Tables segmentées : un outil de modélisation efficace pour le raisonnement par contraintes

Gilles Audemard[1] **Christophe Lecoutre**[1] **Mehdi Maamar**[1]

[1] CRIL, Univ. Artois & CNRS, 62300 Lens, France
{audemard,lecoutre}@cril.fr

Résumé

Ces dernières années, il y a eu un intérêt croissant pour les structures de tables et de diagrammes de décision en programmation par contraintes (CP). Ceci est dû au caractère universel de ces structures, permettant la représentation de toute (ou groupe de) contrainte sous forme extensionnelle, et aux algorithmes de filtrage efficaces développés pour des contraintes basées sur des tables (ordinaires / étoilées / compressées / smart) et des diagrammes de décision à valeurs multiples. Dans cet article, nous proposons le concept de tables segmentées où les entrées dans les tables peuvent combiner des valeurs ordinaires, des valeurs universelles ($*$) et des sous-tables. Les tables segmentées peuvent être considérées comme une généralisation des tables compressées. Nous proposons un algorithme établissant la cohérence d'arc généralisée (GAC) sur des contraintes de tables segmentées, et montrons l'intérêt pratique, ainsi qu'en terme de modélisation, sur un problème difficile.

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.

La programmation par contraintes (CP) est un paradigme de modélisation qui s'est avéré efficace pour résoudre diverses formes de problèmes combinatoires, au moyen d'algorithmes d'inférence et de recherche hautement optimisés [4, 2, 14, 11]. La force de CP est sa capacité à prendre en compte tout type d'information, au stade de la modélisation, en raison de la disponibilité de structures permettant une forme universelle de représentation. Ces structures permettent d'introduire des contraintes énumérant, en extension, ce qui peut être accepté (ou non) : elles sont appelées contraintes *table*. Par exemple, en fouille de données, un utilisateur peut rechercher des motifs fréquents avec certaines fonctionnalités, ce qui peut être exprimé par une combinaison de contraintes (généralement, des contraintes arithmétiques ou de table) qui peuvent être facilement ajoutées au modèle en raison de la flexibilité de CP [8].

Il est intéressant de noter que l'efficacité pratique des algorithmes de filtrage pour les contraintes table a été considérablement améliorée au cours de la dernière décennie, conduisant à un algorithme état de l'art appelé Compact-Table [6]. Cependant, un problème majeur reste l'espace nécessaire pour stocker les tables, c'est-à-dire l'ensemble des tuples acceptés (ou interdits) par les contraintes. Pour y remédier, plusieurs formes compactes de tables ont été introduites dans la littérature, notamment les tables *étoilées* [9, 16], permettant l'utilisation de la valeur universelle '*', et les tables *compressées* [10, 17, 15], autorisant l'inclusion de sous-ensembles de valeurs dans les tuples. Les tables *sliced* [7] et les tables *smart* [13] sont deux autres représentations compactes sophistiquées.

Il est important de noter que, parfois, les tables sont simplement des choix simples et naturels pour faire face à des situations délicates. C'est le cas lorsqu'aucune contrainte (globale) adéquate n'existe ou lorsqu'une combinaison logique de (petites) contraintes doit être représentée sous forme d'une contrainte table unique pour des raisons d'efficacité. Si nécessaire, un autre

argument montrant l'importance des structures universelles comme les tables, mais aussi les MDD (Multivalued Decision Diagrams), est la montée en puissance des techniques de tabulation, c'est-à-dire le processus de conversion des sous-problèmes en tables, à la main, à l'aide d'heuristiques [1] ou par annotations [5].

