Automatic Synthesis of Smart Table Constraints
by Abstraction of Table Constraints

Baudouin Le Charlier¹, Minh Thanh Khong¹, Christophe Lecoutre², Yves Deville¹

¹ Université catholique de Louvain, Belgium
² CRIL-CNRS, Université d’Artois, France
{baudouin.lecharlier, minh.khong, yves.deville}@uclouvain.be, lecoutre@cril.fr

Abstract

The smart table constraint represents a powerful modeling tool that has been recently introduced. This constraint allows the user to represent compactly a number of well-known (global) constraints and more generally any arbitrarily structured constraints, especially when disjunction is at stake. In many problems, some constraints are given under the basic and simple form of tables explicitly listing the allowed combinations of values. In this paper, we propose an algorithm to convert automatically any (ordinary) table into a compact smart table. Its theoretical time complexity is shown to be quadratic in the size of the input table. Experimental results demonstrate its compression efficiency on many constraint cases while showing its reasonable execution time. It is then shown that using filtering algorithms on the resulting smart table is more efficient than using state-of-the-art filtering algorithms on the initial table.

1 Introduction

Constraint Programming (CP) is a popular paradigm to deal with combinatorial problems in Artificial Intelligence (AI). Typically, problems are modeled under the form of Constraint Networks (CNs) [Montanari, 1974], which are composed of variables to be assigned subject to constraints that must be satisfied (with possibly, an objective function to minimize or maximize). Modeling is a delicate issue, requiring from the user a certain level of expertise in order to obtain CNs that can be efficiently handled by solving systems (called constraint solvers).

Although there exists in the scientific literature a large catalog [Beldiceanu et al., 2014] of patterns of constraints, called global constraints, there are situations where the only possibility offered to the user is to enumerate the list of allowed (or disallowed) combinations of values for some specific variables, hence forming so-called table constraints. Generating table constraints can also result from the unawareness of appropriate global constraints by an unexperienced user. Finally, it is sometimes very useful to combine subsets of related constraints to form table constraints (a kind of join operation) in order to be able to reason more globally, and benefit from a better filtering of the search space.

For all reasons mentioned above, it is very common to deal with CNs that embed constraints defined extensionally by tables that may happen to be very large. Fortunately, filtering table constraints can be quite efficient as demonstrated by the recently proposed algorithm called CT [Deumeleare et al., 2016], and an independently proposed related version STRbit [Wang et al., 2016] following a decade of research effort on this topic [Lhomme and Régis, 2005; Lecoutre and Szymanek, 2006; Gent et al., 2007; Ullmann, 2007; Lecoutre, 2011; Lecoutre et al., 2012; Mairy et al., 2012]. However, as efficient as a dedicated filtering algorithm for table constraints, as CT, can be, the size of the tables is certainly penalizing when it comes to compare CT with an algorithm based on the representation of the same constraints under a more compact form when it exists.

An elegant data structure that sometimes permits a very compact representation of tables is the Multi-valued Decision Diagram (MDD) [Srinivasan et al., 1990], which is an arc-labelled directed acyclic graph (DAG) eliminating prefix and suffix redundancy. Two notable algorithms using MDDs as main data structure are mddc [Cheng and Yap, 2010] and MDD4R [Perez and Régis, 2014]. Other compression-based approaches keep the structure of tables, but replace ordinary tuples by compressed tuples [Katsirelos and Walsh, 2007; Régis, 2011; Xia and Yap, 2013] or short tuples [Nightingale et al., 2011; Jefferson and Nightingale, 2013]. Compressed tuples allow us to replace values by sets of values: a compressed tuple thus represents all the ordinary tuples from the Cartesian product of the sets. Short tuples allow some variables to be discarded, by introducing the symbol *: actually, such variables can take any values from their respective domains.

