Sliced Table Constraints: Combining Compression and Tabular Reduction

Nebras Gharbi, Fred Hemery, Christophe Lecoutre, Olivier Roussel

Sliced Table Constraints: Combining Compression and Tabular Reduction

Nebras Gharbi, Fred Hemery, Christophe Lecoutre, and Olivier Roussel

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

Abstract. Many industrial applications require the use of table constraints (e.g., in configuration problems), sometimes of significant size. During the recent years, researchers have focused on reducing space and time complexities of this type of constraint. Static and dynamic reduction based approaches have been proposed giving new compact representations of table constraints and effective filtering algorithms. In this paper, we study the possibility of combining both static and dynamic reduction techniques by proposing a new compressed form of table constraints based on frequent pattern detection, and exploiting it in STR (Simple Tabular Reduction).

Introduction

Table constraints, i.e., constraints given in extension by listing the tuples of values allowed or forbidden for a set of variables, are widely studied in constraint programming (CP). This is because such constraints are present in many real-world applications from areas such as design and configuration, databases, and preferences modeling. Sometimes, table constraints provide the unique natural or practical way for a non-expert user to express her constraints. So far, research on table constraints has mainly focused on the development of fast algorithms to enforce generalized arc consistency (GAC), which is a property that corresponds to the maximum level of filtering when constraints are treated independently. GAC algorithms for table constraints have attracted considerable interest, dating back to GAC4 [21] and GAC-Schema [2]. Classical algorithms iterate over lists of tuples in different ways ; e.g., see [2, 19, 18]. A recent AC5-based algorithm has been proposed in [20], and has been shown efficient on table constraints of small arity. For tables constraint of large arity, it is recognized that maintaining dynamically the list of supports in constraint tables does pay off: these are the variants of simple tabular reduction (STR) [23, 15, 16].

Table constraints are important for modeling parts of many problems, but they admit practical boundaries because the memory space required to represent them may grow exponentially with their arity. To reduce space complexity, researchers have focused on various forms of compression. Tries [6], Multi-valued Decision Diagrams (MDDs) [3] and Deterministic Finite Automata (DFA) [22]

are general structures used to represent table constraints in a compact way, so as to facilitate the filtering process. Cartesian product is another classical mechanism to represent compactly large sets of tuples. For instance, it has been applied successfully for handling sets of solutions [10], symmetry breaking [5, 4], and learning [13, 17]. So far, this form of compression has been used in two distinct GAC algorithms for table constraints: by revisiting the general GAC-schema [14] and by combining compressed tuples with STR [24]. The latter work shows how variants STR2 and STR3 can advantageously benefit from compressed tuples when the compression ratio is high.

Recently, we have proposed an original compression approach based on data-mining algorithms [7], where all occurrences of the most frequent patterns in a table are replaced by their indices in a so-called patterns table. Using data-mining techniques for compressing table constraints has also been studied in [11], but in a very different manner since additional variables and values are needed, and constraints are reformulated. The same authors also studied compression of SAT clauses in [12]. In [7], a pattern was defined as a sequence of consecutive values, which prevented us from benefiting of optimized STR variants. In this paper, we propose to relax this condition (of consecutive values), considering any sub-tuple as a possible frequent pattern, and identifying the most frequent ones by means of data-mining techniques. Consequently, every table can be "sliced", where each slice associates a pattern μ with a sub-table containing all extensions of μ that can be found in the original table. We propose an algorithm to deal with sliced table constraints: we build it on the basis of the optimized algorithm STR2.

The paper is organized as follows. After recalling some technical background in Section 1, we present, in Section 2, a compression process for table constraints, detailing the algorithm used to obtain the new form of sliced table constraints. Next, we describe, in Section 3, an optimized algorithm to enforce GAC on sliced table constraints. Finally, after giving some experimental results in Section 4, we conclude.

1 Technical Background

A (discrete) *constraint network* (CN) N is a finite set of n variables "interconnected" by a finite set of e constraints. Each *variable* x has a *domain* which is the finite set of values that can be assigned to x. The *initial* domain of a variable x is denoted by $dom^{init}(x)$ whereas the *current* domain of x is denoted by $dom(x)$; we always have $dom(x) \subseteq dom^{init}(x)$. Each *constraint* c involves an ordered set of variables, called the *scope* of c and denoted by $scp(c)$, and is semantically defined by a *relation*, denoted by $rel(c)$, which contains the set of tuples allowed for the variables involved in c. A (positive) *table constraint* c is a constraint such that $rel(c)$ is defined explicitly by listing the tuples that are allowed by c ; an example is given below. The *arity* of a constraint c is the size of $scp(c)$, and will usually be denoted by r.

