Automatic Synthesis of Smart Table Constraints by Abstraction of Table Constraints

Baudouin Le Charlier[1], **Minh Thanh Khong**[1], **Christophe Lecoutre**[2], **Yves Deville**[1]
[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 results 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 [Demeulenaere *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 Régin, 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 Régin, 2014]. Other compression-based approaches keep the structure of tables, but replace ordinary tuples by compressed tuples [Katsirelos and Walsh, 2007; Régin, 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 $\langle x, y, z\rangle$ that can take their values in $\{1,2,3\}$ can be represented by a smart table containing only one smart tuple:

x	y	z
$= z$	≥ 2	$*$

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 $dom(x)$, 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 $scp(c)$, and is semantically defined by a *relation*, denoted by $rel(c)$, that contains the set of tuples allowed for the variables involved in c. The *arity* of a constraint c is $|scp(c)|$.

Let $\tau = (a_1, \ldots, a_n)$ be a tuple of values associated with a sequence of variables $vars(\tau) = \langle x_1, \ldots, x_n \rangle$. The ith value of τ is denoted by $\tau[i]$ or $\tau[x_i]$, and we say that τ is *valid* iff $\forall i \in 1..n, \tau[i] \in dom(x_i)$. The tuple τ is a *support* on a constraint c such that $vars(\tau) = scp(c)$ iff τ is a valid tuple allowed by c; we also say that τ *satisfies* c. If τ is a support on a constraint c involving a variable x such that $\tau[x] = a$, we say that τ 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 $\forall x \in scp(c), \forall a \in dom(x)$, 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 $sols(N)$.

A *table constraint* c is a constraint such that $rel(c)$ 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 $table(sc)$, which contains a set of smart tuples. If $scp(sc)$ is $\langle x_1, \ldots, x_n \rangle$, then a *smart tuple* σ in $table(sc)$ is a sequence $\langle s_1, \ldots, s_n \rangle$ of column constraints, where a *column constraint* s_i can be either a *unary column constraint* of one of the two forms $x_i = *$ and x_i <op> a, or a *binary column constraint* of the form x_i <op> x_j, with a being a constant and <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 <op> $x_j + a$ 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 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. 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 <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 at_0, at_1, $\ldots$, at_n, of more and more "abstract" smart tables, such that $ct = at_0$ and $at_n = at$. Each table at_i consists of smart tuples of the form $\tau\sigma$ where τ is a concrete (ordinary) tuple and σ is an abstract (smart) tuple. We have $vars(\tau) = \langle x_1, \ldots, x_{n-i} \rangle$

[1] In this paper, we only consider *positive* tables, i.e., tables containing allowed tuples.

while σ may involve any variable $x_1, \ldots x_n$. Such a smart tuple abstracts (covers) the set of all concrete tuples $\tau\tau'$ that satisfy $\tau\sigma$. Thus, we build the smart table "from right to left" by progressively lengthening the abstract suffixes σ: At each iteration, we abstract the rightmost "concrete" column of the table.

To compute at_{i+1} from at_i, we first compute a new smart table nat_i, the tuples of which are all of the form $\tau' s_{n-i}\,\sigma$,[2] 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 $\tau' s_{n-i}\,\sigma$, there exists a tuple $\tau' a\,\sigma$ belonging to at_i. Then, we obtain at_{i+1} by choosing a specific subset of $at_i \cup nat_i$, 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 $at_i \cup nat_i$. 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 τ' and any given suffix σ, the set S of all values a such that $\tau' a\,\sigma$ belongs to at_i. We are allowed to add, to nat_i, any (and all) smart tuple(s) $\tau' s_{n-i}\,\sigma$ such that s_{n-i} determines a subset of S. For *binary* column constraints s_{n-i}, we consider, for any concrete tuples τ'', of length $j-1$, and τ''', of length $n-i-j-1^2$, and for any value a, the set S of all values b such that the tuple $\tau'' a\,\tau''' b\,\sigma$ belongs to at_i. We are allowed to add, to nat_i, any (and all) smart tuple(s) $\tau'' a\,\tau''' s_{n-i}\,\sigma$ such that s_{n-i} is of the form x_{n-i} <op> x_j and the set $\{b \in dom(x_{n-i}) \mid b$ <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, one 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 at_{n-j} is computed. Typically, when a column constraint $*$ is chosen, at stage $n-j$, corresponding to a column constraint x_{n-i} <op> x_j, at stage i.