Les tables compressées généralisent les tables étoilées puisque chaque élément d'un tuple compressé peut être donné par n'importe quel sous-ensemble de valeurs (et par conséquent, l'ensemble complet de valeurs, tout comme $*$). Par exemple, le tuple compressé $\tau = (a, \{b, c\}, b, \{a, b, c\})$ capture 6 tuples *ordinaires*, parmi lesquels on trouve (a, b, b, a) et (a, c, b, c). Dans cet article, nous introduisons des tables segmentées qui généralisent les tables compressées puisque les sous-ensembles peuvent être étendus à plusieurs variables. Par exemple, $\Gamma = (\{(a, a), (a, c), (b, b)\}, *, \{(a, b), (c, c)\})$ est un tuple segmenté composé d'un premier segment représentant une (sous-)table sur deux variables, un deuxième segment avec la valeur universelle $*$ et un troisième segment représentant à nouveau une (sous-)table sur deux variables. Tout tuple ordinaire obtenu à partir du produit cartésien implicite de ces segments est représenté de manière compacte par le tuple segmenté, comme par exemple (a, a, a, a, b) et (b, b, c, c, c) . Notez qu'une forme générale de raisonnement logique avec plusieurs contraintes (segments) a été proposée dans [12].

Pour tout savoir sur les tables segmentées, nous invitons le lecteur à se référer à l'article publié dans les actes de la conférence ECAI'20 [3]. Dans cet article, le concept de contraintes de tables segmentées est introduit, ainsi qu'une vue synthétique, suivie d'une description plus détaillée, d'un algorithme appliquant GAC sur des contraintes de tables segmentées. L'intérêt en terme de modélisation et d'efficacité pratique est démontré sur un problème d'optimisation difficile consistant à générer des grilles de mots croisés (avec position libre des cases noires) appelé CD (Crosswords Design), utilisé lors des compétitions XCSP3.

Remerciements Ce travail a reçu le soutien du projet CPER Data de la région Hauts-de-France.

Références

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

[2] K.R. Apt : *Principles of Constraint Programming*. Cambridge University Press, 2003.

[3] G. Audemard, C. Lecoutre et M. Maamar : Segmented tables : An efficient modeling tool for constraint reasoning. *In Proceedings of ECAI'20*, pages 315–322, 2020.

[4] R. Dechter : *Constraint processing*. Morgan Kaufmann, 2003.

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

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

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

[8] T. Guns, A. Dries, S. Nijssen, G. Tack et L. De Raedt : Miningzinc : A declarative framework for constraint-based mining. *Artificial Intelligence*, 244:6–29, 2017.

[9] C. Jefferson et P. Nightingale : Extending simple tabular reduction with short supports. *In Proceedings of IJCAI'13*, pages 573–579, 2013.

[10] G. Katsirelos et T. Walsh : A compression algorithm for large arity extensional constraints. *In Proceedings of CP'07*, pages 379–393, 2007.

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

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

[13] J.-B. Mairy, Y. Deville et C. Lecoutre : The smart table constraint. *In Proceedings of CPAIOR'15*, pages 271–287, 2015.

[14] F. Rossi, P. van Beek et T. Walsh, éditeurs. *Handbook of Constraint Programming*. Elsevier, 2006.

[15] H. Verhaeghe, C. Lecoutre, Y. Deville et P. Schaus : Extending Compact-Table to basic smart tables. *In Proceedings of CP'17*, pages 297–307, 2017.

[16] H. Verhaeghe, C. Lecoutre et P. Schaus : Extending Compact-Table to negative and short tables. *In Proceedings of AAAI'17*, pages 3951–3957, 2017.

[17] W. Xia et R. Yap : Optimizing STR algorithms with tuple compression. *In Proceedings of CP'13*, pages 724–732, 2013.

Segmented Tables: an Efficient Modeling Tool for Constraint Reasoning

Gilles Audemard[1] and **Christophe Lecoutre**[1] and **Mehdi Maamar**[1]

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.

1 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 [10, 2, 29, 19]. 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 allow us to introduce constraints enumerating, in extension, what can be accepted (or not): they are called *table* constraints. For example, in data mining, a user can ask for frequent patterns together with some specific features, 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 [14].

Interestingly, 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 [12], and STRbit [33]. 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 [17, 32], allowing the use of the universal value '*', and *compressed* tables [18, 34, 31], allowing subsets of values in tuples. Sliced tables [13] and *smart* tables [26] are two other sophisticated compact representation.

It is important to note that, sometimes, tables simply happen to be simple and natural choices for dealing with tricky situations. This is the case when no adequate (global) constraint exists or when a logical combination of (small) constraints must be represented as a unique table constraint for efficiency reasons. 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, using heuristics [1] or by annotations [11].