Example 1. Let c be a positive table constraint on variables x_1, x_2, x_3, x_4, x_5 with $dom(x_1) = dom(x_2) = dom(x_3) = dom(x_4) = dom(x_5) = \{a, b, c\}$. Table 1 represents the constraint c with 7 allowed tuples.

	x_1	x_2	x_3	x_4	x_5
τ_1	c,	b,	c,	a,	c)
τ_2	a,	a,	b,	c,	a)
τ_3	a,	c,	b,	c,	a)
τ_4	b,	a,	c,	b,	c)
τ_5	b,	a,	a,	b,	b)
τ_6	c,	c,	b,	c,	a)
τ_7	a,	c,	a,	c,	a)

Table 1. Table constraint c on x_1, x_2, x_3, x_4, x_5

Let $X = \{x_1, \ldots, x_r\}$ be an ordered set of variables. An *instantiation* I of X is a set $\{(x_1, a_1), \ldots, (x_r, a_r)\}$ also denoted by $\{x_1 = a_1, \ldots, x_r = a_r\}$ such that $\forall i \in 1..r, a_i \in dom^{init}(x_i)$; X is denoted by $vars(I)$ and each a_i is denoted by $I[x_i]$. An instantiation I is *valid* iff $\forall (x, a) \in I, a \in dom(x)$. An *$r$-tuple* τ on X is a sequence of values $(a_1, \ldots, a_r)$ such that $\forall i \in 1..r, a_i \in dom^{init}(x_i)$; the individual value a_i will be denoted by $\tau[x_i]$. For simplicity, we shall use both concepts of instantiation and tuple interchangeably. For example, an r-tuple τ on $scp(c)$ is *valid* iff the underlying instantiation is valid. An r-tuple τ on $scp(c)$ is a *support* on the r-ary constraint c iff τ is a valid tuple which is allowed by c. If τ is a support on a constraint c involving a variable x and 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 consistency defined as follows:

Definition 1. *A constraint c is* generalized arc consistent *(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 from domains all values that have no support on a constraint. Many algorithms have been devised for establishing GAC according to the nature of the constraints. For table constraints, STR [23] is such an algorithm: it removes invalid tuples during search of supports using a sparse set data structure which separates valid tuples from invalid ones. This method of seeking supports improves search time by avoiding redundant tests on invalid tuples that have already been detected as invalid during previous GAC enforcements. STR2 [15], an optimization of STR, limits some basic operations concerning the validity of tuples and the identification of supports, through the introduction of two important sets called S^{sup} and S^{val} (described later). In the extreme best case, STR2 can be r times faster than STR.

We now introduce the concepts of pattern and sub-table that will be useful for compression.

Definition 2. *A **pattern** μ of a constraint c is an instantiation I of some variables of c. We note $scp(\mu)$ its scope, which is equal to $vars(I)$, $|\mu|$ its length, which is equal to $|scp(\mu)|$, and $nbOcc(\mu)$ its number of occurrences in $rel(c)$, which is $|\{\tau \in rel(c) \mid \mu \subseteq \tau\}|$.*

Example 2. In Table 1, μ_1=$\{x_1 = a, x_4 = c, x_5 = a\}$ and μ_2=$\{x_3 = c, x_5 = c\}$ are patterns of respective lengths 3 and 2, with $scp(\mu_1)$=$\{x_1, x_4, x_5\}$ and $scp(\mu_2)$=$\{x_3, x_5\}$.

Definition 3. *The **sub-table** T associated with a pattern μ of a constraint c is obtained by removing μ from tuples of c that contain μ and ignoring other tuples.*

$$T = \{\tau \setminus \mu \mid \tau \in rel(c) \wedge \mu \subseteq \tau\}$$

The scope of T is $scp(T) = scp(c) - scp(\mu)$

Example 3. Table 2 represents the sub-table associated with the pattern μ_1=$(x_1 = a, x_4 = c, x_5 = a)$ of c, described in Table 1.

x_2	x_3
a	b
c	b
c	a

Table 2. The sub-table T_1 associated with the pattern μ_1 of c

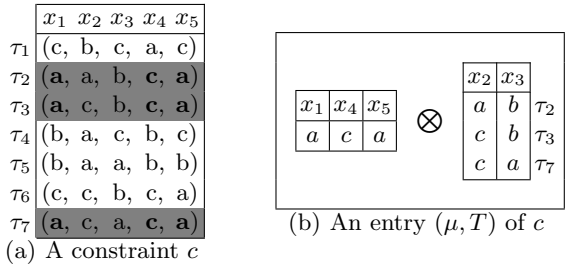
Definition 4. *An **entry** for a constraint c is a pair (μ, T) such that μ is a pattern of c and T is the sub-table associated with μ.*

Since the set of tuples represented by an entry (μ, T) represents in fact the Cartesian product of μ by T, we shall also use the notation $\mu \otimes T$ to denote a constraint entry. Notice that after the slicing process of a constraint into a set of entries, the set of tuples which are not associated with any pattern can be stored in a so called *default entry* denoted by $(\emptyset, T)$.

Example 4. The pattern μ=$(x_1 = a, x_4 = c, x_5 = a)$ of the constraint c, depicted in Figure 1(a), appears in tuples τ_2, τ_3 and τ_7. Thus, μ and the resulting sub-table form an entry for c, as shown in Figure 1(b).

Testing the validity of classical or compressed tuples is an important operation in filtering algorithms of (compressed) table constraints. For sliced table constraints, we extend the notion of validity to constraint entries.