Computing the coverings

Since the table nat_i usually is bigger, and sometimes much bigger, than at_i, we have to choose the most "promising" smart tuples from $at_i \cup nat_i$ to build at_{i+1}. We use the following heuristic. We currently have smart tuples of the form $\tau\sigma$ where the τ are concrete (ordinary) and the σ are abstract (smart). We may foresee that minimizing the number of different suffixes σ in at_{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 σ in $at_i \cup nat_i$ the number of concrete tuples in *ct* that satisfies σ. Let us denote this number by $card(\sigma)$. The computation of $card(\sigma)$ is not costly since we can incrementally maintain the number of concrete tuples $\tau\tau'$ covered by each abstract tuple $\tau\sigma$. We enumerate the sequence of smart tuples $\tau\sigma$ in $at_i \cup nat_i$, in decreasing order of $card(\sigma)$, until a covering of *ct* is obtained, and we define at_{i+1} as the set of all those smart tuples. For every selected suffix σ, all corresponding tuples $\tau\sigma$ are put into at_{i+1} to allow a correct incremental computation of $card(\sigma)$ 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 at_i, but only at the end, within *at*. Since $at = at_n$, we proceed differently at the last step (note that the prefixes τ are empty, then): Each time an abstract (final) tuple is selected to be put in *at*, we recompute 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* ($= at_0$), at_1, at_2, at_3 ($= at$) computed by our algorithm for the example of Section 1 are shown below. The tables $at_0 \cup nat_0$, $at_1 \cup nat_1$, and $at_2 \cup nat_2$, which are not shown, respectively contain 28, 18, and 4 tuples. In all of them, there is a single suffix σ for which $card(\sigma)$ is maximal (equal to 6). The table *at* is different from but equivalent to the smart table proposed in Section 1.

	x_1	x_2	x_3		x_1	x_2	x_3
ct	= 1	= 2	= 1	at_1	= 1	= 2	$= x_1$
	= 1	= 3	= 1		= 1	= 3	$= x_1$
	= 2	= 2	= 2		= 2	= 2	$= x_1$
	= 2	= 3	= 2		= 2	= 3	$= x_1$
	= 3	= 2	= 3		= 3	= 2	$= x_1$
	= 3	= 3	= 3		= 3	= 3	$= x_1$
at_2	= 1	≥ 2	$= x_1$	*at*	$*$	≥ 2	$= x_1$
	= 2	≥ 2	$= x_1$				
	= 3	≥ 2	$= x_1$				

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 τ' and σ are fixed, and sets of tuples of the form $\tau'' a\,\tau''' b\,\sigma$ where τ'', a, τ''', and σ are fixed (see Section 3.1). To determine these sets we sort the array representing the smart table at_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

[2] We have $vars(\tau')$, $vars(\tau'')$, and $vars(\tau''')$ being respectively $\langle x_1, \ldots, x_{n-i-1}\rangle$, $\langle x_1, \ldots, x_{j-1}\rangle$, and $\langle x_{j+1}, \ldots, x_{n-i-1}\rangle$.

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 σ are consecutive, as well as all smart tuples of the form $\tau' a\, \sigma$, where τ' and σ 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 $\tau'' a\, \tau''' s_{n-i}\, \sigma$ where s_{n-i} is of the form x_{n-i} <op> x_j, we have to lexicographically sort the array on the columns $x_n, \ldots, x_{n-i+1}, x_1, \ldots, x_{j-1}, x_{j+1}, \ldots, x_{n-i-1}, x_j, x_{n-i}$. To compute the next smart tuples (those in which s_{n-i} is of the form x_{n-i} <op> x_{j-1}), we only have to reorder the array locally on the tuples that share the same suffix σ 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 $x_n, \ldots, x_{n-i}, x_1, \ldots, x_{n-i-1}$ 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 σ' (of length $i+1$). This list is sorted on $card(\sigma')$ 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 σ (of length i), in at_i, we denote the number of tuples sharing this suffix, by $m_{i\sigma}$. 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} <op> x_j is done in $O((n-i-j)\, d\, m_{i\sigma})$. (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, $m_{i\sigma}$ tuples are sequentially processed to add new smart tuples in nat_i. This is done $(n-i)$ times in $O((n-i)\, m_{i\sigma})$. Globally, in $O((n-i)^2 m_{i\sigma})$. The number of new tuples added to nat_i is $O((n-i)\, m_{i\sigma})$. They are then sorted inside nat_i in $O((n-i)^2\, d\, m_{i\sigma})$. Summing on all suffixes $m_{i\sigma}$, we get that nat_i is built in $O((n-i)^2 d\, m_i)$. Afterwards, the time needed to compute $at_i \cup nat_i$ by merging at_i and nat_i is $O(i\,(n-i)\, m_i)$. Putting together all previous results, we obtain that $at_i \cup nat_i$ is computed from at_i in $O((n-i)^2 d\, m_i + (n-i)^2 d\, m_i + i\,(n-i)\, m_i)$, which is equal to $O((i + (n-i)d)(n-i)m_i)$.