Recently, both compressed and short tuples have been generalized [Mairy et al., 2015] by smart tuples that authorize the presence of simple arithmetic constraints. As an illustration, taken from [Mairy et al., 2015], the following set of (ordinary) tuples \{(1, 2, 1), (1, 3, 1), (2, 2, 2), (2, 3, 2), (3, 2, 3), (3, 3, 3)\} on variables \(x, y, z\) that can take their values in \{1, 2, 3\} can be represented by a smart table containing only one smart tuple:

<table>
<thead>
<tr>
<th>(x)</th>
<th>(y)</th>
<th>(z)</th>
</tr>
</thead>
<tbody>
<tr>
<td>(= z)</td>
<td>(\geq 2)</td>
<td>*</td>
</tr>
</tbody>
</table>
Note that ordinary tables, and even compressed and short tables, can be exponentially larger than smart tables. Besides, as shown in [Mairy et al., 2015], smart tables can encode compactly many constraints, including a dozen of well-known global constraints. Importantly, smart table constraints correspond to a disjunction of conjunctions of basic arithmetic constraints, and can be viewed as a subset of the logic algebra defined in [Bacchus and Walsh, 2005]. Assuming the acyclicity of the (CN that can be associated with) each smart tuple, a polynomial filtering algorithm has been proposed and shown to be effective in practice. The level of filtering achieved is the property called Generalized Arc Consistency, and equivalent to that of constructive disjunction [Carlson and Carlsson, 1995; Hentenryck et al., 1998; Lhomme, 2004; 2012; Jefferson et al., 2010].

In this paper, we propose to automatically synthesize smart table constraints from table constraints. The compression algorithm is inspired by abstract interpretation and incrementally abstracts the tuples in the input table. The algorithm has a (worst case) time complexity quadratic in the size of the input table. The compression algorithm has been applied on several classes of constraints to demonstrate its compression efficiency and its quasi linear execution time on the considered benchmarks. The algorithm is also able to restrict the generated tuples to short tuples, allowing us to use existing filtering algorithms handling short tuples, such as CT* [Verhaeghe et al., 2017]. Finally, different filtering algorithms are compared on the different classes of constraints: MDD and CT (filtering using MDD and Compact table on the input table); ST (filtering on the resulting smart table); ST* and CT* (filtering on the resulting table obtained with the short tuples only option of the compression algorithm). Best results are generally obtained by ST and CT*, showing the practical interest of the compression algorithm. Moreover, the versatility of the compression algorithm to generate smart tuples or short tuples, allows to apply different existing state-of-the-art filtering algorithm.

2 Smart Table Constraints

A constraint network (CN) \( N \) is composed of a set of variables and a set of constraints. Each variable \( x \) has an associated domain, denoted by \( \text{dom}(x) \), that contains the finite set of values that can be assigned to it. The size of the largest domain is denoted by \( d \). Each constraint \( c \) involves a sequence of variables, called the scope of \( c \) and denoted by \( \text{scp}(c) \), and is semantically defined by a relation, denoted by \( \text{rel}(c) \), that contains the set of tuples allowed for the variables involved in \( c \). The arity of a constraint \( c \) is \( |\text{scp}(c)| \).

Let \( \tau = (a_1, \ldots, a_n) \) be a tuple of values associated with a sequence of variables \( \text{vars}(\tau) = \langle x_1, \ldots, x_n \rangle \). The \( i \)-th value of \( \tau \) is denoted by \( \tau[i] \) or \( \tau[x_i] \), and we say that \( \tau \) is valid iff \( \forall i \in 1..n, \tau[i] \in \text{dom}(x_i) \). The tuple \( \tau \) is a support on a constraint \( c \) such that \( \text{vars}(\tau) = \text{scp}(c) \) iff \( \tau \) is a valid tuple allowed by \( c \); we also say that \( \tau \) satisfies \( c \). If \( \tau \) is a support on a constraint \( c \) involving a variable \( x \) such that \( \tau[x] = a \), we say that \( \tau \) is a support for \( (x,a) \) on \( c \). Generalized Arc Consistency (GAC) is a well-known domain filtering property defined as follows: a constraint \( c \) is GAC iff \( \forall x \in \text{scp}(c), \forall a \in \text{dom}(x) \), there exists at least one support for \( (x,a) \) on \( c \). A CN \( N \) is GAC iff every constraint of \( N \) is GAC. Enforcing GAC is the task of removing all values that have no support on some constraint(s). A solution of \( N \) is the assignment of a value to each variable of \( N \) such that all constraints of \( N \) are satisfied; the set of solutions is denoted by \( \text{sols}(N) \).