Definition 5. *An entry (μ, T) is* valid *iff at least one tuple of the Cartesian product $\mu \otimes T$ is valid. Equivalently, an entry is valid iff its pattern is valid and its sub-table contains at least one valid sub-tuple.*



(a) A constraint c

(b) An entry (μ, T) of c

Fig. 1. Example of a constraint entry

2 Compression Method

Several data mining algorithms, such as Apriori [1] and FP-Growth [8] among others, can be used to identify frequent patterns. Since our objective is compression, we do not have to identify each possible frequent pattern but only the ones that are useful for compression, and specifically at most one pattern per tuple. The construction of an FP-Tree (Frequent-Pattern Tree) which is the first step in the FP-Growth algorithm is especially well suited to this goal since it identifies each long and frequent pattern. This construction requires only three scans of the table.

We briefly explain the construction of the FP-Tree in the context of table compression, using the constraint given in Table 1 as an example. Details of the general method can be found in [8, 9]. The algorithm takes one parameter *minSupport* (minimum support) which is the minimal number of occurrences of a pattern that we require to consider it as frequent. In our example, we shall use *minSupport*=2 to identify patterns which occur at least twice.

In a first step, we collect the number of occurrences of each value. By abuse of terminology, we shall call frequency the number of occurrences of a value. This step requires one scan of the table. The result on our example is given in Figure 2(a). Then, in a second scan, we sort each tuple in decreasing order of frequency of values. The result is given in Figure 2(b) where the frequency of a value is given in parentheses. Values which have a frequency below the threshold *minSupport* are removed from the tuple (they are identified in bold face) because they cannot appear in a frequent pattern. Once a tuple is sorted and possibly reduced, it is inserted in the FP-Tree which is essentially a trie where each branch represents the frequent part of a tuple and each node contains the number of branches which share that node. Each edge from a parent to its child is labeled with a value. The root node does not have any label.

Figure 3(a) represents the FP-tree obtained on our running example. The first tuple inserted in the tree is the beginning of τ_1, that is $(x_1 = c, x_3 = c, x_5 = c)$. This creates the leftmost branch of the tree. Each node of this branch is given a frequency of 1. The second tuple inserted is $(x_4 = c, x_5 = a, x_1 = a, x_2 = a, x_3 = b)$ which creates the third leftmost branch in the tree (each node having

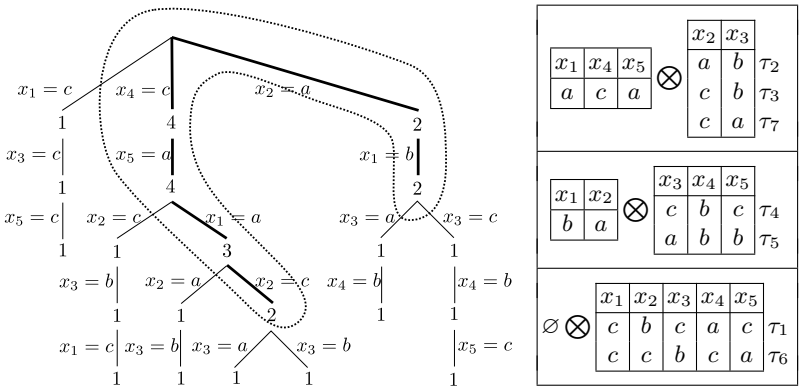
	x_1	x_2	x_3	x_4	x_5
a	3	3	2	1	4
b	2	1	3	2	1
c	2	3	2	4	2

(a) frequencies

τ_1	(2) $x_1 = c$	(2) $x_3 = c$	(2) $x_5 = c$	(1) $\mathbf{x_2 = b}$	(1) $\mathbf{x_4 = a}$
τ_2	(4) $x_4 = c$	(4) $x_5 = a$	(3) $x_1 = a$	(3) $x_2 = a$	(3) $x_3 = b$
τ_3	(4) $x_4 = c$	(4) $x_5 = a$	(3) $x_1 = a$	(3) $x_2 = c$	(3) $x_3 = b$
τ_4	(3) $x_2 = a$	(2) $x_1 = b$	(2) $x_3 = c$	(2) $x_4 = b$	(2) $x_5 = c$
τ_5	(3) $x_2 = a$	(2) $x_1 = b$	(2) $x_3 = a$	(2) $x_4 = b$	(1) $\mathbf{x_5 = b}$
τ_6	(4) $x_4 = c$	(4) $x_5 = a$	(3) $x_2 = c$	(3) $x_3 = b$	(2) $x_1 = c$
τ_7	(4) $x_4 = c$	(4) $x_5 = a$	(3) $x_1 = a$	(3) $x_2 = c$	(2) $x_3 = a$

(b) tuples sorted according to decreasing frequencies

Fig. 2. First two steps of the compression



(a) FP-tree (b) Compressed table

Fig. 3. FP-tree and compressed table

a frequency of 1 at this step). When τ_3 is inserted, the new branch $(x_4 = c, x_5 = a, x_1 = a, x_2 = c, x_3 = b)$ shares its first three edges with the last branch, hence the frequency of the corresponding nodes is incremented and becomes 2. The other tuples are inserted in the same way. In the end, nodes with a frequency below the threshold *minSupport* are pruned. The remaining tree is depicted with thick lines and circled by a dashed line in Figure 3(a).