We now turn to the computation of the covering by which at_{i+1} is computed from $at_i \cup nat_i$. At most m sub-arrays of $at_i \cup nat_i$ are selected to be put in at_{i+1} and the computation of the subset of *ct* they each cover is done in $O(m\, n)$. Therefore, the overall computation of the covering is done in $O(m^2\, n)$. The same figures are obtained for the covering method used at the last iteration.

Finally, we estimate an upper-bound to compute *at* from *ct*. The main difficulty is due to m_i, which is a number difficult to predict. Experimentally, we observe that it increases for small values of i and decreases afterwards. We thus postulate shamelessly that m_i is $O(m)$. Then, by summing the previous results on all values of i, we get the complexity formula $O(m\, n^2 (d\, n + m))$. In many situations, m dominates $d\, n$, since it is $O(d^n)$, in general. We thus obtain $O(m^2 n^2)$, as the final worst case complexity of our method. Remember that $m\, n$ 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_i, the space complexity is $O(m\, n^2)$ for all phases of the algorithm except for the computation of the last covering, which is $O(m\, m_{n-1})$. [3]

3.4 Additional Remarks

The choice we make to abstract the table *ct* from right to left is arbitrary. Nevertheless, it allows us to consider all pairs of variables x_i, x_j as candidates for column constraints of the form x_i <op> x_j. The restrictive condition $j < i$ ensures that the smart tuples are acyclic but it prevents us to put two column constraints that share the same variable x_i (on the left hand side) in the same smart tuple. Using another permutation of the table columns may, in many cases, solve the problem if we swap the variables in one or both of the two constraints. But, of course, trying all permutations of the table columns is computationally unsatisfactory. Even more sadly, there are smart tables for which no permutation of the variables may ensure that $j < i$ for every column constraints x_i <op> x_j, in the table. In spite of these pessimistic observations, our method often gives good results in practice, as shown in Section 4.

The reference algorithm presented in Subsections 3.1 and 3.2 can be improved and/or simplified in several ways. We only cite two. A simple improvement consists of replacing column constraints of the form x_i <op> x_j by equivalent column constraints involving a single variable,

[3] If m_{n-1} is big, the final smart table probably is uninteresting. The algorithm may stop without computing it.

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: `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 `glb`, arity n and maximum domain size d, we generate a table (constraint) containing all tuples accepted by the global constraint `glb-n-d`. For example, `lex-10-4` is the constraint `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 `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 <op> a or x_i <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_{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 *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 <op> x_j, not sharing the same variable, which allows the algorithm to synthesize an optimal table for any permutation. For `lex`, the results are also optimal although some tuples involve many constraints x_i <op> x_j. So, for some bad permutations, the execution time may become ten times greater than for the good ones. For constraints `atMost1` and `maximum`, the results are not always optimal because optimal smart tables contain many constraints x_i <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 `random-13-4`, the number of tuples is less than for the original tables. Columns *tot*, *%nat*, *%cc* give the average execution time, in seconds, and the proportion of time spent to abstract tuples (*%nat*) and to compute coverings (*%cc*). We see that the timings devoted to both tasks are similar on most examples. Column cp_1 is the ratio $\overline{m_i}/m$, where $\overline{m_i}$ is the average value of m_i, the number of tuples of the table at_i. It