A table constraint \( c \) is a constraint such that \( \text{rel}(c) \) is defined explicitly by listing (in an ordinary table) the ordinary tuples that are allowed\(^1\) by \( c \). A (smart) table constraint \( sc \) is defined by a smart table, denoted by \( \text{table}(sc) \), which contains a set of smart tuples. If \( \text{scp}(sc) = \langle x_1, \ldots, x_n \rangle \), then a smart tuple \( \sigma \) in \( \text{table}(sc) \) is a sequence \( \langle s_1, \ldots, s_n \rangle \) of column constraints, where a column constraint \( s_i \) can either be an unary column constraint of one of the two forms \( x_i = * \) and \( x_i < \text{op} > a \), or a binary column constraint of the form \( x_i < \text{op} > x_j \), with \( a \) being a constant and \( < \text{op} > \) an operator in \( \{<, \leq, =, \neq, >, \geq \} \).

Thus, a smart tuple contains exactly \( n \) column constraints, of which the \( i \)-th involves the variable \( x_i \), on the left. This means that we consider here a slightly simpler form of smart table constraints than in [Mairy et al., 2015], where the number of constraints is not fixed, and additional constraints of the form \( x_i \in S, x_i \notin S, \) and \( x_i < \text{op} > x_j \) are allowed. Naturally, any classical tuple \((a_1, \ldots, a_n)\) can be re-written as the smart tuple \( \langle x_1 = a_1, \ldots, x_n = a_n \rangle \). The semantics of smart table constraints is simple and natural: a (ordinary) tuple \( \tau \) is allowed by a smart table constraint \( sc \) iff there exists at least one smart tuple \( \sigma \) in \( \text{table}(sc) \) such that \( \tau \) satisfies \( \sigma \). Here, a column constraint \( x_i = * \) is always satisfied. In order to achieve GAC efficiently, the filtering algorithm described in [Mairy et al., 2015] assumes no cycle in any constraint graph that can be associated with any smart tuple.

3 The Synthesis Algorithm

The problem of synthesizing an equivalent smart table from a given table is best viewed as a set covering problem: each smart tuple is semantically equivalent to a set of (ordinary) tuples; so we aim at minimizing the number of elements of a set of smart tuples covering the given table constraint. Since the set covering problem is NP-hard, we follow a heuristic approach guided by abstract interpretation principles [Cousot and Cousot, 1977].

3.1 High-level Description of the Method

Let \( ct \) be a concrete, i.e., ordinary, table constraint of arity \( n \). In this paper, we aim at abstracting \( ct \) by a smart table \( at \) made of tuples of the form \( \langle s_1, \ldots, s_n \rangle \), as defined previously. Moreover, if \( s_i \) is of the form \( x_i < \text{op} > x_j \), we impose that \( j < i \) for reasons that will become clear shortly. The basic idea of our method is that we compute a sequence \( a_{t_0}, a_{t_1}, \ldots, a_{t_n} \), of more and more “abstract” smart tables, such that \( ct = a_{t_0} \) and \( at_n = at \). Each table \( at \) consists of smart tuples of the form \( \tau \sigma \) where \( \tau \sigma \) is a concrete (ordinary) tuple and \( \tau \sigma \) is an abstract (smart) tuple. We have \( \text{vars}(\tau) = \langle x_1, \ldots, x_{n-i} \rangle \)

\(^1\)In this paper, we only consider positive tables, i.e., tables containing allowed tuples.
while $\sigma$ may involve any variable $x_1, \ldots, x_n$. Such a smart tuple abstracts (covers) the set of all concrete tuples $\tau \sigma'$ that satisfy $\tau \sigma$. Thus, we build the smart table “from right to left” by progressively lengthening the abstract suffixes $\sigma$. At each iteration, we abstract the rightmost “concrete” column of the table.

To compute $a_{i+1}$ from $a_i$, we first compute a new smart table $nat_i$, the tuples of which are all of the form $\tau' s_{n-i}$, where $s_{n-i}$ has the form defined above. These tuples must be such that for every concrete tuple of the form $\tau' a \tau$ that satisfies $\tau' s_{n-i}$, there exists a tuple $\tau' a \sigma$ belonging to $a_i$. Then, we obtain $a_{i+1}$ by choosing a specific subset of $a_i \cup nat_i$, which covers $ct$.

The above method can be shown correct by induction on $i$. Its accuracy depends on the way we choose the new smart tuples in $nat_i$, and the specific subset of $a_i \cup nat_i$. The way we make these choices is explained in the next two subsections.

