains of the y variables of the branches. It also removes supported values from sl. At this point, it is important to note that the code presented for unary branches, binary branches and simple trees already covers all the examples given in this paper.

We now study the complexity of our approach. The complexity of filtering a smart tuple depends on the complexity of filtering each of its treeCSPs, as they are independent. For a smart tuple σ (on variables with maximal domain size d), the time complexities for the different operators are:

- for the unary branches:

<op>	isValid	collect	filterX
$=$	$O(1)$	$O(1)$	$O(1)$
$\neq$	$O(1)$	$O(1)$	$O(1)$
$>\geq<\leq$	$O(1)$	$O(d)$	$O(d)$
$\in$	$O(d)$	$O(d)$	$O(d)$

- for the binary branches:

<op>	isValid	collect/ collectY	filterX/ filterY
$=$	$O(d)$	$O(d)$	$O(d)$
$\neq$	$O(1)$	$O(1)$	$O(1)$
$>\geq<\leq$	$O(1)$	$O(d)$	$O(d)$

```
1   class GeneralTree:TreeCSP
2       fields: Variable[] allVars, TreeCSP[][] branches,
3                 Value treeHeight, Domain[] domCopy
4       assert ∀1 < l ≤ treeHeight, ∀b ∈ branches[l],
5                ∃b_2 ∈ branches[l − 1] : b.x = b_2.y
6       isValid():
7           forall x ∈ allVars :
8               domCopy[x] ← dom(x)
9           forall l ∈ treeHeight..1 :
10              forall T ∈ branches[l]:
11                  if #T.vars = 1
12                      T.filterX(domCopy[T.x])
13                  else
14                      T.filterX(domCopy[T.x], domCopy[T.y])
15          return domCopy[branches[1][1].x] ≠ ∅
16      collect(sl):
17          forall l ∈ 1..treeHeight :
18              forall T ∈ branches[l]:
19                  sl[T.x] ← sl[T.x] \ domCopy[T.x]
20                  if #T.vars = 2:
21                      T.filterY(domCopy[T.x], domCopy[T.y])
22          forall T ∈ branches[treeHeight] : #T.vars = 2
23              sl[T.y] ← sl[T.y] \ domCopy[T.y]
24
```

Algorithm 6: Class for general trees

Each tuple constraint is either its own tree or belongs to a larger tree. If the branch is its own tree, the time complexities of `isValid` and `collect` are $O(d)$ for any operator. If the branch is included in a simple or general tree, then GAC_tree guarantees that the `collectY`, `filterX` and `filterY` methods are called a constant number of times. The time complexity imputable to the branch is thus $O(d)$ for validity testing and value collection. This makes the treatment of one smart tuple with k tuple constraints $O(k \cdot d + r)$, where r is the arity of the table constraint. The last term comes from the treatment of unrestricted variables. The initialization of sl at the beginning of smartSTR and the actual filtering of the domains at the end are $O(r \cdot d)$. The total time complexity of one call to smartSTR for a smart table constraint of arity r with t smart tuples is thus $O(r \cdot d + t \cdot k \cdot d + t \cdot r)$. For a classical table constraint of arity r with t' tuples, we have that STR2 has a time complexity of $O(r \cdot d + t' \cdot r)$. In all the examples given, we have that $k \leq r$ (less tuple constraints than variables). We also have that the number of smart tuples is at least $d+1$ times less than the number of classical tuples. In those conditions, the complexity of filtering the smart table is less than the complexity of using STR2 on the table without smart tuples. Indeed, we have $t \cdot k \cdot d + t \cdot r \leq t' \cdot r$.

3 Experimental Results

Optimization present in STR2 can also be included in smartSTR. The obtained algorithm is then called smartSTR2. Comparing SmartSTR2 with all specialized algorithms developed over the years for the global constraints mentioned earlier is clearly beyond the scope of this paper. However, we shall show the interest of SmartSTR2 on a few case studies. Comparing a propagator F with SmartSTR2 on a global constraint means that, in the same CSP, all the instances of the global constraint are either propagated with F or their encoding in smart table constraint is propagated with SmartSTR2. We have conducted an experimentation (with the solver AbsCon) on a laptop computer, equipped with Intel(R) Core(TM) i7-2820QM CPU @ 2.30GHz, under Linux. Results are given in seconds, or corresponds to number of visited nodes per second. We have checked that all tested approaches were traversing the exact same search trees (most of the time using dom/ddeg as variable ordering heuristic for this purpose).