n-d	Size				Cpu			Cplty	
	m	*mi-ma*	m_{av}	*cmp*	*tot*	*%nat*	*%cc*	cp_1	cp_2
Constraints `notAllEqual-n-d`									
5-5	3.1K	4-4	4.0	99.87	0.04	43.12	56.39	0.31	2
9-3	19K	8-8	8.0	99.95	1.03	44.12	54.45	0.24	5
6-6	46K	5-5	5.0	99.98	0.89	27.76	71.38	0.23	3
7-5	78K	6-6	6.0	99.99	2.32	30.77	68.34	0.22	4
9-4	262K	8-8	8.0	99.99	14.71	28.64	70.45	0.19	6
8-5	390K	7-7	7.0	99.99	19.78	37.98	61.40	0.19	6
Constraints `lex-n-d`									
6-4	2.0K	3-3	3.0	99.85	0.07	51.86	47.51	0.47	5
6-5	7.8K	3-3	3.0	99.96	0.22	44.94	54.47	0.45	4
8-4	32K	4-4	4.0	99.98	3.15	34.02	65.22	0.62	11
8-5	195K	4-4	4.0	99.99	39.85	13.95	85.76	0.56	25
10-4	524K	5-5	5.0	99.99	62.05	30.16	69.16	0.27	11
8-6	840K	4-4	4.0	99.99	128.61	19.94	78.76	0.39	19
Constraints `distinctVectors-n-d`									
8-3	6.4K	4-4	4.0	99.93	0.23	68.93	29.44	0.27	4
8-4	65K	4-4	4.0	99.99	1.66	51.79	46.63	0.21	3
12-3	530K	6-6	6.0	99.99	41.22	58.09	40.29	0.18	6
10-4	1.04M	5-5	5.0	99.99	57.23	56.51	42.24	0.17	5
Constraints `element-n-d`									
9-7a	15K	7-8	7.1	99.95	0.85	61.43	37.16	0.47	6
7-6	38K	5-5	5.0	99.98	0.78	52.60	46.33	0.37	2
7-7	84K	5-5	5.0	99.99	1.66	45.02	54.07	0.33	2
7-8	163K	5-5	5.0	99.99	3.48	40.91	58.28	0.30	3
9-7b	546K	7-7	7.0	99.99	26.04	42.14	56.94	0.31	5
8-7	705K	6-6	6.0	99.99	23.19	41.32	57.99	0.27	4
Constraints `atMost1-n-d`									
7-4	8.7K	4-24	10.7	99.95	0.21	48.54	50.70	0.27	3
8-4	29K	4-48	20.1	99.98	0.95	40.40	58.62	0.26	4
12-3	39K	16-61	33.5	99.95	10.81	48.79	50.31	0.89	22
9-4	96K	6-62	29.7	99.99	7.94	35.10	64.15	0.41	9
14-3	184K	20-77	59.1	99.98	101.47	38.13	61.08	0.89	39
10-4	314K	8-77	45.8	99.99	43.41	35.80	63.61	0.46	13
Constraints `maximum-n-d`									
6-5	3.1K	5-16	9.2	99.84	0.08	43.73	55.90	0.42	4
6-6	7.7K	5-16	7.3	99.93	0.14	40.89	58.72	0.32	3
10-3	19K	9-16	14.6	99.95	2.47	24.86	74.39	0.27	12
8-5	78K	7-45	25.5	99.99	7.74	22.90	76.59	0.46	12
10-4	262K	9-43	22.5	99.99	55.39	27.08	72.24	0.53	21
9-5	390K	8-68	35.8	99.99	64.98	20.35	79.20	0.45	18
Constraints `random-n-d`									
6-10a	86K	55-301	150.9	99.93	3.38	28.63	70.97	0.60	6
6-10b	158K	113-702	350.0	99.92	10.57	16.55	83.20	0.64	11
6-10c	535K	73-429	199.9	99.98	80.52	8.97	90.89	0.68	25
8-8a	61K	89-174	125.4	99.85	3.61	37.34	62.23	0.57	7
8-8b	124K	22-40	26.7	99.98	7.53	32.71	66.88	0.52	7
8-8c	446K	8-304	104.6	99.99	40.74	19.21	80.55	0.44	11
13-4a	60K	2-17	6.5	99.99	14.07	40.27	59.09	0.55	17
13-4b	106K	2-9	5.1	99.99	30.94	33.25	66.19	0.57	22
13-4c	322K	9-38	18.8	99.99	115.21	27.67	71.61	0.50	27

Table 1: Results of synthesis algorithm obtained on global constraints and randomly generated smart table constraints.

