Segmented Tables: an Efficient Modeling Tool for Constraint Reasoning

#2953

Abstract

These last years, there has been a growing interest for structures like tables and decision diagrams in Constraint Programming (CP). This is due to the universal character of these structures, enabling the representation of any (group of) constraints under extensional form, and to the efficient filtering algorithms developed for constraints based on (ordinary/short/compressed/smart) tables and multi-valued decision diagrams. In this paper, we propose the concept of segmented tables where entries in tables can combine ordinary values, universal values ($*$) and sub-tables. Segmented tables can be seen as a generalization of compressed tables. We propose an algorithm enforcing Generalized Arc Consistency (GAC) on segmented table constraints, and show their modeling and practical interests on a realistic problem.

Introduction

Constraint Programming (CP) is a modeling paradigm that has been shown quite effective for solving various forms of combinatorial problems, by means of highly optimized inference and search algorithms (Dechter 2003; Apt 2003; Rossi, van Beek, and Walsh 2006; Lecoutre 2009). The strength of CP is its ability to take any kind of information into consideration, at modeling time, because of the availability of structures permitting a universal form of representation. These structures actually allow us to introduce constraints enumerating, in extension, what can be accepted (or not): they are called *table* constraints. For example, in the domain of data-mining, a user can ask for frequent patterns together with some specific characteristics, which can be expressed as a combination of user's constraints (typically, arithmetic or table constraints) that can be easily added to the model due to the flexible nature of CP (Guns et al. 2017). More specifically, in a dataset of purchases, the user may want to know which pattern is frequent and the day at which it is frequent (Bessiere, Lazaar, and Maamar 2018).

Interestingly enough, the practical efficiency of filtering algorithms for table constraints has been substantially improved over the past decade, leading to the state-of-the-art Compact-Table (Demeulenaere et al. 2016), and STR-bit (Wang et al. 2016). However, one major issue remains the space required to store the tables, i.e., all the tuples that are accepted (or forbidden) by the constraints. To address it, several compact forms of tables have been introduced in the literature, notably *short* tables (Jefferson and Nightingale 2013; Verhaeghe, Lecoutre, and Schaus 2017), allowing the presence of the universal value '*', and *compressed* tables (Katsirelos and Walsh 2007; Xia and Yap 2013; Verhaeghe et al. 2017), allowing us to deal with subsets of values in tuples. Sliced tables (Gharbi et al. 2014) and *smart* tables (Mairy, Deville, and Lecoutre 2015) are two other forms of sophisticated compact representation.

It is important to note that, sometimes, some of these tables simply happen to be simple and natural choices for dealing with tricky situations. This is the case when no known (global) constraint exists or when a logical combination of (small) constraints is needed for filtering efficiency reasons. Basically, table constraints offer the user a direct way to handle disjunction (a choice between tuples), and this is clearly emphasized with smart tables (Mairy, Deville, and Lecoutre 2015). If ever needed, another argument showing the importance of universal structures like tables, and also MDDs (Multi-valued Decision Diagrams), is the rising of tabulation techniques, i.e., the process of converting sub-problems into tables, by hand, by means of heuristics (Akgun et al. 2018) or by annotations (Dekker et al. 2017).

Compressed tables generalize short tables since each element of a compressed tuple can be any subset of values (and consequently, the full set of values, just like '*'). For example, the compressed tuple $\tau = (a, \{b, c\}, b, \{a, b, c\})$ captures 6 *ordinary* tuples, among which we find (a, b, b, a) and (a, c, b, c). In this paper, we introduce segmented tables that generalize compressed tables since subsets are possibly extended over several variables. For example, the segmented tuple $\Gamma = (\{(a, a), (a, c), (b, b)\}, *, \{(a, b), (c, c)\}$ is composed of a first segment representing a (sub-)table over two variables, a second segment being the universal value $*$ and a third segment representing again a (sub-)table over two variables. Any ordinary tuple obtained from the implicit Cartesian product of these segments is compactly represented by the segmented tuple, as for example (a, a, a, a, b) and (b, b, c, c, c).

The paper is organized as follows. After some technical background, we introduce segmented tables and constraints. Then, we provide a synthetic view, followed by a fully detailed description, of an algorithm enforcing GAC on seg-

mented table constraints. Before giving some perspectives and conclusions, we show the modeling and practical interest of segmented tables on a challenging problem called CD (Crosswords Design), used in the 2018 XCSP3 Competition.

Technical Background

A *Constraint Network* (CN) P is composed of a sequence $\texttt{vars}(P)$ of distinct variables and a set $\texttt{ctrs}(P)$ of constraints. Each *variable* x has an associated domain, $\texttt{dom}(x)$, that contains the finite set of values that can be assigned to it. Each *constraint* c involves a sequence of distinct variables, called the *scope* of c and denoted by $\texttt{scp}(c)$, and is semantically defined by a *relation*, $\texttt{rel}(c)$, that contains the set of tuples allowed for the variables involved in c. When a tuple τ is allowed (or accepted) by a constraint c, we say that c is *satisfied* by τ. The *arity* of a constraint c is $|\texttt{scp}(c)|$. Let X be a sequence of variables, an *instantiation* of X maps each variable $x \in X$ to a value in $\texttt{dom}(x)$. An instantiation is *complete* for P iff $X = \texttt{vars}(P)$. A *solution* of P is a complete instantiation satisfying all constraints of P; the set of solutions of P is denoted by $\texttt{sols}(P)$; when $\texttt{sols}(P) \neq \emptyset$, P is said to be *satisfiable*. If $X = \langle x_1, \ldots, x_p \rangle$ and $Y = \langle y_1, \ldots, y_q \rangle$ are two sequences of p and q variables, the sequence $\langle x_1, \ldots, x_p, y_1, \ldots, y_q \rangle$ of $p+q$ variables is denoted by $X \odot Y$.