We now have to identify patterns in the FP-tree which are relevant for compression. Each node of the tree corresponds to a frequent pattern μ which can be read on the path from the root to the node. The frequency f of this pattern is given by the node itself. The savings that can be obtained by factoring this frequent pattern is $|\mu| \times (f - 1)$ values (we can save each occurrence of the pattern but one). In our example, we can see that the pattern $(x_4 = c, x_5 = a)$ can save 6 values, the pattern $(x_4 = c, x_5 = a, x_1 = a)$ can also save 6 values but the pattern $(x_4 = c, x_5 = a, x_1 = a, x_2 = c)$ can save only 4 values. Therefore, we further prune the tree by removing nodes that save less values than their

parent. The leaves of the tree we obtain represent the frequent pattern used in the compression: $(x_4 = c, x_5 = a, x_1 = a)$ and $(x_2 = a, x_1 = b)$.

To complete the compression, we create an entry for each frequent pattern we have identified and fill them in a last scan of the table. For each tuple, we use the FP-tree to identify if the (sorted) tuple starts with a frequent pattern, in which case we add the rest of the tuple to the corresponding sub-table. Tuples which do not start with a frequent pattern are added to the default entry.

Algorithm 1 summarizes the different steps of the compression process.

Algorithm 1: compress(T: table, minSupport: integer)

```
1  compute the frequency of each value of T
2  for i ∈ 1..|T| do
3  |   τ ← T[i]
4  |   sort τ by decreasing order of value frequency and remove values less
   |   frequent than minSupport
5  |   add τ to the FP-Tree (will update the nodes frequency)
6  |   tmp[i] ← τ
7  prune the tree by removing nodes which are less frequent than minSupport or
   such that |μ| × (f − 1) is smaller than for their parent
8  for i ∈ 1..|T| do
9  |   τ ← tmp[i]
10 |   lookup in the tree if τ starts with a frequent pattern μ. If it does not, μ ← ∅
11 |   add T[i] \ μ to the sub-table corresponding to μ
```

3 Filtering Sliced Table Constraints

In order to enforce GAC on sliced table constraints, our idea is to adapt Simple Tabular Reduction (STR), and more specifically the optimized variant STR2, on the compressed form of this kind of constraint. As a sliced table constraint is composed of several entries, each one composed of both a pattern and a sub-table, the filtering process we propose acts at two distinct levels. At a high level, the validity of each entry is checked, and at a low-level, the validity of each pattern and each sub-tuple is checked. Remember that an entry is valid iff both its pattern is valid and at least one tuple from its sub-table is valid (See Definition 5). In this section, we first describe the employed data structures, then we introduce our GAC algorithm, and finally we give an illustration.

3.1 Data structures

A sliced table constraint c is represented by an array `entries`$[c]$ of p entries. Managing the set of valid entries, called *current*[1] entries, is performed as follows:

[1] Current entries correspond to valid entries at the end of the previous evocation of the algorithm.

– `entriesLimit`[c] is the index of the last current entry in `entries`[c]. The elements in `entries`[c] at indices ranging from 1 to `entriesLimit`[c] are the current entries of c.
– removing an entry (that has become invalid) at index i is performed by a call of the form removeEntry(c, i). Such a call swaps the entries at indices i and `entriesLimit`[c], and then decrements `entriesLimit`[c]. Note that the initial order of entries is not preserved.
– restoring a set of entries can be performed by simply changing the value of `entriesLimit`[c].

Each entry in `entries` can be represented as a record composed of a field `pattern` and a field `subtable`. More precisely:

– the field `pattern` stores a partial instantiation μ, and can be represented in practice as a record of two arrays: one for the variables, the scope of the pattern, and the other for the values.
– the field `subtable` stores a sub-table T, and can be represented in practice as a record of two arrays: one for the variables, i.e., the scope of the sub-table T, and the other, a two-dimensional array, for the sub-tuples.

In our presentation, we shall directly handle μ and T without considering all implementation details ; for example, T will be viewed as a two-dimensional array. Managing the set of valid sub-tuples, called *current* sub-tuples, of T, is performed as follows:

– $limit[T]$ is the index of the last current sub-tuple in T. The elements in T at indices ranging from 1 to `limit`[T] are the current sub-tuples of T.
– removing a sub-tuple (that has become invalid) at index i is performed by a call of the form removeSubtuple(T, i). Such a call swaps the sub-tuples at indices i and `limit`[T], and then decrements `limit`[T]. Note that the initial order of sub-tuples is not preserved.
– restoring a set of sub-tuples can be performed by simply changing the value of `limit`[T].

Note that the management of both current entries and current sub-tuples is in the spirit of STR. Also, as in [15], we introduce two sets of variables, called S^{val} and S^{sup}. The set S^{val} contains uninstantiated variables (and possibly, the last assigned variable) whose domains have been reduced since the previous invocation of the filtering algorithm on c. To set up S^{val}, we need to record the domain size of each modified variable x right after the execution of STR-slice on c: this value is recorded in `lastSize`[x]. The set S^{sup} contains uninstantiated variables (from the scope of the constraint c) whose domains contain each at least one value for which a support must be found. These two sets allow us to restrict loops on variables to relevant ones. We also use an array `gacValues`[x] for each variable x. At any time, `gacValues`[x] contains all values in $dom(x)$ for which a support has already been found: hence, values for a variable x without any proved support are exactly those in $dom(x) \setminus$ `gacValues`[x]. Note that the sets

S^{val} and S^{sup} are initially defined with respect to the full scope of c. However, for each sub-table we also shall use local sets S^{lval} and S^{lsup} of S^{val} and S^{sup} as explained later.