In natural language processing, one task is to determine whether a given sentence is well-formed (i.e., to what extent, it respects a grammar). A constraint model (R. Coletta, personal communication) has been recently developed for this problem, denoted here by TAL. It involves the Element constraint (with R as a variable as described earlier in the paper). Instances for this optimization problem are defined by entering an input sentence. In this model, Element constraints represent about 8% of the constraints. We compare SmartSTR2 with GACElt that corresponds to the GAC propagator based on watched literals [7]. In this context, the two algorithms are very close in term of performance as shown by Table 1.

sentence	GACElt	SmartSTR2
phrase1	3.6	3.7
phrase2	17.6	17.9
phrase3	54.4	54.2
phrase4	46.8	46.8
phrase5	82.4	82.6

Table 1. CPU time to solve *TAL* instances.

A *BIBD* is a standard combinatorial problem. We consider here the model introduced in [22] and the series of instances tested in [5]. There is a lexicographic constraint between any two adjacent rows or columns. We compare SmartSTR2 with GACLex that corresponds to the filtering procedure described in [13] and is a variant of [6]. Table 2 shows the results we have obtained with both algorithms. Interestingly, one can observe that replacing the specialized propagator GACLex with the general-purpose SmartSTR2 has a very limited cost. Similar results are obtained with the social golfer problem.

The *RectanglePacking* problem [26] consists of packing all squares from size 1×1 to $n \times n$ into a rectangle of size $w \times h$. We adopt the model and search parameters given in [23, 10]. Table 3 shows that SmartSTR2 is very efficient on this problem. It

v-b-r-k-λ	GACLex	SmartSTR2
6-50-25-3-10	1.3	1.6
6-60-30-3-12	1.5	2.1
6-70-35-3-10	2.2	2.8
10-90-27-3-6	5.8	7.3
9-108-36-3-9	11.4	14.2
15-70-14-3-2	7.4	7.9
12-88-22-3-4	7.0	8.3
9-120-40-3-10	17.9	25.1
10-120-36-3-8	10.6	14.0
13-104-24-3-4	99.1	108.6

Table 2. CPU time to solve *BIBD* instances.

n-w-h	GAC-valid	ShortSTR2	SmartSTR2
18-31-69	1,821	2,784	57,249
19-47-53	2,003	3,166	57,221
20-34-85	1,324	1,579	45,600
21-38-88	849	1,295	40,600
22-39-88	981	1,035	41,162
23-64-68	983	1,292	40,495
24-56-88	446	790	32,758
25-43-129	661	347	30,544
26-70-89	544	703	31,374
27-47-148	326	175	26,786

Table 3. Nodes searched per second for *RectanglePacking* instances.

clearly outperforms ShortSTR2, and seems to be at least as efficient as the other methods proposed in [23] (not implanted in our system) when we compare their results with ours. Note that GAC-valid (sometimes called GAC-schema) is another general approach, given here as a baseline.

For our last experiment, we consider Case Study 4 in [10], where a problem, denoted by *AllDistinctVectors* here, involves the VectorDiff constraint. An instance *p-a-d* of this problem has exactly p vectors (arrays of variables), each vector of length a and each variable with a domain whose size is equal to d: any pair of vectors must be distinct. In [10], it has been shown that ShortSTR2 is an interesting competitor to HaggisGAC. When we consider Boolean variables only (i.e., $d = 2$), SmartSTR2 is slightly slower than ShortSTR2 (because tables are small). However, when we increase d, Table 4 shows that, just when applying GAC stand-alone, SmartSTR2 is clearly superior to ShortSTR2. This can be explained by the size of the constraint tables. For example, for *40-100-40*, tables contain $156,000$ and 100 tuples in ShortSTR2 and SmartSTR2, respectively.

p-a-d	ShortSTR2	SmartSTR2
40-100-2	0.07	0.07
40-100-8	1.55	0.18
40-100-16	6.49	0.18
40-100-24	14.7	0.19
40-100-32	28.1	0.20
40-100-40	44.5	0.21

Table 4. CPU time to enforce GAC on *AllDistinctVectors* instances.

4 Conclusion

Smart tuples generalize (classical) tuples in tables of constraints, as well as short and compressed tuples. They allow a compact and natural representation of many constraints, including important global constraints. Smart table constraints can be seen as a subset of the logical algebra defined in [1]. Restricting smart table constraints to this subset allows an efficient filtering of the constraints. The contribution of this paper is to introduce the smart table constraint and propose a practical GAC filtering algorithm for it. Its practical interest is also demonstrated. We do believe that there exist many optimisations and extensions to this work that still deserve to be explored.