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, $\Gamma = (\{(a, a), (a, c), (b, b)\}, *, \{(a, b), (c, c)\})$ is a segmented tuple 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). Note that a general form of logically reasoning with several constraints (segments) was proposed in [24], but on an AC-6 [3] basis.

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 segmented 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 XCSP3 Competitions.

2 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 (accepted) by a constraint c, we say that c is *satisfied* by τ. The *arity* of a constraint c is $|\texttt{scp}(c)|$. An *instantiation* of a sequence of variables 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; $\texttt{sols}(P)$ denote the set of solutions of P; when $\texttt{sols}(P) \neq \emptyset$, P is *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 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 $\texttt{vars}(\tau) =$

[1] CRIL, Univ. Artois & CNRS, Lens, France

$\langle x_1, \ldots, x_r \rangle$. 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 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 sequence of variables $\langle x_1, x_2, x_3 \rangle$, 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.

3 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 refer to the specific types of segments. This is why Γ^{tt}, Γ^{eq} and Γ^{tb} denote 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 $\texttt{rel}(c) = \bigcup_{\Gamma \in T} \texttt{sols}(P^\Gamma)$.*

$$
\begin{array}{c|cccccccccc}
 & x_1 & x_2 & x_3 & x_4 & x_5 & x_6 & x_7 & x_8 & x_9 & x_{10} \\
\Gamma_1 & \begin{bmatrix} a & a \\ b & c \\ c & b \end{bmatrix} & & \begin{bmatrix} a \\ b \\ c \end{bmatrix} & b & \begin{bmatrix} a \\ c \end{bmatrix} & \begin{bmatrix} a \\ c \end{bmatrix} & * & \begin{bmatrix} b & a \\ b & c \\ c & b \end{bmatrix} & & \begin{bmatrix} a \\ c \\ a \end{bmatrix} \\
\Gamma_2 & \begin{bmatrix} a & b \\ b & a \\ b & c \\ c & b \end{bmatrix} & & \begin{bmatrix} a & b \\ a & a \\ b & b \\ c & a \end{bmatrix} & b & * & a & \begin{bmatrix} a & b \\ b & c \\ c & a \end{bmatrix} & & \\
\Gamma_3 & a & c & * & \begin{bmatrix} a & b \\ b & a \\ c & a \end{bmatrix} & * & b & \begin{bmatrix} a & a \\ b & b \\ c & c \end{bmatrix} & b & &
\end{array}
$$

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

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 $\gamma_{1_1} : \langle x_1, x_2, x_3 \rangle \in \{(a, b, a), (b, a, c), (c, b, b)\}$, $\gamma_{1_2} : x_4 = b$, $\gamma_{1_3} : \langle x_5, x_6 \rangle \in \{(a, a), (c, c)\}$, ... As an example of tuple represented by Γ_1, we find $(a, b, a, b, a, a, a, b, a, a)$. We can 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.*

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.

To conclude this section, do note that segmented tables are a generalization of both compressed and sliced tables. While a compressed or sliced table can be represented by a segmented table, the reverse is

not true. In particular, a compressed table constraint [18] can be seen as a particular segmented constraint where each table segment has arity 1 (whereas segmented tables allow us to use table segments of any arity). A sliced table [13] can be seen as a segmented table with exactly two segments (one for the pattern and one for the sub-table). Segemented tables can also be cast as logic programs in the 'Propria' system built over the CLP scheme [28].

4 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.

In Algorithm 1, Function filter() must be called (i.e., systematically triggered by the solving engine) every time 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.

```
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)
```

Algorithm 1: Synthetic Filtering Algorithm

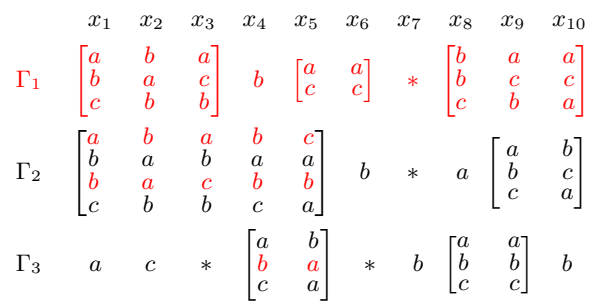

Figure 2: If b is removed from $\texttt{dom}(x_4)$, some parts of the segmented table become invalid (displayed in red color). We can then infer that $x_5 \neq c$.