3.2 Algorithm

Algorithm 2 is a filtering procedure, called STR-slice, that establishes GAC on a specified sliced table constraint c belonging to a CN N. Lines 1–10, which are exactly the same as those in Algorithm 5 of [15], allow us to initialize the sets S^{val}, S^{sup} and `gacValues`. Recall that S^{val} must contain the last assigned variable, denoted by lastPast(P), if it belongs to the scope of c. Lines 11–22 iterate over all current entries of c. To test the validity of an entry, we check first the validity of the pattern μ (Algorithm 3), and then, only when the pattern is valid, we check the validity of the sub-table T by scanning it (Algorithm 4). If an entry is no more valid, it is removed at line 22. Otherwise, considering the values that are present in the pattern, we have to update `gacValues` as well as S^{sup} when a first support for a variable is found. Lines 23–30, which are exactly the same as those in Algorithm 5 of [15], manage the reduction of domains: unsupported values are removed at line 25 and if the domain of a variable x becomes empty, an exception is thrown at line 27. Also, the set of variables X_{evt} reduced by STR-slice is computed and returned so that these "events" can be propagated to other constraints.

Algorithm 4 is an important function, called scanSubtable, of STR-slice. Its role is to iterate over all current (sub)tuples of a given sub-table, in order to collect supported values and to remove invalid tuples. Note that when this function is called, we have the guarantee that the pattern associated with the sub-table is valid (note the "and then" short-circuit operator at line 14 of Algorithm 2). The first part of the function, lines 1–10, allow us to build the local sets S^{lval} and S^{lsup} from S^{val} and S^{sup}. Such sets are obtained by intersecting S^{val} with $scp(T)$ and S^{sup} with $scp(T)$, respectively. Once the sets S^{lval} and S^{lsup} are initialized, we benefit from optimized operations concerning validity checking and support seeking, as in STR2. The second part of the function, lines 9–21, consists in iterating over all current sub-tuples of T. This is a classical STR2-like traversal of a set of tuples. Finally, line 22 returns true when there still exists at least one valid sub-tuple.

It is interesting to note the lazy synchronization performed between the global unique set S^{sup} and the specific local sets S^{lsup} (one such set per sub-table). When a variable x is identified as "fully supported", it is immediately removed from S^{sup} (see line 19 of Algorithm 2 and line 18 of Algorithm 4). Consequently, that means that the next sub-tables (entries) will benefit from such a reduction, but the information is only transmitted at initialization (lines 6–8 of Algorithm 4). On the other hand, once initialized, the global set S^{val} is never modified during the execution of STR-slice.

Backtracking issues: In our implementation, entries and tuples can be restored by modifying the value of the limit pointers (`entriesLimit`[c] and `limit`[T]

Algorithm 2: STR-slice(c: constraint)

Input : c is a sliced table constraint of the CN N to be solved
Output : the set of variables in $scp(c)$ with reduced domain

```
   // Initialization of sets S^val and S^sup, as in STR2
1  S^val ← ∅
2  S^sup ← ∅
3  if lastPast(P) ∈ scp(c) then
4      S^val ← S^val ∪ {lastPast(P)}
5  foreach variable x ∈ scp(c) | x ∉ past(P) do
6      gacValues[x] ← ∅
7      S^sup ← S^sup ∪ {x}
8      if |dom(x)| ≠ lastSize[c][x] then
9          S^val ← S^val ∪ {x}
10         lastSize[c][x] ← |dom(x)|
   // Iteration over all entries of c
11 i ← 1
12 while i ≤ entriesLimit[c] do
13     (μ, T) ← entries[c][i]              // ith current entry of c
14     if isValidPattern(μ) and then scanSubtable(T) then
15         foreach variable x ∈ scp(μ) | x ∈ S^sup do
16             if μ[x] ∉ gacValues[x] then
17                 gacValues[x] ← gacValues[x] ∪ {μ[x]}
18                 if |dom(x)| = |gacValues[x]| then
19                     S^sup ← S^sup \ {x}
20         i ← i + 1
21     else
22         removeEntry(c, i)            // entriesLimit[c] decremented
   // domains are now updated and X_evt computed, as in STR2
23 X_evt ← ∅
24 foreach variable x ∈ S^sup do
25     dom(x) ← gacValues[x]
26     if dom(x) = ∅ then
27         throw INCONSISTENCY
28     X_evt ← X_evt ∪ {x}
29     lastSize[c][x] ← |dom(x)|
30 return X_evt
```

Algorithm 3: isValidPattern(μ: pattern): Boolean

```
1  foreach variable x ∈ scp(μ) do
2      if μ[x] ∉ dom(x) then
3          return false
4  return true
```