References

1. Fahiem Bacchus and Toby Walsh. Propagating logical combinations of constraints. In *IJCAI*, pages 35–40, 2005.

2. C. Bessiere and J. Régin. Arc consistency for general constraint networks: preliminary results. In *Proceedings of IJCAI'97*, pages 398–404, 1997.
3. Christian Bessiere and Jean-Charles Régin. Local consistency on conjunctions of constraints. In *Proceedings of ECAI'98 Workshop on Non-binary constraints*, pages 53–59, 1998.
4. Björn Carlson and Mats Carlsson. Compiling and executing disjunctions of finite domain constraints. In *Proceedings of ICLP'95*, pages 117–131, 1995.
5. A. Frisch, B. Hnich, Z. Kiziltan, I. Miguel, and T. Walsh. Global constraints for lexicographic orderings. In *Proceedings of CP'02*, pages 93–108, 2002.
6. A. Frisch, B. Hnich, Z. Kiziltan, I. Miguel, and T. Walsh. Propagation algorithms for lexicographic ordering constraints. *Artificial Intelligence*, 170(10):803–834, 2006.
7. Ian P. Gent, Chris Jefferson, and Ian Miguel. Watched literals for constraint propagation in minion. In *Proceedings of CP 2006*, pages 182–197. Springer-Verlag, 2006.
8. Ian P. Gent, Chris Jefferson, Ian Miguel, and Peter Nightingale. Data structures for generalised arc consistency for extensional constraints. In *Proceedings of AAAI 07*, pages 191–197, 2007.
9. Christopher Jefferson, Neil CA Moore, Peter Nightingale, and Karen E Petrie. Implementing logical connectives in constraint programming. *Artificial Intelligence*, 174(16):1407–1429, 2010.
10. Christopher Jefferson and Peter Nightingale. Extending simple tabular reduction with short supports. In *Proceedings of IJCAI'13*, pages 573–579, 2013.
11. George Katsirelos and Fahiem Bacchus. GAC on conjunctions of constraints. In *Proceedings of CP'01*, pages 610–614, 2001.
12. George Katsirelos and Toby Walsh. A compression algorithm for large arity extensional constraints. In *Proceedings of CP'07*, pages 379–393, 2007.
13. C. Lecoutre. *Constraint networks: techniques and algorithms*. ISTE/Wiley, 2009.
14. C. Lecoutre, C. Likitvivatanavong, and R.H.C. Yap. A path-optimal GAC algorithm for table constraints. In *Proceedings of ECAI'12*, pages 510–515, 2012.
15. Christophe Lecoutre. STR2: optimized simple tabular reduction for table constraints. *Constraints*, 16(4):341–371, 2011.
16. Christophe Lecoutre and Radoslaw Szymanek. Generalized arc consistency for positive table constraints. In *Proceedings of CP'06*, pages 284–298, 2006.
17. O. Lhomme. Practical reformulations with table constraints. In *Proceedings of ECAI'12*, pages 911–912, 2012.
18. Olivier Lhomme. Arc-consistency filtering algorithms for logical combinations of constraints. In *Proceedings of CPAIOR'04*, pages 209–224. Springer, 2004.
19. Olivier Lhomme and Jean-Charles Régin. A fast arc consistency algorithm for n-ary constraints. In *Proceedings of AAAI'05*, pages 405–410, 2005.
20. Alan K Mackworth and Eugene C Freuder. The complexity of some polynomial network consistency algorithms for constraint satisfaction problems. *Artificial intelligence*, 25(1):65–74, 1985.
21. Jean-Baptiste Mairy, Pascal Van Hentenryck, and Yves Deville. Optimal and efficient filtering algorithms for table constraints. *Constraints*, 19(1):77–120, 2014.
22. P. Meseguer and C. Torras. Solving strategies for highly symmetric CSPs. In *Proceedings of IJCAI'99*, pages 400–405, 1999.
23. Peter Nightingale, Ian Philip Gent, Christopher Anthony Jefferson, and Ian James Miguel. Short and long supports for constraint propagation. *Journal of Artificial Intelligence Research*, 46:1–45, 2013.
24. Guillaume Perez and Jean-Charles Régin. Improving GAC-4 for table and MDD constraints. In *Proceedings of CP'14*, pages 606–621, 2014.
25. Jean-Charles Régin. Improving the expressiveness of table constraints. In *Proceedings of CP'11 Workshop on Constraint Modelling and Reformulation*, 2011.

26. H. Simonis and B. O'Sullivan. Search strategies for rectangle packing. In *Proceedings of CP'08*, pages 52–66, 2008.
27. Julian R Ullmann. Partition search for non-binary constraint satisfaction. *Information Sciences*, 177(18):3639–3678, 2007.
28. Pascal Van Hentenryck, Vijay Saraswat, and Yves Deville. Design, implementation, and evaluation of the constraint language cc(fd). *The Journal of Logic Programming*, 37(1-3):139–164, 1998.
29. Jörg Würtz and Tobias Müller. Constructive disjunction revisited. In *Proceedings of KI'96*, pages 377–386, 1996.
30. Wei Xia and Roland H. C. Yap. Optimizing STR algorithms with tuple compression. In *Proceedings of CP'13*, pages 724–732, 2013.