n-d	Table Size			Filtering Algorithm				
	m	m^s	m^*	MDD	CT	ST	CT*	ST*
Constraints `notAllEqual-n-d`								
5-5	3.1K	4	80	42	31	**20**	25	34
9-3	19K	8	48	35	51	**17**	22	41
6-6	46K	5	150	59	95	**20**	28	59
7-5	78K	6	120	52	108	**20**	27	58
9-4	262K	8	96	49	223	**17**	42	44
8-5	390K	7	140	57	330	**21**	30	72
Constraints `lex-n-d`								
6-4	2.0K	3	110	62	27	25	**23**	28
6-5	7.8K	3	260	101	47	28	**27**	59
8-4	32K	4	446	187	73	30	**28**	84
8-5	195K	4	1.3K	350	195	**30**	32	273
10-4	524K	5	1.7K	548	372	31	**28**	392
8-6	840K	4	3.2K	742	845	**33**	35	1.0K
Constraints `distinctVectors-n-d`								
8-3	6.4K	4	24	62	32	**15**	18	18
8-4	65K	4	48	169	94	**20**	23	21
12-3	530K	6	36	374	300	**18**	20	18
10-4	1.04M	5	60	533	833	**19**	23	24
Constraints `element-n-d`								
9-7a	15K	7	21	49	45	29	34	**25**
7-6	38K	5	30	110	107	37	**28**	28
7-7	84K	5	35	107	170	36	29	**28**
7-8	163K	5	40	98	220	44	**27**	29
9-7b	546K	7	35	126	483	49	**25**	27
8-7	705K	6	42	120	841	53	35	**33**
Constraints `atMost1-n-d`								
7-4	8.7K	4	5.8K	44	47	**35**	40	1.6K
8-4	29K	4	20K	52	66	**35**	68	6.6K
12-3	39K	17	33K	**43**	69	83	63	8.8K
9-4	96K	6	69K	**54**	108	**54**	108	23K
14-3	184K	21	159K	**45**	133	97	139	38K
10-4	314K	8	236K	**50**	244	61	229	80K
Constraints `maximum-n-d`								
6-5	3.1K	5	1.0K	46	31	66	**27**	202
6-6	7.7K	5	3.1K	62	52	83	**35**	990
10-3	19K	9	521	63	47	66	**21**	90
8-5	78K	7	16K	**55**	105	79	**55**	6.0K
10-4	262K	9	19K	**58**	202	84	59	6.3K
9-5	390K	8	65K	**60**	336	90	105	23K
Constraints `random-n-d`								
6-10a	86K	55	61K	325	177	177	**155**	33K
6-10b	158K	296	38K	308	239	303	**143**	19K
6-10c	535K	429	32K	311	700	440	**122**	17K
8-8a	61K	91	25K	169	128	127	**96**	10K
8-8b	124K	22	79K	223	199	**89**	182	34K
8-8c	446K	190	65K	213	497	177	**148**	27K
13-4a	60K	3	13K	54	78	**33**	36	2K
13-4b	106K	4	19K	133	106	53	**52**	4.9K
13-4c	322K	11	314K	82	233	**53**	277	99K

Table 2: Filtering on compressed tables from global constraints and randomly generated smart table constraints.

supports our "postulate" that $m_i = O(m)$. Finally, column cp_2 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^s 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, short and smart 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

[Bacchus and Walsh, 2005] Fahiem Bacchus and Toby Walsh. Propagating logical combinations of constraints. In *Proceedings of IJCAI'05*, pages 35–40, 2005.

[Beldiceanu *et al.*, 2014] Nicolas Beldiceanu, Mats Carlsson, and Jean-Xavier Rampon. Global constraint catalog. Technical Report T2012:03, TASC-SICS-LINA, 2014.

[Carlson and Carlsson, 1995] Björn Carlson and Mats Carlsson. Compiling and executing disjunctions of finite domain constraints. In *Proceedings of ICLP'95*, pages 117–131, 1995.

[Cheng and Yap, 2010] Kenil C. K. Cheng and Roland H. C. Yap. An MDD-based generalized arc consistency algorithm for positive and negative table constraints and some global constraints. *Constraints*, 15(2):265–304, 2010.

[Cousot and Cousot, 1977] Patrick Cousot and Radhia Cousot. Abstract interpretation: a unified lattice for static analysis of programs by construction of approximation of fixpoints. In *Proceedings of POPL'77*, pages 238–252, 1977.

[Demeulenaere *et al.*, 2016] Jordan Demeulenaere, Renaud Hartert, Christophe Lecoutre, Guillaume Perez, Laurent Perron, Jean-Charles Régin, and Pierre Schaus. Efficiently filtering table constraints with reversible sparse bit-sets. In *Proceedings of CP'16*, pages 207–223, 2016.

[Gent *et al.*, 2007] Ian P. Gent, Christopher Jefferson, Ian Miguel, and Peter Nightingale. Data structures for generalised arc consistency for extensional constraints. In *Proceedings of AAAI'07*, pages 191–197, 2007.