Algorithm 4: scanSubtable(T: sub-table): Boolean

```
   Input  : T is a sub-table coming from an entry of the constraint c
   Output : true iff there is at least one valid tuple in the sub-table T
   // Initialization of local sets S^lval and S^lsup from S^val and S^sup
 1 S^lval ← ∅
 2 foreach variable x ∈ S^val do
 3 |   if x ∈ scp(T) then
 4 |   |   S^lval ← S^lval ∪ {x}
 5 S^lsup ← ∅
 6 foreach variable x ∈ S^sup do
 7 |   if x ∈ scp(T) then
 8 |   |   S^lsup ← S^lsup ∪ {x}
   // Iteration over all (sub)tuples of T
 9 i ← 1
10 while i ≤ limit[T] do
11 |   τ ← T[i]                    // ith current sub-tuple of T
12 |   if isValidSubtuple(S^lval, τ) then
13 |   |   foreach variable x ∈ S^lsup do
14 |   |   |   if τ[x] ∉ gacValues[x] then
15 |   |   |   |   gacValues[x] ← gacValues[x] ∪ {τ[x]}
16 |   |   |   |   if |dom(x)| = |gacValues[x]| then
17 |   |   |   |   |   S^lsup ← S^lsup \ {x}
18 |   |   |   |   |   S^sup ← S^sup \ {x}
19 |   |   i ← i + 1
20 |   else
21 |   |   removeSubtuple(T, i))            // limit[T] decremented
22 return limit[T] > 0
```

Algorithm 5: isValidSubtuple(S^{lval}: variables, τ: tuple): Boolean

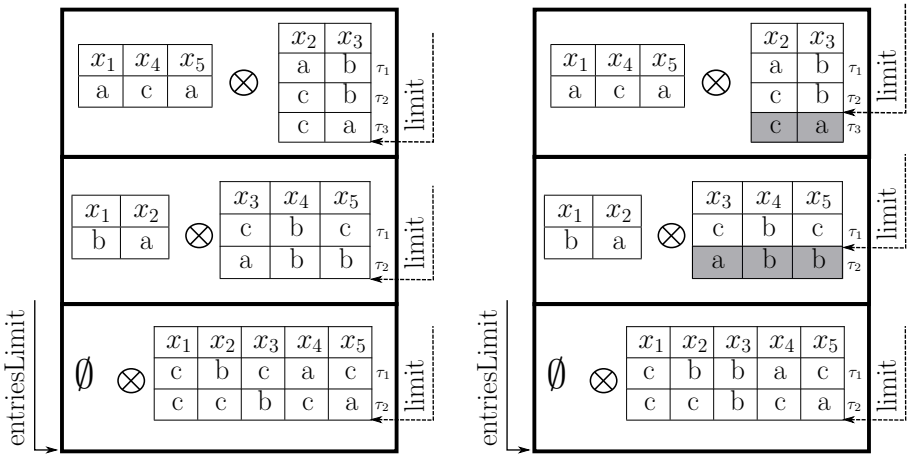
```
1 foreach variable x ∈ S^lval do
2 |   if τ[x] ∉ dom(x) then
3 |   |   return false
4 return true
```

for each sub-table T of c), recorded at each search depth. Restoration is then achieved in $O(1 + p)$ (for each constraint) where p is the number of entries. However, by introducing a simple data structure, it is possible to only call the restoration procedure when necessary, limiting restoration complexity to $O(1)$ in certain cases: it suffices to register the limit pointers that need to be updated when backtracking, and this for each level. When the search algorithm back-

tracks, we also have to deal with the array `lastSize`. As mentioned in [15], we can record the content of such an array at each depth of search, so that the original state of the array can be restored upon backtracking.

As GAC-slice is a direct extension of STR2, it enforces GAC.

3.3 Illustration



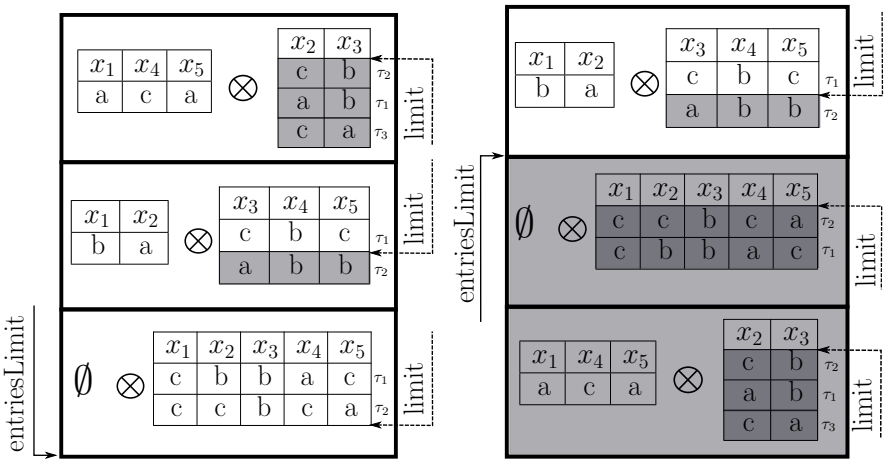
(a) Before the call (b) After the call

Fig. 4. STR-slice called on a slice table constraint after the event $x_3 \neq a$

Figures 4 and 5 illustrate the different steps for filtering a sliced table constraint, when STR-slice is called after an event. In Figure 4, considering that the new event is simply $x_3 \neq a$ (i.e., the removal of the value a from $dom(x_3)$), STR-slice starts checking the validity of the current entries (from 1 to `entriesLimit`). So, for the first entry, the validity of the pattern μ=$\{x_1 = a, x_4 = c, x_5 = a\}$ is first checked. Since μ remains valid (our hypothesis is that the event was only $x_3 \neq a$), the sub-table of the first entry is scanned. Here, only the sub-tuple $\{x_2 = c, x_3 = a\}$ is found invalid, which modifies the value of `limit` for the sub-table of this first entry. After the call to STR-slice, the constraint is as in Figure 4(b).