For simplicity, a variable-value pair (x, a) such that $x \in \texttt{vars}(P)$ and $a \in \texttt{dom}(x)$ is called a *literal* (of P). Let $\tau = (a_1, \ldots, a_r)$ be a tuple of values associated with a sequence of variables $\texttt{vars}(\tau) = \langle x_1, \ldots, x_r \rangle$. The ith value a_i of τ is denoted by $\tau[i]$ or $\tau[x_i]$. τ is *valid* iff $\forall i \in 1..r, \tau[i] \in \texttt{dom}(x_i)$. τ is a *support* on a constraint c iff $\texttt{vars}(\tau) = \texttt{scp}(c)$ and τ is a valid tuple allowed by c. When a support exists on c, c is said to be *satisfiable*. 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* the literal (x, a) on c; equivalently, we say that the literal (x, a) is supported (on c). Enforcing Generalized Arc Consistency (GAC) on a constraint c means removing all literals (values) without any support on c.

A *table constraint* c is a constraint such that $\texttt{rel}(c)$ is explicitly defined by listing (in a table) the tuples that are allowed by c. Over the years, there have been many developments about compact forms of tables. Ordinary tables contain *ordinary* tuples, i.e., classical sequences of values as in $(1, 2, 0)$. Short tables can additionally contain *short* tuples, which are tuples involving the special symbol $*$ as in $(0, *, 2)$, and compressed tables can additionally contain *compressed* tuples, which are tuples involving sets of values as in $(0, \{1, 2\}, 3)$. Assuming that the tuples mentioned just above are associated with the ordered set of variables $\{x_1, x_2, x_3\}$, in $(0, *, 2)$, x_2 can take any value from its domain and in $(0, \{1, 2\}, 3)$, x_2 can take the value 1 or the value 2. Smart tables are composed of *smart* tuples, which are tuples containing arithmetic expressions (column constraints). Finally, a *basic* smart table is a restricted form of smart table where column constraints are unary. In term of expressiveness, basic smart tables are equivalent to compressed tables.

Segmented Tables and Constraints

A segmented table contains segmented tuples that are built from so-called segments. In this section, we introduce some formal definitions before giving an illustration.

Definition 1 *A* segment*, or* tuple segment*, is a constraint γ that can take one of the three following forms:*

- $x_i = *$*, a unary tautology constraint, always holding whatever is the value assigned to x_i; it is called a* tautology *segment;*
- $x_i = a$*, a unary equality constraint, holding only when x_i is set to the value a; it is called an* equality *segment;*
- $\langle x_{i_1}, x_{i_2}, \ldots, x_{i_{r_i}} \rangle \in T$*, a table constraint, holding when the values assigned to the sequence of variables corresponds to a tuple accepted by the table T (which contains ordinary tuples of length r_i); it is called a* table *segment.*

Note that for any segment γ, $\texttt{scp}(\gamma)$ denotes the sequence of variables involved in γ; we have $\texttt{scp}(\gamma) = \langle x_i \rangle$ for equality and tautology segments, and $\texttt{scp}(\gamma) = \langle x_{i_1}, x_{i_2}, \ldots, x_{i_{r_i}} \rangle$ for table segments. The table T required for a table segment γ will be denoted by $\texttt{table}(\gamma)$.

Now, we consider a sequence of r (distinct) variables $X = \langle x_1, x_2, \ldots, x_r \rangle$.

Definition 2 *A* segmented tuple Γ *over X is a sequence of segments $\langle \gamma_1, \gamma_2, \ldots, \gamma_p \rangle$ such that:*

$$X = \texttt{scp}(\gamma_1) \odot \texttt{scp}(\gamma_2) \odot \cdots \odot \texttt{scp}(\gamma_p).$$

The semantics of Γ, i.e., the set of tuples represented by Γ, is given by $\texttt{sols}(P^\Gamma)$ where P^Γ is the CN defined by $\texttt{vars}(P^\Gamma) = X$ and $\texttt{ctrs}(P^\Gamma) = \{\gamma_1, \gamma_2, \ldots, \gamma_p\}$.

In some occasions, we shall need to refer to the specific types of segments. This is why we denote by Γ^{tt}, Γ^{eq} and Γ^{tb} the respective sets of tautology, equality and table segments in Γ.

Definition 3 *A* segmented table constraint*, or* segmented constraint *for short, is a constraint c defined by a segmented table, denoted by $\texttt{seg\_table}(c)$, which is a set T of segmented tuples over $\texttt{scp}(c)$. The semantics of c is given by $\texttt{rel}(c) = \bigcup_{\Gamma \in T} \texttt{sols}(P^\Gamma)$.*

Example 1 *Let $X = \langle x_1, x_2, \ldots, x_{10} \rangle$ be a sequence of 10 variables with domains $\{a, b, c\}$. Figure 1 shows a segmented table constraint over X. Its table contains 3 segmented tuples Γ_1, Γ_2 and Γ_3. The first segmented tuple Γ_1 is defined as a sequence of five segments $\langle \gamma_{1_1}, \gamma_{1_2}, \gamma_{1_3}, \gamma_{1_4}, \gamma_{1_5} \rangle$. We have $\gamma_{1_1} : \langle x_1, x_2, x_3 \rangle \in \{(a, b, a), (b, a, c), (c, b, b)\}$, $\gamma_{1_2} : x_4 = b$, $\gamma_{1_3} : \langle x_5, x_6 \rangle \in \{(a, a), (c, c)\}$, $\gamma_{1_4} : x_7 = *$, $\gamma_{1_5} : \langle x_8, x_9, x_{10} \rangle \in \{(b, a, a), (b, c, c), (c, b, a)\}$. As an example of tuple represented (accepted) by Γ_1, we find $(a, b, a, b, a, a, a, b, a, a)$. It is rather easy to observe that all segmented tuples do not overlap, i.e., do not share any tuple. Consequently, the number of ordinary tuples represented by these 3 segmented tuples is exactly $(3 \times 2 \times 3 \times 3) + (4 \times 3 \times 3) + (3 \times 3 \times 3 \times 3) = 165$. This shows the possible compression benefit of using segmented tables.*