### Computing the new abstract tuples

To find out new smart tuples involving unary column constraints $s_{n-i}$, we consider, for any given prefix $\tau'$ and any given suffix $\sigma$, the set $S$ of all values $a$ such that $\tau' a \sigma$ belongs to $a_i$. We are allowed to add, to $nat_i$, any (and all) smart tuples $\tau' s_{n-i}$ such that $s_{n-i}$ determines a subset of $S$. For binary column constraints $s_{n-i}$, we consider, for any concrete tuples $\tau' a \tau'$, of length $l-1$, and $\tau'' a\tau''$, of length $n-i-j-1$, and for any value $a$, the set $S$ of all values $b$ such that the tuple $\tau'' a\tau'' b \sigma$ belongs to $a_i$. We are allowed to add, to $nat_i$, any (and all) smart tuples $\tau'' a\tau'' s_{n-i}$ such that $s_{n-i}$ is of the form $x_{n-i} <\text{op}> x_j$ and the set $\{ b \in \text{dom}(x_{n-i}) \mid b <\text{op}> a \}$ is a subset of $S$.

Except for very small concrete tables $ct$, it is too costly to add all allowed new smart tuples to $nat_i$. For smart tuples where $s_{n-i}$ is unary, we add a minimal set of smart tuples, covering the corresponding concrete tuples. For smart tuples where $s_{n-i}$ is binary, however, we a priori add all allowed new smart tuples. The justification for that is twofold: On the one hand, the number of allowed column constraints is normally less for binary ones, and, on the other hand, it is difficult to foresee at this stage which binary column constraint is the best choice. Good choices are estimated more accurately when the smart table $a_{n-j}$ is computed. Typically, when a column constraint $\sigma$ is chosen, at stage $n-j$, corresponding to a column constraint $x_{n-i} <\text{op}> x_j$, at stage $i$.

### Computing the coverings

Since the table $nat_i$ usually is bigger, and sometimes much bigger, than $a_i$, we have to choose the most “promising” smart tuples from $a_i \cup nat_i$ to build $a_{i+1}$. We use the following heuristic. We currently have smart tuples of the form $\tau \sigma$ where the $\tau$ are concrete (ordinary) and the $\sigma$ are abstract (smart). We may foresee that minimizing the number of different suffixes $\sigma$ in $a_{i+1}$ will lead to a final smart table with a small number of smart tuples: At least, we know that the corresponding suffixes of these final smart tuples are part of the selected ones. To determine this minimal (or, at least, small) number of suffixes, we compute, for every suffix $\sigma$ in

$$ct \cup nat_i \text{ the number of concrete tuples in } ct \text{ that satisfies } \sigma.$$ Let us denote this number by $\text{card}(\sigma)$. The computation of $\text{card}(\sigma)$ is not costly since we can incrementally maintain the number of concrete tuples $\tau \sigma'$ covered by each abstract tuple $\tau \sigma$. We enumerate the sequence of smart tuples $\tau \sigma$ in $a_i \cup nat_i$, in decreasing order of $\text{card}(\sigma)$, until a covering of $ct$ is obtained, and we define $a_{i+1}$ as the set of all those smart tuples. For every selected suffix $\sigma$, all corresponding tuples $\tau \sigma$ are put into $a_{i+1}$ to allow a correct incremental computation of $\text{card}(\sigma)$ at the next iterations.

This method to compute a covering is not intended to compute a minimal (or close to minimal) one but it aims at leaving the door open to future good choices. The point is not to determine a minimal covering within $a_i$, but only at the end, within $at$. Since $at = a_{m+1}$, we proceed differently at the last step (note that the prefixes $\tau$ are empty, then): Each time an abstract (final) tuple is selected to be put in $at$, we compute the number of concrete tuples covered by the remaining tuples in $at_{n-1} \cup nat_{n-1}$, which are not covered by the abstract tuples already in $at_n$. Then we continue to select tuples according to the updated numbers. This method is likely to produce a smaller covering than the previous one.