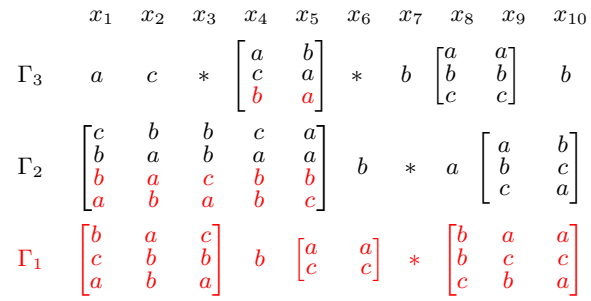

Figure 3: If b is removed from $\texttt{dom}(x_4)$, invalid parts of the segmented table are removed by swapping.

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, while considering the entire segmented table, let us suppose that b has been removed from $\texttt{dom}(x_4)$, instead of $\texttt{dom}(x_3)$. 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 $\texttt{dom}(x_5)$.

5 Detailed Description of the Algorithm

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 [7, 9], variants of Simple Tabular Reduction (STR) have been proved[2] to be quite competitive [30, 20, 21]. 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

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

```
Method enforceGAC()
    S^val ← {x ∈ scp : prevSizes[x] ≠ |dom(x)|}
    S^sup ← {x ∈ scp : |dom(x)| > 1}
    foreach x ∈ scp do
        gacValues[x] ← ∅
    traverseTable()
    if tableLimit = 0 then
        return FAILURE
    filterDomains()
    foreach variable x ∈ S^val ∪ S^sup do
        prevSizes[x] ← |dom(x)|
    return SUCCESS
Method traverseTable()
    i ← 1
    while i ≤ tableLimit do
        if isValidSegmentedTuple(Γ_i) then
            collectValues(Γ_i)
            i ← i + 1
        else                          // Γ_i must be removed
            swap Γ_i and Γ_tableLimit
            tableLimit ← tableLimit − 1
Method isValidSegmentedTuple(Γ)
    foreach x = a ∈ Γ^eq do
        if x ∈ S^val ∧ a ∉ dom(x) then
            return false
    foreach γ ∈ Γ^tb do
        S ← S^val ∩ scp(γ)
        if S = ∅ then
            continue
        i ← 1
        while i ≤ tableLimit[γ] do
            if isValidSubtuple(τ_i, S) then
                i ← i + 1
            else                      // τ_i must be removed
                swap τ_i and τ_tableLimit[γ]
                tableLimit[γ] ← tableLimit[γ] − 1
        if tableLimit[γ] = 0 then
            return false
    return true
Method isValidSubtuple(τ, S)
    foreach x ∈ S do
        if τ[x] ∉ dom(x) then
            return false
    return true
```

Algorithm 2: Class SegmentedConstraint

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.

```
Method collectValues(Γ)
    foreach x = ∗ ∈ Γ^tt do
        if x ∈ S^sup then
            remove x from S^sup
    foreach x = a ∈ Γ^eq do
        add a to gacValues[x]
        if |gacValues[x]| = |dom(x)| then
            remove x from S^sup
    foreach γ ∈ Γ^tb do
        S ← S^sup ∩ scp(γ)
        if S = ∅ then
            continue
        i ← 1
        while i ≤ tableLimit[γ] do
            foreach x ∈ S do
                if τ_i[x] ∉ gacValues[x] then
                    add τ_i[x] to gacValues[x]
                    if |gacValues[x]| = |dom(x)| then
                        remove x from S^sup
            i ← i + 1
Method filterDomains()
    foreach variable x ∈ S^sup do
        foreach value a ∈ dom(x) do
            if a ∉ gacValues[x] then
                remove a from dom(x)
```

Algorithm 3: Class SegmentedConstraint (continued)

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 $\texttt{tableLimit}[\gamma]$, 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 [20]. 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.