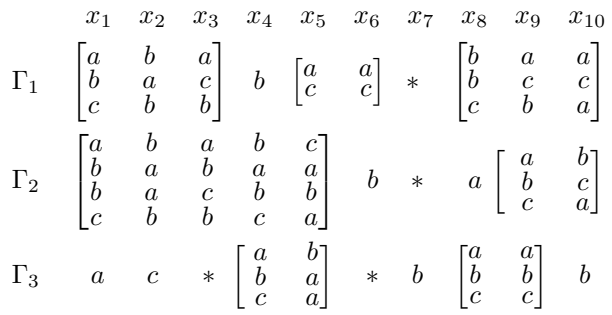

Figure 1: A segmented table constraint, composed of three segmented tuples Γ_1, Γ_2 and Γ_3.

When looking for supports of literals (in the context of a filtering procedure), one has to determine which segmented tuples are valid. Validity for a segment γ means that the constraint γ is satisfiable. Similarly, validity for a segmented tuple Γ means that Γ is satisfiable (more precisely, the set of segments/constraints in Γ is satisfiable).

Definition 4 *A segment γ is* valid *iff γ is satisfiable.*

This is the case when γ is a tautology segment, or γ is an equality segment $x = a$ with $a \in \texttt{dom}(x)$, or γ is a table segment and $\texttt{table}(\gamma) \cap \prod_{x \in \texttt{scp}(\gamma)} \texttt{dom}(x) \neq \emptyset$. Note that the intersection of $\texttt{table}(\gamma)$ with the Cartesian product of the current domains of variables in $scp(\gamma)$ is exactly the set of supports on γ, meaning that γ is satisfiable when the intersection is not empty.

Definition 5 *A segmented tuple Γ is* valid *iff Γ is satisfiable, i.e.,* $\texttt{sols}(P^\Gamma) \neq \emptyset$.

Proposition 1 *A segmented tuple Γ is* valid *iff every segment in Γ is valid.*

Proof: Because, by definition, segments do not overlap (share variables), when every segment in Γ is valid, we necessarily have $\texttt{sols}(P^\Gamma) \neq \emptyset$. ■

As an illustration, let us consider again Figure 1. If b is removed from $\texttt{dom}(x_{10})$, then Γ_3 becomes clearly invalid. If a and c are respectively removed from $\texttt{dom}(x_5)$ and $\texttt{dom}(x_6)$, then Γ_1 becomes invalid because its third segment becomes invalid.

Synthetic View of Filtering

Like many filtering algorithms developed for constraints in extensional form (i.e., using structures like tables or decision diagrams), the principle is to explore the underlying structure of the constraints so as to identify (and to delete) the literals (values) that are not supported. In this section, we present a synthetic view of an original filtering algorithm dedicated to segmented table constraints. Important implementation details (notably, the data structures) will be given in the next section.

For this high-level description, we only need to introduce the structure gacValues. For each variable x in the scope of the constraint c to be filtered, the filtering algorithm computes $\texttt{gacValues}[x]$, the set of values for x that are supported on c.

Algorithm 1: Synthetic Filtering Algorithm

```
1  Function filter(c: Segmented Table Constraint)
2      gacValues[x] ← ∅, ∀x ∈ scp(c)
3      foreach Γ ∈ seg_table(c) do
4          if Γ is valid then
               // we can collect values
5              foreach x = * ∈ Γ^tt do
6                  gacValues[x] ← dom(x)
7              foreach x = a ∈ Γ^eq do
8                  add a to gacValues[x]
9              foreach γ ∈ Γ^tb do
10                 SUPs ← table(γ) ∩ ∏_{x∈scp(γ)} dom(x)
11                 foreach x ∈ scp(γ) do
12                     foreach a ∈ {τ[x] : τ ∈ SUPs} do
13                         add a to gacValues[x]
14     dom(x) ← gacValues[x], ∀x ∈ scp(x)
```

In Algorithm 1, Function filter() must be called everytime a segmented table constraint c must be filtered. To start, the sets $\texttt{gacValues}[x]$ are initialized. Then, every segmented tuple of the table is iterated over: when a segmented tuple Γ is found to be valid, literals supported by Γ can be collected. After processing the segmented table, the sets $\texttt{gacValues}[x]$ represent the new domains for the variables involved in c, indirectly indicating which values must be deleted, and possibly causing a domain wipe out (i.e., an empty domain).

When a segmented tuple is valid, it remains to identify supported literals. This is the role of Lines 5-13 in Algorithm 1. For a tautology segment $x = *$, all values in $\texttt{dom}(x)$ are supported, and then can be directly collected. For an equality segment $x = a$, only the value a is supported. Finally, for a table segment, one has to identify the set SUPs of current supports on this segment. For each variable x involved in the segment, we can consider the projection of SUPs on x: $\{\tau[x] : \tau \in \texttt{SUPs}\}$ is the set of values for x occurring in one tuple of SUPs. These projections correspond to supported values, and then can be collected.