In Figure 5, considering now that the new event is $x_3 \neq b$, we start again with the first current entry. Figuring out that the pattern is still valid, we check the validity of the associated sub-tuples. Since the sub-tuple $\{x_2 = a, x_3 = b\}$ is no more valid, it is swapped with $\{x_2 = c, x_3 = b\}$. This latter sub-tuple is then also found invalid, which sets the value of `limit` to 0. This is illustrated in Figure 5(a). As the sub-table of the first entry is empty, the entry is removed by swapping its position with that of last current entry. After the call to STR-slice,

the constraint is as in Figure 5(b) (note that a second swap of constraint entries has been performed).



(a) After the scan of the first entry (b) After the call

Fig. 5. From Figure 4(b), STR-slice called after the event $x_3 \neq b$

4 Experimental results

In order to show the practical interest of our approach to represent and filter sliced table constraints, we have conducted an experimentation (with our solver AbsCon) using a cluster of bi-quad cores Xeon processors at 2.66 GHz node with 16GiB of RAM under Linux. Because STR2 and STR3 belong to the state-of-the-art GAC algorithms for table constraints, we compare the respective behaviors of STR2, STR3 and STR-slice on different series of instances[2] involving positive table constraints with arity greater than 2. STR1 is also included as a baseline.

For STR-slice, we select exclusively frequent patterns with a number of occurrences at least equal to 10% of the number of tuples in the table. This value was obtained after several experiments, similarly we chose 10 for the minimum size of sub-tables. Automatically tuning, on a specific instance, the frequency threshold for patterns and the minimum sub-table size is part of our future work.

We use MAC with the dom/ddeg variable ordering and lexico as value ordering heuristic, to solve all these instances. A time-out of $1,200$ seconds was set per instance. The two chosen heuristics guarantee that we explore the very same search tree regardless of the filtering algorithm used.

[2] Available at `http://www.cril.univ-artois.fr/CSC09`.

Instance	#ins	STR1	STR2	STR3	STR-slice
a7-v24-d5-ps05	11	298.05	147.73	189.14	**115.30** (66% – 5.74)
bdd	70	44.53	**13.44**	99.21	20.35 (86% – 0.59)
crossword-ogd	43	90.05	39.35	**25.69**	29.59 (75.51% – 0.36)
crossword-uk	43	95.20	45.88	**44.33**	47.21 (88.69% – 0.18)
renault	46	19.66	14.39	**13.37**	17.20 (47.15% – 0.67)

Table 3. Mean CPU time (in seconds) to solve instances from different series (a time-out of 1,200 seconds was set per instance) with MAC. Mean compression ratio and CPU time are given for STR-slice between parentheses.

Instance	STR1	STR2	STR3	MDD	STR-slice
a7-v24-d5-ps0.5-psh0.7-9	879	334	367	**25.5**	200 (69% – 5.41)
a7-v24-d5-ps0.5-psh0.9-6	353	195	324	**16.6**	174 (62% – 5.82)
bdd-21-2713-15-79-11	78.5	**23.5**	48.5	82.6	31.7 (88.05% – 0.28)
crossword-ogd-vg12-13	799	342	**208**	$> 1,200$	242 (73.46% – 0.74)
crossword-uk-vg10-13	1,173	**576**	589	$> 1,200$	598 (89.63% – 0.48)

Table 4. CPU time (in seconds) on some selected instances solved by MAC.

Table 3 shows mean results (CPU time in seconds) per series. For each series, the number of tested instances is given by #ins ; it corresponds to the number of instances solved by all three variants within $1,200$ seconds. Note that the mean compression ratios and CPU times (in seconds) are also given for STR-slice between parentheses. We define the compression ratio as the size of the sliced tables over the size of the initial tables, where the size of a (sliced) table is the number of values over all patterns and (sub-)tables. The results in Table 3 show that STR-slice is competitive with both STR2 and STR3. Surprisingly, although the compression ratio obtained for the instances of the series *renault* is rather encouraging, the CPU time obtained for STR-slice is disappointing. We suspect that the presence of many constraints with small tables in the *renault* instances is penalizing for STR-slice because, in that case, the overhead of managing constraint entries is not counterbalanced by the small absolute spatial reduction. Table 4 presents the results obtained on some instances.

In term of space, STR3 is the variant that uses the most amount of memory (sometimes by a very large factor). STR-slice, although requiring a few additional data structures is the cheapest STR variant in term of memory (approximately as indicated by the compression ratio in Tables 3 and 4). Note that other compression approaches of the literature such as those based on MDDs [3] may outperform STR variants when the compression ratio is (very) high. This

is the case, for example, on the instances of series *a7-v24-d5-ps05*. However, on other series such as *crossword*, the MDD approach can be outperformed by a large factor by the STR variants (on hard crossword instances, STR2, STR3 and STR-slice are usually about 5 times faster than MDD).