**An example**

As a simple example, the tables $ct = (a_{00})$, $a_1$, $a_2$, $a_3 = (at)$ computed by our algorithm for the example of Section 1 are shown below. The tables $a_0 \cup nat_0$, $a_1 \cup nat_1$, and $a_2 \cup nat_2$, which are not shown, respectively contain 28, 18, and 4 tuples. In all of them, there is a single suffix $\sigma$ for which $\text{card}(\sigma)$ is maximal (equal to 6). The table $at$ is different from but equivalent to the smart table proposed in Section 1.

<table>
<thead>
<tr>
<th>$x_1$</th>
<th>$x_2$</th>
<th>$x_3$</th>
<th>$at_1$</th>
<th>$at_2$</th>
<th>$at$</th>
</tr>
</thead>
<tbody>
<tr>
<td>$=1$</td>
<td>$=2$</td>
<td>$=1$</td>
<td>$=1$</td>
<td>$=2$</td>
<td>$=1$</td>
</tr>
<tr>
<td>$=1$</td>
<td>$=3$</td>
<td>$=1$</td>
<td>$=1$</td>
<td>$=3$</td>
<td>$=1$</td>
</tr>
<tr>
<td>$=2$</td>
<td>$=2$</td>
<td>$=2$</td>
<td>$=2$</td>
<td>$=2$</td>
<td>$=2$</td>
</tr>
<tr>
<td>$=2$</td>
<td>$=3$</td>
<td>$=2$</td>
<td>$=2$</td>
<td>$=3$</td>
<td>$=2$</td>
</tr>
<tr>
<td>$=3$</td>
<td>$=3$</td>
<td>$=3$</td>
<td>$=3$</td>
<td>$=3$</td>
<td>$=3$</td>
</tr>
</tbody>
</table>

| $a_{28}$ | $\geq 2$ | $\geq 2$ | $\geq 2$ | $\geq 2$ | $\geq 2$ |
| $a_{18}$ | $\geq 2$ | $\geq 2$ | $\geq 2$ | $\geq 2$ | $\geq 2$ |
| $a_4$    | $\geq 2$ | $\geq 2$ | $\geq 2$ | $\geq 2$ | $\geq 2$ |

| $= x_1$ | $= x_1$ | $= x_1$ |

### 3.2 Implementation Issues

Now we explain how the method of Section 3.1 can be implemented efficiently, i.e., how we actually implemented it.

We encode column constraints $s_i$ involving $x_i$ as integer values, leaving $i$ implicit, and we use the natural ordering on integers as a total ordering on column constraints (with the same $i$). The encoding is also such that a column constraint $x_i = a$ is represented by the integer $a$. So a concrete tuple is a particular case of a smart tuple. Smart tables are represented by two-dimensional arrays.

To compute new smart tuples to be put in $nat_i$, we have to determine sets of tuples of two forms: sets of tuples of the form $\tau' a \sigma$, where $\tau'$ and $\sigma$ are fixed, and sets of tuples of the form $\tau'' a \tau'' b \sigma$ where $\tau'' a, \tau''$, and $\sigma$ are fixed (see Section 3.1). To determine these sets we sort the array representing the smart table $a_i$ (let us simply identify them) according to different lexicographical orderings. As an invariant, we impose that, at the start of every iteration of the algorithm, the
table at_i is lexicographically sorted on the columns x_n, \ldots, x_{n-i+1}, x_1, \ldots, x_{n-i}. This implies that all smart tuples with the same suffix \( \sigma \) are consecutive, as well as all smart tuples of the form \( \tau \sigma \alpha \sigma \), where \( \tau \) and \( \sigma \) are fixed. Thus, we can determine all new smart tuples involving unary column constraints \( s_{n-i} \) through a single traversal of the array. Moreover, we can progressively modify the ordering of the array to successively compute the smart tuples involving binary column constraints \( s_{n-i} \): For instance, to compute the smart tuples of the form \( \tau'' \sigma \tau'' s_{n-i} \) where \( s_{n-i} \) is of the form \( x_{n-i} < \text{op} > x_j \), we have to lexicographically sort the array on the columns \( x_n, \ldots, x_{n-i+1}, x_1, \ldots, x_j, x_{n-j}, x_{n-j-1}, \ldots, x_{n-i-1}, x_{n-i} \). To compute the next smart tuples (those in which \( s_{n-i} \) is of the form \( x_{n-i} < \text{op} > x_{j-1} \)), we only have to re-order the array locally on the tuples that share the same suffix \( \sigma \) and, if it makes sense, same prefixes of length \( j - 2 \). It is also possible to fill in the table \( nat_i \) in such a way that it is lexicographically sorted on the columns \( x_n, \ldots, x_{n-i}, x_1, \ldots, x_{n-j-1} \) and to progressively reorder \( at_i \) in the same way. Therefore, the smart table \( at_i \cup nat_i \), can be computed by simply merging \( at_i \) and \( nat_i \) in linear time on the size of \( at_i \cup nat_i \).