As an illustration, let us consider the first table segment of Γ_2 in Figure 1. If we suppose that b has been removed from $\texttt{dom}(x_3)$, then we have $\texttt{SUPs} = \{(a, b, a, b, c), (b, a, c, b, b)\}$. For x_1, x_2, x_3, x_4 and x_5, the supported values that can be collected are then $\{a, b\}$, $\{a, b\}$, $\{a, c\}$, $\{b\}$ and $\{b, c\}$, respectively.

Now, let us suppose that b has been removed from $\texttt{dom}(x_4)$. We can see in Figure 2 that some parts of the segmented table become invalid: this is displayed in red color. Now, after collecting, we have $\texttt{gacValues}[x_i] = \{a, b, c\}$ for all variables x_i except for x_4 and x_5 for which we have $\texttt{gacValues}[x_4] = \{a, c\}$ and $\texttt{gacValues}[x_5] = \{a, b\}$. As

$\texttt{dom}(x_4)$ was already $\{a, c\}$ (our initial assumption), after the collecting process, we can only deduce that c must be removed from $\texttt{dom}(x_5)$.

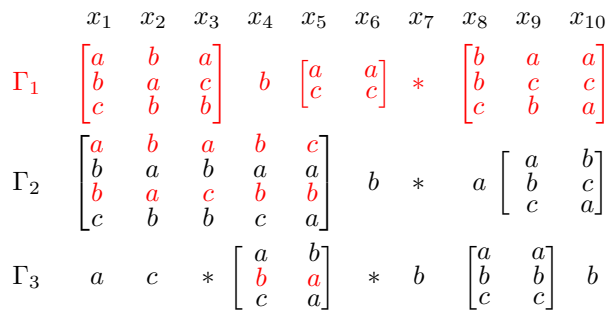

Figure 2: If b is removed from $\texttt{dom}(x_4)$, some parts of the segmented table becomes invalid (displayed in red color). We can then infer that $x_5 \neq c$.

Detailed Description of the Algorithm

After having introduced a synthetic filtering view, we propose now a rigorous detailed implementation. To enforce GAC on segmented table constraints, we have to deal with a main segmented table, and some secondary ordinary tables attached to table segments. Being careful about efficiency, we chose to use tabular reduction, which is a time-tested technique for dynamically maintaining tables. Indeed, based on the structure of sparse sets (Briggs and Torczon 1993; le Clément de Saint-Marcq et al. 2013), variants of Simple Tabular Reduction (STR) have been proved[1] to be quite competitive (Ullmann 2007; Lecoutre 2011; Lecoutre, Likitvivatanavong, and Yap 2015). For the main table of segmented tuples, we maintain the set of valid segmented tuples by partitioning it in two parts. More specifically, at any time, we aim at respecting the following invariant: the segmented tuples indexed from 1 to the value of `tableLimit` are valid whereas segmented tuples with an index strictly greater than `tableLimit` are invalid. For simplicity, we shall denote by Γ_i the segmented tuple indexed by i in the current table at a given time. In the process of maintaining the table, if a segmented tuple Γ_i becomes invalid, it suffices to swap it with the one indexed by `tableLimit`, and then to decrement `tableLimit`; this is illustrated in Figure 3, where segmented tuples Γ_1 and Γ_3 are swapped. Similarly, for any table segment γ, the valid (ordinary) tuples are indexed from 1 to the value of $\texttt{tableLimit}[\gamma]$. In the context of a table segment γ (and so, without any ambiguity), we shall denote by τ_i the tuple indexed by i in the current table of γ.

The class SegmentedConstraint, Algorithm 2, allows us to represent any segmented table constraint c, with the possibility of enforcing GAC at any time by simply calling Method enforceGAC(). As fields of this class we first find `scp` for representing the scope $\langle x_1, \ldots, x_r \rangle$ of c. As indicated above, for dealing with tables, we simply use `tableLimit` and

[1]The state-of-the-art algorithm Compact-Table also uses tabular reduction (sparse sets) to maintain the list of non-zero words.

Algorithm 2: Class SegmentedConstraint

```
1  Method enforceGAC()
2      S^val ← {x ∈ scp : prevSizes[x] ≠ |dom(x)|}
3      S^sup ← {x ∈ scp : |dom(x)| > 1}
4      foreach x ∈ scp do
5          gacValues[x] ← ∅
6      traverseTable()
7      if tableLimit = 0 then
8          return FAILURE
9      filterDomains()
10     foreach variable x ∈ S^val ∪ S^sup do
11         prevSizes[x] ← |dom(x)|
12     return SUCCESS

13 Method traverseTable()
14     i ← 1
15     while i ≤ tableLimit do
16         if isValidSegmentedTuple(Γ_i) then
17             collectValues(Γ_i)
18             i ← i + 1
19         else                    // Γ_i must be removed
20             swap Γ_i and Γ_tableLimit
21             tableLimit ← tableLimit − 1

22 Method isValidSegmentedTuple(Γ)
23     foreach x = a ∈ Γ^eq do
24         if x ∈ S^val ∧ a ∉ dom(x) then
25             return false
26     foreach γ ∈ Γ^tb do
27         S ← S^val ∩ scp(γ)
28         if S = ∅ then
29             continue
30         i ← 1
31         while i ≤ tableLimit[γ] do
32             if isValidSubtuple(τ_i, S) then
33                 i ← i + 1
34             else                // τ_i must be removed
35                 swap τ_i and τ_tableLimit[γ]
36                 tableLimit[γ] ← tableLimit[γ] − 1
37         if tableLimit[γ] = 0 then
38             return false

39 Method isValidSubtuple(τ, S)
40     foreach x ∈ S do
41         if τ[x] ∉ dom(x) then
42             return false
43     return true
```

`tableLimit`$[\gamma]$, while getting access to tuples with notations Γ_i and τ_i. We also have three fields $\texttt{S}^{\texttt{val}}$, $\texttt{S}^{\texttt{sup}}$ and `prevSizes` in the spirit of STR2 (Lecoutre 2011). The set $\texttt{S}^{\texttt{val}}$ contains variables whose domains have been reduced since the previous call to Method enforceGAC() on c. To set up $\texttt{S}^{\texttt{val}}$, we need to record the domain size of each variable x right after the execution of enforceGAC() on c: this

value is recorded in $\texttt{prevSizes}[x]$. The set $\texttt{S}^{\texttt{sup}}$ contains unbound variables 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. Finally, $\texttt{gacValues}[x]$ is the set used to collect values proved to be GAC during the filtering process.