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() 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 segments. For an equality segment $x = a$, we just check if x must be tested according to $\mathtt{S}^{val}$ (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 $\mathtt{S}$ of variables occurring in both $\mathtt{S}^{val}$ and $\mathtt{scp}(\gamma)$. If ever this set $\mathtt{S}$ 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 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 $\mathtt{S}^{sup}$ 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 to be a support by adding a to $\mathtt{gacValues}[x]$, and we determine if x can be discarded from $\mathtt{S}^{sup}$ by comparing its size with that of $\mathtt{dom}(x)$. For a table segment, we start by computing the set $\mathtt{S}$ of variables occurring in both $\mathtt{S}^{sup}$ and $\mathtt{scp}(\gamma)$. If ever this set $\mathtt{S}$ 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 $\mathtt{S}$.

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 [17]. 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 of arity r is as follows. First, note that the space complexity of $\mathtt{scp}$, $\mathtt{S^{val}}$, $\mathtt{S^{sup}}$ and $\mathtt{prevSizes}$ is $O(r)$. 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 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 $O(rd)$, where d is the size of the greatest variable domain. 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.

Finally, even when some segmented tuples overlap, the algorithm that we propose remains correct because we only consider positive tables (i.e., tuples accepted by constraints are given) in this paper.

6 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 problem was used in the COP track of the 2018 XCSP3 Competition [6, 23]. The problem is stated as follows: given a grid order n and two dictionaries, a main dictionary (containing *main* words) and an auxiliary thematic dictionary (containing *thematic* words), 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, where no letter can be put). 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 and compare several models for Problem CD, using two main templates (actually, two ways of defining variables) denoted by a and b. Our aim is to compare various models so as to highlight the interest of segmented tables.

First Template. For the first template a, the variables are defined as follows:

- $x_{i,j}$, the letter put in the grid at the intersection of row i and column j, with $i \in 1..n$ and $j \in 1..n$. Possible letters are 'a', 'b', ..., 'z', and BP.
- br_i, the benefit obtained on row i according to the words formed by letters put in the ith row of x, with $i \in 1..n$. For example, if $n = 10$ and we have on row i a main word of size 3, followed by a BP and a thematic word of size 6, then $br_i = 0 + 6 = 6$.
- bc_j, the benefit obtained on column j according to the words formed by letters put in the jth column of x, with $j \in 1..n$.

The objective is to maximize $\sum_{i \in 1..n} br_i + \sum_{j \in 1..n} bc_j$. The three first models we propose for CD are called CD_a^{seg}, CD_a^{mdd}, and CD_a^{reg} and only involve $2 \times n$ 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. To build constraints, we reason from valid n-patterns, 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}. Now, for each valid pattern, we can build several segmented tuples when considering two possibilities for each word length in the pattern: taking a word of this length from either the main dictionary or the thematic dictionary. As an illustration, let us consider that $n = 4$ and we have twenty-six 1-letter words $\{a, b, \ldots, z\}$, 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 on each row involves 4 variables (since $n = 4$) and a table containing several segmented tuples built from the valid 4-patterns. This is illustrated in Figure 4 for the first row constraint, where the first pattern ('4') gives Γ_1 and Γ_2, the second pattern ('BP 3') gives Γ_3 and Γ_4 and so on. Of course, we proceed similarly with columns: there are n segmented table constraints for dealing with the n columns. Note that the constraint forbidding the presence of two adjacent BPs is directly taken into consideration by the segmented tables (tuples).

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. At this point, it is

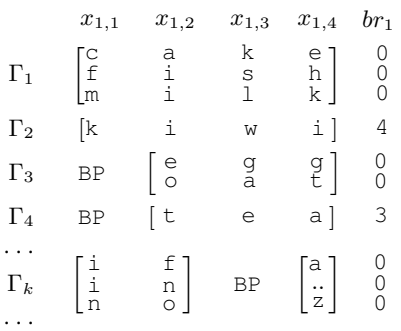

Figure 4: A Segmented Table for CD ($n = 4$).

important to note that using ordinary tables, instead of segmented tables, cannot be considered for 'large' values of n. If there are $20,000$ 7-letter words in the main dictionary, then the number of ordinary tuples generated from the pattern '7 BP 7' (when considering only words from the main dictionary) is $400,000,000$. Similarly, building an MDD while enumerating tuples is not viable. However, one can generate MDDs from segmented tables. One way to proceed is as follows. First, build an (ordered reduced) MDD from each segmented table. Second, merge all these MDDs by iteratively applying an efficient procedure [8, 27]. By replacing segmented constraints by MDD constraints (built this way), we then obtain a second model CD_a^{mdd}. Finally, if we directly consider the MDDs corresponding to the segmented tables (valid patterns) without any reduction (merging), we obtain a non-deterministic diagram for which a constraint `regular` can be used: this is model CD_a^{reg}. Although not tested in this paper, weighted automata could also be envisioned. However, we believe that required unfolding operations at propagation time might be too costly.