To compute \( at_{i+1} \), i.e. the covering of \( at_i \cup nat_i \), we first build a list of descriptors of the sub-arrays of \( at_i \cup nat_i \), sharing the same abstract suffix \( \sigma' \) (of length \( i + 1 \)). This list is sorted on \( \text{card}(\sigma') \) in decreasing order. Then we go through the list to select sub-arrays until \( ct \) is completely covered: for each selected smart tuple, we generate the set of concrete tuples represented by the smart tuple, and we “mark” them. This can be done in linear time on the size of the set of covered concrete tuples. The algorithm to compute the final (“minimal”) covering is more sophisticated: We first build, once for all, for each concrete tuple in \( ct \), a list of all abstract tuples that cover it (in \( at_i \cup nat_i \)). When a concrete tuple is marked “covered”, we traverse this list and, for every abstract tuple of the list, we decrease, by one, a counter giving the number of unmarked concrete tuples covered by this abstract tuple.

This method to dynamically recompute the number of concrete tuples covered by all still unselected abstract tuples seems to us as efficient as possible. In fact, it is strongly related to the Galois connections celebrated in abstract interpretation [Cousot and Cousot, 1977].

3.3 Complexity Analysis

Our method is more efficient when it is accurate: If the table \( at \) is comparatively small with respect to \( ct \), it is synthesized quicker. Thus, the complexity analysis we provide hereunder is rather pessimistic. We complement it later, with experimental efficiency results.

Remember that we call an iteration the whole sequence of processes that are executed to compute \( at_{i+1} \) from \( at_i \). We first analyze the time complexity of such an iteration for \( i \) given. Let us call \( m_i \) the number of tuples in \( at_i \). For any given abstract suffix \( \sigma \) (of length \( i \)), in \( at_i \), we denote the number of tuples sharing this suffix, by \( m_{\sigma \sigma} \). To build \( nat_i \), those tuples are sorted \( n - i \) times. The sorting that is preliminary to find out constraints of the form \( x_{n-i} < \text{op} > x_j \), is done in \( O((n - i - j)^2 d m_i) \). (We use a kind of radix sort.) Summing on all suffixes and all values of \( j \), we get a time complexity equal to \( O((n - i)^2 d m_i) \) for all sorting operations. After each sorting operation, \( m_{\sigma \sigma} \) tuples are sequentially processed to add new smart tuples in \( nat_i \). This is done \( (n - i) \) times in \( O((n - i)^2 m_{\sigma \sigma}) \). Globally, in \( O((n - i)^2 m_{\sigma \sigma}) \). The number of new tuples added to \( nat_i \) is \( O((n - i)^2 m_{\sigma \sigma}) \). They are then sorted inside \( nat_i \) in \( O((n - i)^2 d m_i) \). Summing on all suffixes \( m_{\sigma \sigma} \), we get that \( nat_i \) is built in \( O((n - i)^2 d m_i) \). Afterwards, the time needed to compute \( at_i \) and \( nat_i \) by merging \( at_i \) and \( nat_i \), is \( O((n - i) m_i) \). Putting together all previous results, we obtain that \( at_{i+1} \cup nat_i \) is computed from \( at_i \) in \( O((n - i)^2 d m_i) \). Thus, the complexity formula \( O(m n^2 d n + m) \). In many situations, \( m \) dominates \( d n \), since it is \( O(d n^2) \), in general. We thus obtain \( O(m n^2) \), as the final worst case complexity of our method. Remember that \( m \) is the size of \( ct \). So, we estimate that our method, as it is implemented, is quadratic in the size of the input table (time complexity). With the same hypothesis on \( m \), the space complexity is \( O(m n^2) \) for all phases of the algorithm except for the computation of the last covering, which is \( O(m m_{n-1}) \).
when possible, in the smart tuples of the final table. For instance, when the smart tuple contains a column constraint of the form \( x_j = a \). A sometimes interesting simplification consists of collecting only unary column constraints, or even only \( * \). So the final table only contains compressed or short tuples and specialized filtering algorithms can be applied to it.