Algorithm 3: Class SegmentedConstraint (continued)

```
1  Method collectValues(Γ)
2      foreach x = * ∈ Γ^tt do
3          if x ∈ S^sup then
4              remove x from S^sup
5      foreach x = a ∈ Γ^eq do
6          add a to gacValues[x]
7          if |gacValues[x]| = |dom(x)| then
8              remove x from S^sup
9      foreach γ ∈ Γ^tb do
10         S ← S^sup ∩ scp(γ)
11         if S = ∅ then
12             continue
13         i ← 1
14         while i ≤ tableLimit[γ] do
15             foreach x ∈ S do
16                 if τ_i[x] ∉ gacValues[x] then
17                     add τ_i[x] to gacValues[x]
18                     if |gacValues[x]| = |dom(x)| then
19                         remove x from S^sup
20             i ← i + 1
21 Method filterDomains()
22     foreach variable x ∈ S^sup do
23         foreach value a ∈ dom(x) do
24             if a ∉ gacValues[x] then
25                 remove a from dom(x)
```

At the beginning of Method enforceGAC(), the sets $\texttt{S}^{\texttt{val}}$, $\texttt{S}^{\texttt{sup}}$ and $\texttt{gacValues}[x]$ (initially, no value has been proved to be GAC) are first initialized. Then, the main method traverseTable() is called to update tables and collect values. If after such a 'traversal', the value of `tableLimit` is 0, it means that no more segmented tuple is valid, and consequently a failure is identified. Otherwise, domains are updated by calling Method filterDomain() in order to remove the values that have not been collected in sets `gacValues`. Before returning SUCCESS (for indicating that filtering has been achieved without generating a domain wipe-out), the array `prevSizes` is modified in anticipation of the next call.

Method traverseTable() simply iterates over the segmented tuples from the current table (by considering indexes from 1 to `tableLimit`). When a segmented tuple Γ_i is found to be valid, Method collectValues() is called. Otherwise, Γ_i is removed from the current table.

To check whether a segmented tuple Γ is valid, Method isValidSegmentedTuple() is called. Because tautology segments are always valid, we only focus on equality and table

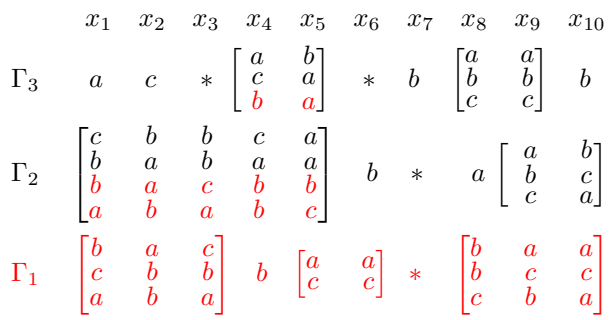

Figure 3: If b is removed from $\texttt{dom}(x_4)$, invalid parts of the segmented table are removed by swapping.

segments. For an equality segment $x = a$, we just check if x must be tested according to $\texttt{S}^{val}$ (although this test can be safely discarded) and if a belongs to the current domain of x. For a table segment, we start by computing the set $\texttt{S}$ of variables occurring in both $\texttt{S}^{val}$ and $\texttt{scp}(\gamma)$. If ever this set $\texttt{S}$ is empty, it means that nothing has changed for γ since the previous call to Method enforceGAC(), and consequently the table of γ is up-to-date (this is why we 'continue'). Otherwise, we iterate over the current table of the segment to only keep the tuples that are valid (tests being performed by Method isValidSubtuple()). If the table becomes empty, it disqualifies the segmented tuple by returning 'false'.

Finally, Method collectValues(), Algorithm 3, allows us to collect all values that admit a support on at least a tuple. For a tautology segment $x = *$, we simply remove x from $\texttt{S}^{sup}$ because, from now on, no more supports have to be sought for x. For an equality segment $x = a$, we indicate that a has just been found a support (although, this was possibly done) by adding a to $\texttt{gacValues}[x]$, and we determine if x can be discarded from $\texttt{S}^{sup}$ by comparing its size with that of $\texttt{dom}(x)$. For a table segment, we start by computing the set $\texttt{S}$ of variables occurring in both $\texttt{S}^{sup}$ and $\texttt{scp}(\gamma)$. If ever this set $\texttt{S}$ is empty, it means that no relevant support can be found in the current table of γ (consequently, we can 'continue'). Otherwise, we iterate over the current table of γ, looking for supports of values with respect to variables in $\texttt{S}$.