A general observation from this rather preliminary experimentation is that STR-slice is a competitor to STR2 and STR3, but a not a competitor that takes a real advantage. Several perspectives to improve this situation are developed in the conclusion.

Conclusion

In this paper, we have presented an original approach for filtering table constraints: it combines a new compression technique using the concept of sliced table constraints and an optimized adaptation of the tabular reduction (as in STR2). Our preliminary experimentation shows that STR-slice is a competitor to the state-of-the-art STR2 and STR3 algorithms. To make STR-slice indispensable, many further developments are necessary. First, we think that the tuning of the parameters used for guiding compression should be automatized (possibly, employing some machine learning techniques). STR-slice could then benefit from a better compression. Second, we believe that, in the rising context of big data, new constraint problems should emerge rapidly where constraints could be of (very) large arity and involve very large tables. STR-slice could advantageously handle such "huge" constraints, especially if we consider that slicing could be conducted recursively on the sub-tables (another perspective of this work). Finally, we think that the concept of sliced table constraints is interesting on its own for modeling, as certain forms of conditionality can be represented in a simple and natural way, directly with sliced table constraints.

Acknowledgments

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

References

1. R. Agrawal and R. Srikant. Fast algorithms for mining association rules in large databases. In *Proceedings of the 20th International Conference on Very Large Data Bases*, pages 487–499, 1994.
2. C. Bessiere and J. Régin. Arc consistency for general constraint networks: preliminary results. In *Proceedings of IJCAI'97*, pages 398–404, 1997.
3. K. Cheng and R. Yap. An MDD-based generalized arc consistency algorithm for positive and negative table constraints and some global constraints. *Constraints*, 15(2):265–304, 2010.
4. T. Fahle, S. Schamberger, and M. Sellman. Symmetry breaking. In *Proceedings of CP'01*, pages 93–107, 2001.

5. F. Focacci and M. Milano. Global cut framework for removing symmetries. In *Proceedings of CP'01*, pages 77–92, 2001.
6. I.P. Gent, C. Jefferson, I. Miguel, and P. Nightingale. Data structures for generalised arc consistency for extensional constraints. In *Proceedings of AAAI'07*, pages 191–197, 2007.
7. N. Gharbi, F. Hemery, C. Lecoutre, and O. Roussel. Optimizing STR algorithms with tuple compression. In *Proceedings of JFPC'13*, pages 143–146, 2013.
8. J. Han, J. Pei, and Y. Yin. Mining frequent patterns without candidate generation. In *Proceedings of SIGMOD'00*, pages 1–12, 2000.
9. J. Han, J. Pei, Y. Yin, and R. Mao. Mining frequent patterns without candidate generation: A frequent-pattern tree approach. *Data Mining and Knowledge Discovery*, 8(1):53–87, 2004.
10. P.D. Hubbe and E.C. Freuder. An efficient cross product representation of the constraint satisfaction problem search space. In *Proceedings of AAAI'92*, pages 421–427, 1992.
11. S. Jabbour, L. Sais, and Y. Salhi. A mining-based compression approach for constraint satisfaction problems. *CoRR*, abs/1305.3321, 2013.
12. S. Jabbour, L. Sais, Y. Salhi, and T. Uno. Mining-based compression approach of propositional formulae. In *Proceedings of CIKM'2013*, pages 289–298, 2013.
13. G. Katsirelos and F. Bacchus. Generalized nogoods in CSPs. In *Proceedings of AAAI'05*, pages 390–396, 2005.
14. G. Katsirelos and T. Walsh. A compression algorithm for large arity extensional constraints. In *Proceedings of CP'07*, pages 379–393, 2007.
15. C. Lecoutre. STR2: Optimized simple tabular reduction for table constraints. *Constraints*, 16(4):341–371, 2011.
16. C. Lecoutre, C. Likitvivatanavong, and R. Yap. A path-optimal GAC algorithm for table constraints. In *Proceedings of ECAI'12*, pages 510–515, 2012.
17. C. Lecoutre, L. Sais, S. Tabary, and V. Vidal. Transposition Tables for Constraint Satisfaction. In *Proceedings of AAAI'07*, pages 243–248, 2007.
18. C. Lecoutre and R. Szymanek. Generalized arc consistency for positive table constraints. In *Proceedings of CP'06*, pages 284–298, 2006.
19. O. Lhomme and J.C. Régin. A fast arc consistency algorithm for n-ary constraints. In *Proceedings of AAAI'05*, pages 405–410, 2005.
20. J.-B. Mairy, P. van Hentenryck, and Y. Deville. An optimal filtering algorithm for table constraints. In *Proceedings of CP'12*, pages 496–511, 2012.
21. R. Mohr and G. Masini. Good old discrete relaxation. In *Proceedings of ECAI'88*, pages 651–656, 1988.
22. G. Pesant. A regular language membership constraint for finite sequences of variables. In *Proceedings of CP'04*, pages 482–495, 2004.
23. J.R. Ullmann. Partition search for non-binary constraint satisfaction. *Information Science*, 177:3639–3678, 2007.
24. W. Xia and R. Yap. Optimizing STR algorithms with tuple compression. In *Proceedings of CP'13*, pages 724–732, 2013.