### 4 Experimental Results

To show the practical interest of the algorithm described in this paper, we have conducted an experimentation using some well-known global constraints: \( \text{lex, element, maximum, atMost1, notAllEqual, and distinctVectors} \). We choose a few global constraints, because their natural smart forms are already known. Consequently, we can fairly study the compression efficiency of the proposed algorithm, which is our main objective. Indeed, we can simply compare the size of the initial smart tables with the size of the generated smart tables. The protocol is the following: for a given global constraint \( \text{glb, arity } n \) and maximum domain size \( d \), we generate a table (constraint) containing all tuples accepted by the global constraint \( \text{glb-} n- d \). For example, \( \text{lex-10-4} \) is the constraint \( \text{lex} \) whose scope contains 10 variables with 4 values per domain; the corresponding table constraint contains 524,800 tuples. We have also generated random smart table constraints \( \text{random-} n- d \) with 6 smart tuples for each constraint: each smart tuple is composed of column constraints randomly chosen with the same probability \( 1/13 \) for "\( * \)" and every column constraint of the form \( x_i < \text{op} > x_j \) or \( x_i < \text{op} > x_j \). Once a random smart table is generated, we build the corresponding ordinary table.

Our experimentation includes two phases. In a first phase, we study the compression capability of our algorithm and its practical efficiency. In a second phase, we compare the performance of using automatically synthesized smart constraints with respect to state-of-the-art algorithms on ordinary tables, short tables and MDDs.

Table 1 shows how our synthesis algorithm behaves on the benchmarks. We apply the algorithm to ten randomly chosen permutations of the columns of each table constraint. Columns \( m, mi-ma, m_{\text{av}} \), respectively indicate the number of tuples of the ordinary table, the minimum and maximum number of tuples of the resulting smart table and its average value. Column \( \text{cmp} \) is the ratio \((m - mi)/m\). The algorithm provides optimal results on the tables notAllEqual, distinctVectors, and element, which contain few constraints \( x_i < \text{op} > x_j \), not sharing the same variable, which allows the algorithm to synthesize an optimal table for any permutation. For \( \text{lex} \), the results are also optimal although some tuples involve many constraints \( x_i < \text{op} > x_j \). So, for some bad permutations, the execution time may become ten times greater than for the good ones. Constraints atMost1 and maximum, the results are not always optimal because optimal smart tables contain many constraints \( x_i < \text{op} > x_j \); thus, an optimal table such that \( j < i \) for all constraints does not exist for all permutations. Surprisingly, for some examples and permutations, our algorithm provides solutions shorter than the original ones, i.e., on most instances of atMost1. For the random constraints, the results are less good because the original smart tables contains too few "\( * \)" constraints. Nevertheless, on two constraints \( \text{random-13-4} \), the number of tuples is less than for the original tables. Columns \( \text{tot}, \% \text{nat}, \% \text{cc} \) give the average execution time, in seconds, and the proportion of time spent to abstract tuples (\( \% \text{nat} \)) and to compute coverings (\( \% \text{cc} \)). We see that the timings devoted to both tasks are similar on most examples. Column \( \text{cp1} \) is the ratio \( m_{\text{av}}/m_\text{cp1} \), where \( m_{\text{cp1}} \) is the average value of \( m_\text{av} \), the number of tuples of the table \( at\_i \).
Table 2: Filtering algorithms on constraints introduced earlier. For a fair comparison, we proceeded as follows: for each algorithm, we iteratively run its execution and randomly removed 10% of the values (until a failure occurs). This way, many different call contexts were simulated. This inner process was repeated 1,000 times, and we additionally took the average time over 10 executions. Using the same seed, the different filtering algorithms are all tested on the same search trees. In Table 2, columns \( m, m^* \) and \( m^* \) respectively indicate the sizes of the ordinary, smart and short tables (these last two being synthesized by our algorithm). The other columns give the average times (in milliseconds) obtained by MDD4R [Perez and Régin, 2014] on multi-valued decision diagrams (built initially before being exploited), state-of-the-art algorithm Compact Table on ordinary (CT) [Demeulenaere et al., 2016] and short (CT*) [Verhaeghe et al., 2017] tables, and finally Smart Table (filtering algorithm depicted in [Mairy et al., 2015]) on smart (ST) and short (ST*) tables. It is important to note that we only report filtering execution time here, so the time required to build MDDs and short tables is not taken into account. First, we can observe that the compression capability of our algorithm, when parameterized to output short tables, is sometimes very good. This is the case for constraints notAllEqual, distinctVectors, and element. Interestingly, even when the compression ratio is not too impressive (from 10% to 90%), it appears that CT* outperforms CT, sometimes quite largely. This is the case for constraints maximum and random. However, we believe that the main result of our experimentation is that ST applied to the synthesized smart tables is particularly robust. When ST is the fastest algorithm, on large tables it can outperform state-of-the-art MDD and CT algorithms by one or two orders of magnitude; see for example, lex-8-6 or distinctVectors-10-4. When ST is not the fastest algorithm, it remains very close to the best competitor.