Proposition 2 *When called on a segmented table constraint c, Method* enforceGAC()*, Algorithm 2 establishes GAC on c.*

Proof: Let us suppose that the segmented table corresponds to an ordinary table, that is, every segment is an equality segment. In that case, we obtain a classical filtering STR scheme, and we know that GAC is reached. If tautology segments are also involved, the segmented table corresponds to a short table, and GAC is guaranteed (Jefferson and Nightingale 2013). Now, in case table segments are present, one can rather easily check that validity and collecting operations are correct. ■

The worst-case space complexity of representing a segmented table constraint c is as follows. First, note that `scp`, $\texttt{S}^{\texttt{val}}$, $\texttt{S}^{\texttt{sup}}$ and `prevSizes` are $O(r)$. Because representing a tautology or equality segment is $O(1)$, representing all such segments is $O(rt)$ with t being the number of segmented

tuples. Now, for each table segment γ, let us denote the arity and the size of the table of γ by r^γ and t^γ, and let us denote by c^{tb} the set of table segments over all segmented tuples (i.e., in the entire table). The space complexity for a segment table is $O(r^\gamma t^\gamma)$. The overall space complexity is then $O(rt + \Lambda)$ with $\Lambda = \sum_{\gamma \in c^{tb}} r^\gamma t^\gamma$.

The worst-case time complexity of calling enforceGAC() is as follows. Without any table segment, it is $O(rt+rd)$ because in that case, handling the main table (traverseTable() is $O(rt)$ and filtering domains is $O(rd)$. For any table segment γ, checking validity of tuples and collecting values is $O(r^\gamma t^\gamma)$. The overall time complexity is then $O(rt+rd+\Lambda)$.

A very useful feature of tabular reduction (more generally, sparse sets) is the possibility of restoring a table in constant time. During backtrack search, one has simply to record the table limit at each search level. When a backtrack must be performed, it suffices to change the limit in $O(1)$ by using the one recorded at the right level. For an ordinary table, backtracking is $O(1)$, but for a segmented table it is $O(k)$ where k is the number of table segments.

Case Study: Crosswords Design

In this section, both modeling and practical benefits of using segmented table constraints are shown on a difficult optimization problem called Crosswords Design (CD); this is our proof of concept. The problem was used in the COP track of the 2018 XCSP3 Competition[2]. The problem is stated as follows: given a grid order n and two dictionaries, a main dictionary and an auxiliary thematic dictionary, the objective is to fill up a grid of size $n \times n$ with words contained in these dictionaries as well as with some black points/cells (BPs). This is an optimization problem because each word w from the thematic dictionary has value $|w|$ (the length of the word), and the objective is to maximize the overall value. There is one restriction: it is not possible to have two adjacent BPs (on a row or on a column).

In what follows, we introduce a first model based on segmented tables before succinctly presenting two others models based on 'classical' short table constraints. Our aim is to compare these three models so as to highlight the contribution offered by segmented tables.

Using Segmented Tables. The first model we propose for CD is called CD^{seg}, only involves $2 \times n$ constraints, because we can reason with a unique constraint per row and per column. This is rather noteworthy because do note that BPs can be put anywhere in the grid. Without any loss of generality, we introduce m as being the maximal number of words put on a same row or same column. It is always possible to set m in order to avoid discarding any potential solution. For example, if $n = 5$, the maximal number of words is 3, as visible in the following pattern: 1 BP 1 BP 1 where 1 here refers to 1-letter words. So, setting $m = 3$ guarantees us that no solution can be lost. In our model, the variables are:

- $x_{i,j}$, the letter put in the grid at the intersection of row i and column j, with $i \in 1..n \wedge j \in 1..n$. Possible letters are 'a', 'b', ..., 'z', and BP.

[2] http://www.cril.univ-artois.fr/XCSP18/

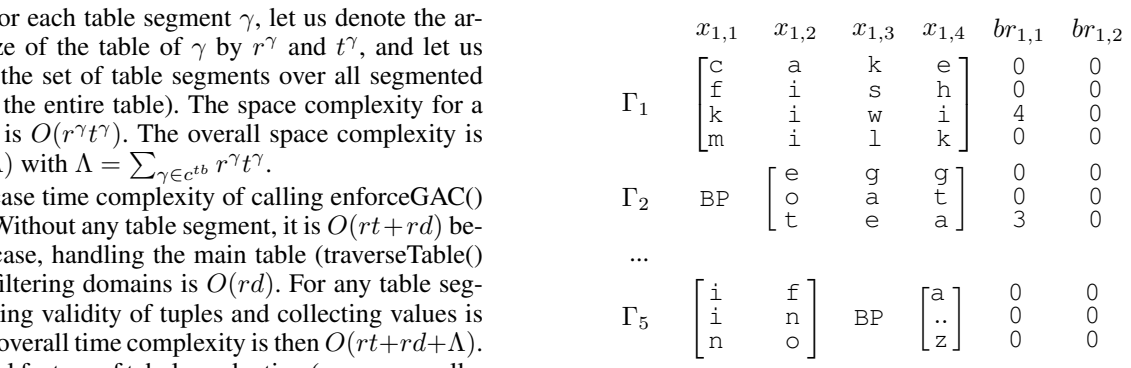

Figure 4: A Segmented Table for CD ($n = 4$).

- $br_{i,k}$, the benefit of putting the kth word on row i, from left to right, with $i \in 1..n \wedge k \in 1..m$. For example, if the second word put on row i comes from the thematic dictionary and is of length 5, then $br_{i,2} = 5$. But if instead there is just one word (of size n) put on row i, then $br_{i,2}$ is necessarily equal to 0.
- $bc_{j,k}$, the benefit of putting the kth word on column j, from top to bottom, with $j \in 1..n \wedge k \in 1..m$.