[Hentenryck *et al.*, 1998] Pascal Van Hentenryck, Vijay A. Saraswat, and Yves Deville. Design, implementation and evaluation of the constraint language cc(FD). *Journal of Logic Programming*, 37(1-3):139–164, 1998.

[Jefferson and Nightingale, 2013] Christopher Jefferson and Peter Nightingale. Extending simple tabular reduction with short supports. In *Proceedings of IJCAI'13*, pages 573–579, 2013.

[Jefferson *et al.*, 2010] Christopher Jefferson, Neil C. A. Moore, Peter Nightingale, and Karen E. Petrie. Implementing logical connectives in constraint programming. *Artificial Intelligence*, 174(16):1407–1429, 2010.

[Katsirelos and Walsh, 2007] George Katsirelos and Toby Walsh. A compression algorithm for large arity extensional constraints. In *Proceedings of CP'07*, pages 379–393, 2007.

[Lecoutre and Szymanek, 2006] Christophe Lecoutre and Radoslaw Szymanek. Generalized arc consistency for positive table constraints. In *Proceedings of CP'06*, pages 284–298, 2006.

[Lecoutre *et al.*, 2012] Christophe Lecoutre, Chavalit Likitvivatanavong, and Roland H. C. Yap. A path-optimal GAC algorithm for table constraints. In *Proceedings of ECAI'12*, pages 510–515, 2012.

[Lecoutre, 2011] Christophe Lecoutre. STR2: Optimized simple tabular reduction for table constraints. *Constraints*, 16(4):341–371, 2011.

[Lhomme and Régin, 2005] Olivier Lhomme and Jean-Charles Régin. A fast arc consistency algorithm for n-ary constraints. In *Proceedings of AAAI'05*, pages 405–410, 2005.

[Lhomme, 2004] Olivier Lhomme. Arc-consistency filtering algorithms for logical combinations of constraints. In *Proceedings of CPAIOR'04*, pages 209–224, 2004.

[Lhomme, 2012] Olivier Lhomme. Practical reformulations with table constraints. In *Proceedings of ECAI'12*, pages 911–912, 2012.

[Mairy *et al.*, 2012] Jean-Baptiste Mairy, Pascal Van Hentenryck, and Yves Deville. An optimal filtering algorithm for table constraints. In *Proceedings of CP'12*, pages 496–511, 2012.

[Mairy *et al.*, 2015] Jean-Baptiste Mairy, Yves Deville, and Christophe Lecoutre. The smart table constraint. In *Proceedings of CPAIOR'15*, pages 271–287, 2015.

[Montanari, 1974] Ugo Montanari. Network of constraints : Fundamental properties and applications to picture processing. *Information Science*, 7:95–132, 1974.

[Nightingale *et al.*, 2011] Peter Nightingale, Ian P. Gent, Christopher Jefferson, and Ian Miguel. Exploiting short supports for generalised arc consistency for arbitrary constraints. In *Proceedings of IJCAI'11*, pages 623–628, 2011.

[Perez and Régin, 2014] Guillaume Perez and Jean-Charles Régin. Improving GAC-4 for Table and MDD constraints. In *Proceedings of CP'14*, pages 606–621, 2014.

[Régin, 2011] Jean-Charles Régin. Improving the expressiveness of table constraints. In *Proceedings of the workshop ModRef'11 held with CP'11*, 2011.

[Srinivasan *et al.*, 1990] Arvind Srinivasan, Timothy Kam, Sharad Malik, and Robert K. Brayton. Algorithms for discrete function manipulation. In *Proceedings of ICCAD'90*, pages 92–95, 1990.

[Ullmann, 2007] Julian R. Ullmann. Partition search for non-binary constraint satisfaction. *Information Science*, 177:3639–3678, 2007.

[Verhaeghe *et al.*, 2017] Hélène Verhaeghe, Christophe Lecoutre, and Pierre Schaus. Extending compact-table to negative and short tables. In *Proceedings of AAAI'17*, pages 3951–3957, 2017.

[Wang *et al.*, 2016] Ruiwei Wang, Wei Xia, Roland H. C. Yap, and Zhanshan Li. Optimizing Simple Tabular Reduction with a bitwise representation. In *Proceedings of IJCAI'16*, pages 787–795, 2016.

[Xia and Yap, 2013] Wei Xia and Roland H. C. Yap. Optimizing STR algorithms with tuple compression. In *Proceedings of CP'13*, pages 724–732, 2013.