$$\sum \frac{m}{m^*}$$ supports our “postulate” that \( m_1 = O(m) \). Finally, column \( cp_{op} \) is the ratio \( tot/(nm) \), i.e., the execution time divided by the size of the concrete table, in microseconds. It shows that, for these benchmarks, the time complexity of our algorithm is much better than predicted by our worst case study: It is almost linear.

Table 2 shows the relative performances of various filtering algorithms on the constraints introduced earlier. For a fair comparison, we proceeded as follows: for each algorithm, we iteratively run its execution and randomly removed 10% of the values (until a failure occurs). This way, many different call contexts were simulated. This inner process was repeated 1,000 times, and we additionally took the average time over 10 executions. Using the same seed, the different filtering algorithms are all tested on the same search trees. In Table 2, columns \( m, m^* \) and \( m^* \) respectively indicate the sizes of the ordinary, smart and short tables (these last two being synthesized by our algorithm). The other columns give the average times (in milliseconds) obtained by MDD4R [Perez and Régin, 2014] on multi-valued decision diagrams (built initially before being exploited), state-of-the-art algorithm Compact Table on ordinary (CT) [Demeulenaere et al., 2016] and short (CT*) [Verhaeghe et al., 2017] tables, and finally Smart Table (filtering algorithm depicted in [Mairy et al., 2015]) on smart (ST) and short (ST*) tables. It is important to note that we only report filtering execution time here, so the time required to build MDDs and short tables is not taken into account. First, we can observe that the compression capability of our algorithm, when parameterized to output short tables, is sometimes very good. This is the case for constraints notAllEqual, distinctVectors, and element. Interestingly, even when the compression ratio is not too impressive (from 10% to 90%), it appears that CT* outperforms CT, sometimes quite largely. This is the case for constraints maximum and random. However, we believe that the main result of our experimentation is that ST applied to the synthesized smart tables is particularly robust. When ST is the fastest algorithm, on large tables it can outperform state-of-the-art MDD and CT algorithms by one or two orders of magnitude; see for example, lex-8-6 or distinctVectors-10-4. When ST is not the fastest algorithm, it remains very close to the best competitor.

5 Conclusion

We presented an algorithm that synthesizes smart tables from table constraints. We demonstrated its accuracy and efficiency when applied to table constraints that can be compactly represented in this form. Benchmarks on global constraints show that our algorithm is able to find the best compression in many situations, and come close to the best results otherwise. Our algorithm is also able to limit its output to short tuples, allowing state-of-the-art filtering algorithm such as CT* to be used [Verhaeghe et al., 2017]. We also discussed its main limitation: the building of smart tables from right to left may prevent it to find an optimal solution when many column constraints of the form \( x_i <op> x_j \) are involved in such a solution. As a future work, we plan to alleviate this limitation by simultaneously exploring several permutations of the table columns. Note that not all table constraints can be compactly represented as smart tables. We will use our algorithm to analyze the whole range of existing table constraints to find out situations where smart tables outperform or are competitive with other classes of constraint representations.

Acknowledgments

The second author is supported by the FRIA-FNRS (Fonds pour la Formation à la Recherche dans l’Industrie et dans l’Agriculture, Belgium).
References