For each row i, we have exactly one segmented table constraint cr_i whose scope is $\texttt{scp}(cr_i) = \{x_{i,j} : j \in 1..n\} \cup \{br_{i,k} : k \in 1..m\}$; the arity of such constraints is then $n+m$. The segmented table contains a segmented tuple for each valid n-pattern, where a valid n-pattern is an alternation of positive numbers and BP, summing up to n (when considering BP as being equal to 1). For example, the set of valid 4-patterns is {4, BP 3, 3 BP, BP 2 BP, 2 BP 1, 1 BP 2, 1 BP 1 BP, BP 1 BP 1}. We illustrate now how segmented tuples are built using a very small set of words. For our illustration, we consider 26 1-letter words $\{a \dots z\}$, three 2-letter words {in, if, no}, three 3-letter words {egg, oat, tea} and four 4-letter words {cake, fish, kiwi, milk}. We assume here that the thematic words are 'kiwi' and 'tea'. Each constraint involves 6 variables (since $m = 2$) and a table containing 8 segmented tuples built from the 8 valid 4-patterns. This is illustrated in Figure 4 for the first row constraint, where the first pattern gives Γ_1, the second pattern Γ_2, and so on.

It is important to note that the constraint forbidding the presence of two adjacent BPs is directly taken into consideration by the segmented tables (tuples). Of course, we proceed similarly with columns: there are n segmented table constraints for dealing with the n columns. Finally the objective is quite simple. It is :

$$\text{MAXIMIZE} \sum_{\substack{i \in 1..n \\ k \in 1..m}} br_{i,k} + \sum_{\substack{j \in 1..n \\ k \in 1..m}} bc_{j,k}$$

Using Short Tables. Segmented tables allow us to put different words together (i.e., in the same constraint), without memory explosion, because of the compactness of the underlying Cartesian product. Now, we turn to a more classical way of modeling, where each word is managed independently. Actually, this is the model used for generating the

instances of the 2018 XCSP3 Competition.

The model, called CD^{sho}, of the competition, involves the three 2-dimensional arrays of variables x, br and bc, introduced for CD^{seg}. It also includes these additional variables:

- $r_{i,k}$, the kth word on row i, from left to right, with $i \in 1..n \wedge k \in 1..m$. The value 0 denotes that the word does not exist, while another value stands for the index (starting at 1) of a word in the two merged dictionaries, referred as DICT. For example, if $r_{i,k} = 1500$ it means that the kth word on row i is the 1500th from DICT.
- $c_{j,k}$, the kth word on column j, from top to bottom, with $j \in 0..n-1 \wedge k \in 0..m-1$.
- $pr_{i,k}$, the position (index of the grid column) of the kth word on row i. The value 0 denotes that the word does not exist (because no more room exists due to words already put on the left), while any other value stands for the position (starting at 1) of the column where the word starts. For example, if $pr_{i,k} = 3$ it means that the kth word on row i starts at the 3rd column.
- $pc_{j,k}$, the position (index of the grid row) of the kth word on column j.

In this model, we introduce $m-1$ constraints per row (and similarly per column): for any $k \in 1..m-1$, we have a short table constraint linking $pr_{i,k}$, $r_{i,k}$, $pr_{i,k+1}$, $br_{i,k}$, and variables in $\{x_{i,j} : j \in 1..n\}$. The connection between two successive words is established by the presence of the variables $pr_{i,k}$, $r_{i,k}$, and $pr_{i,k+1}$ in the same constraint: if the position of the kth word is v and the word has length l, then the position of the k+1th word is $v + l$ (or 0 if the value is too large). Due to lack of space, we do not provide details about the short tables.

A second model, called CD^{sho2}, involves much more constraints. The view point is different: the number p of horizontal words of a certain length l is pre-computed. Then, when we have $4 \times p$ variables indicating which words are chosen in the merged dictionaries, their benefits, their lengths, and their positions. We proceed similarly vertically.

Practical Evaluation. Words have been taken from some Romanian dictionaries: (i) a long *main* dictionary containing a list of $134,938$ words and (ii) a short *thematic* dictionary containing 278 words. For our experimentation, we have mainly compared the two models CD^{seg} and CD^{sho} (the model CD^{sho2} does not scale at all because it quickly runs out of memory). We have executed in the same environment a classical solver that was able to read instances generated for these two models. More specifically, we have used the 13 instances used in the XCSP3 competition (model CD^{sho}) and generated 13 equivalent instances from model CD^{seg}: n ranges from 3 to 15, while m was arbitrarily limited to 4.

Table 1 compares the efficiency of our solver on 5 instances (for both models), in terms of the number of wrong decisions (#wrong) and CPU time. For these rather "small" instances, with n ranging from 3 to 6, optimality needed to be proved. For guiding search, we have used the classical heuristic `dom/wdeg` (Boussemart et al. 2004) and two basic value ordering heuristics: `FirstVal` and `LastVal`, which respectively select the first and last values in the domain of a variable. Clearly, the model using segmented tables makes the solver more efficient, with a speedup usually between 2 and 10, both in terms of wrong decisions and CPU time.

Table 1: Number of wrong decisions (#wrong) and CPU time to solve CD instances (optimality searched).

n (opt)	MODEL CD^{sho}				MODEL CD^{seg}			
	FIRSTVAL		LASTVAL		FIRSTVAL		LASTVAL	
	#wrong	CPU	#wrong	CPU	#wrong	CPU	#wrong	CPU
3 (12)	30	3.6	18	3.5	7	1.6	**4**	**1.5**
4 (24)	126	4.2	120	4.9	153	2.4	**102**	**2.2**
5 (38)	11,430	29.7	3,171	12.1	4,208	8.7	**2,030**	**4.8**
6 (54)	1,834K	10,412	144K	851	235K	567	**96K**	**136**

Table 2: Best bound and CPU time (to reach this bound, given between parentheses) to solve CD instances. `TO` and `MO` stand for Time-Out (2,520s) and Memory-Out.

n	MODEL CD^{sho}				MODEL CD^{seg}	
	CHOCO	CONCRETE	COSOCO	OSCAR	FIRSTVAL	LASTVAL
5	38 (16)	38 (128)	38 (10)	38 (5.0)	38 (8.4)	**38 (4.4)**
6	54 (317)	48 (359)	54 (1,578)	52 (1,098)	54 (408)	**54 (20)**
7	MO	55 (234)	55 (2,424)	MO	55 (748)	**55 (6.9)**
8	MO	67 (174)	4 (59)	MO	60 (2,273)	**67 (40)**
9	MO	76 (1,160)	TO	MO	65 (2,200)	**76 (21)**
10	MO	84 (2,105)	TO	MO	71 (2,338)	**84 (7.5)**

Table 2 shows how competitive is the model CD^{seg}, compared to CD^{sho}, since when the heuristic `LastVal` is used, our solver largely outperforms all competitors of the competition. Here, our goal was to find a solution of cost given by the best competitor in the competition, and then to stop. For example, our solver using `LastVal` as heuristic can find a solution of cost 84 in 7.5 seconds for $n = 10$; this has to be compared with the $2,105$ seconds of Concrete. When the heuristic `FirstVal`, not suited at all to the way optimization is defined (a maximization), is used, only one solver, Concrete, remains superior.

Conclusion

In this paper, we have introduced segmented tables, which are a very general form of tabular representation. Indeed, segmented tables generalize compressed tables where subsets (sub-tables) are limited to one variable only. They also generalize sliced tables where a pattern (sub-tuple) is combined with a unique sub-table. We have presented a detailed description of a filtering GAC algorithm. Interestingly, segmented tables can be further extended to integrate short table segments, and other arithmetic constraints (e.g., other unary constraints like in basic smart tables). This is a perspective of this work. We could envision to represent segmented tables under the form of MDDs. However, if the compression is low (as it is the case when representing words taken from a dictionary), we do believe that a MDD-based filtering algorithm would not be competitive (this was shown for ordinary tables). Because of their segmented structure, developing efficient parallel filtering algorithms for segmented table constraints seems also a promising perspective. Finally, segmented tables are a new powerful modeling tool, as shown in the paper with a challenging problem.

References

Akgun, O.; Gent, I.; Jefferson, C.; Miguel, I.; Nightingale, P.; and Salamon, A. 2018. Automatic discovery and exploitation of promising subproblems for tabulation. In *Proceedings of CP'18*.

Apt, K. 2003. *Principles of Constraint Programming*. Cambridge University Press.

Bessiere, C.; Lazaar, N.; and Maamar, M. 2018. User's constraints in itemset mining. In *24th International Conference, CP 2018, Lille, France, 2018, Proceedings*, 537–553.

Boussemart, F.; Hemery, F.; Lecoutre, C.; and Sais, L. 2004. Boosting systematic search by weighting constraints. In *Proceedings of ECAI'04*, 146–150.

Briggs, P., and Torczon, L. 1993. An efficient representation for sparse sets. *ACM Letters on Programming Languages and Systems* 2(1-4):59–69.

Dechter, R. 2003. *Constraint processing*. Morgan Kaufmann.

Dekker, J.; Bjordal, G.; Carlsson, M.; Flener, P.; and Monette, J.-N. 2017. Auto-tabling for subproblem presolving in minizinc. *Constraints* 22(4):512–529.

Demeulenaere, J.; Hartert, R.; Lecoutre, C.; Perez, G.; Perron, L.; Régin, J.-C.; and Schaus, P. 2016. Compact-table: efficiently filtering table constraints with reversible sparse bit-sets. In *Proceedings of CP'16*, 207–223.

Gharbi, N.; Hemery, F.; Lecoutre, C.; and Roussel, O. 2014. Sliced table constraints: Combining compression and tabular reduction. In *Proceedings of CPAIOR'14*, 120–135.

Guns, T.; Dries, A.; Nijssen, S.; Tack, G.; and Raedt, L. D. 2017. Miningzinc: A declarative framework for constraint-based mining. *Artif. Intell.* 244:6–29.

Jefferson, C., and Nightingale, P. 2013. Extending simple tabular reduction with short supports. In *Proceedings of IJCAI'13*, 573–579.

Katsirelos, G., and Walsh, T. 2007. A compression algorithm for large arity extensional constraints. In *Proceedings of CP'07*, 379–393.

le Clément de Saint-Marcq, V.; Schaus, P.; Solnon, C.; and Lecoutre, C. 2013. Sparse-sets for domain implementation. In *Proceeding of TRICS'13*, 1–10.

Lecoutre, C.; Likitvivatanavong, C.; and Yap, R. 2015. STR3: A path-optimal filtering algorithm for table constraints. *Artificial Intelligence* 220:1–27.

Lecoutre, C. 2009. *Constraint networks: techniques and algorithms*. ISTE/Wiley.

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

Mairy, J.; Deville, Y.; and Lecoutre, C. 2015. The smart table constraint. In *Proceedings of CPAIOR'15*, 271–287.

Rossi, F.; van Beek, P.; and Walsh, T., eds. 2006. *Handbook of Constraint Programming*. Elsevier.

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

Verhaeghe, H.; Lecoutre, C.; Deville, Y.; and Schaus, P. 2017. Extending compact-table to basic smart tables. In *Proceedings of CP'17*, 297–307.

Verhaeghe, H.; Lecoutre, C.; and Schaus, P. 2017. Extending compact-table to negative and short tables. In *Proceedings of AAAI'17*, 3951–3957.

Wang, R.; Xia, W.; Yap, R.; and Li, Z. 2016. Optimizing Simple Tabular Reduction with a bitwise representation. In *Proceedings of IJCAI'16*, 787–795.

Xia, W., and Yap, R. 2013. Optimizing STR algorithms with tuple compression. In *Proceedings of CP'13*, 724–732.