Second Template. For the second template b, 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.

For template b, the variables are defined as follows:

- $x_{i,j}$, defined similarly to template a.
- $br_{i,k}$, 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 $j \in 1..n \wedge k \in 1..m$.

The objective is to maximize the sum of variables $br_{i,k}$ and $bc_{j,k}$.

Let us first introduce the model CD_b^{seg}. For each row i, we have exactly one segmented constraint whose scope is $\{x_{i,j} : j \in 1..n\} \cup \{br_{i,k} : k \in 1..m\}$; the arity of such constraints is then $n + m$. The segmented table contains a segmented tuple for each valid n-pattern. To build a unique segmented table per valid n-pattern, we need to relax the order imposed by Definition 2 on (variables of) segments; this is mainly a form of technical subtlety. We proceed similarly with columns. This model CD_b^{seg} is more complex than CD_a^{seg} but has the advantage of reducing the size of the segmented tables.

It is also possible to consider 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_b^*, of the competition, involves the three 2-dimensional arrays of variables introduced above for CD_b^{seg}, and also some additional arrays. Due to lack of space, we do not provide further details about this model (with its short tables), but the interested reader can consult the model description in [23]. It is also possible to derive MDD constraints from short table constraints: each short table is transformed into an MDD (where some arcs can be labeled with '*'). We then obtain model CD_b^{mdd*}.

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 compared the six models described above. We have used a CP solver ($AbsCon$) that performs a classical backtrack search. For guiding search, we have used the classical heuristic `dom/wdeg` [5] and the value ordering heuristic `LastVal` that selects the last value in the domain of the selected variable. We have generated all CD instances for n ranging from 5 to 15. All experiments have been conducted on a dual socket Intel XEON X5550 quad-core 2.66 GHz with a RAM limit of 16GB.

n		seg_a	mdd_a	reg_a	seg_b	$\text{mdd}*_b$	$*_b$
5	bnd	***38***	*38*	*38*	*38*	*38*	*38*
	cpu	**4.9**	9.4	6.7	6.1	87	55
6	bnd	***54***	*54*	*54*	*54*	*54*	54
	cpu	**64.2**	296	159	196	461	8.2k
7	bnd	*70*	70	70	**68**	64	52
	cpu	9.5k	1.4k	1.5k	**447**	9.9k	8.3k
8	bnd	86	MO	**80**	75	76	TO
	cpu	3.6k	MO	**1.8k**	3.5k	4.9k	TO
9	bnd	91	MO	**91**	81	82	TO
	cpu	2.5k	MO	**1.6k**	1.5k	4.9k	TO
10	bnd	**110**	MO	90	92	70	TO
	cpu	9.8k	MO	7.8k	6.4k	6.8k	TO
11	bnd	**111**	MO	107	100	MO	TO
	cpu	**9.8k**	MO	2.6k	3.7k	MO	TO
12	bnd	120	MO	119	**115**	MO	TO
	cpu	6.0k	MO	3.2k	**9.5k**	MO	TO
13	bnd	113	MO	111	**138**	MO	TO
	cpu	5.4k	MO	5.6k	**4.6k**	MO	TO
14	bnd	143	MO	130	**160**	MO	TO
	cpu	7.5k	MO	9.4k	**4.1k**	MO	TO
15	bnd	156	MO	TO	**185**	MO	TO
	cpu	8.2k	MO	TO	**4.6k**	MO	TO

Table 1: Results obtained on Crosswords Design (CD) instances, for n ranging from 5 to 15. A backtrack search is run on five different models. Results are presented in terms of the best obtained bound (bnd) and the CPU time (k stands for kilo-seconds) to get it within $10,000$ seconds.

Table 1 shows the experimental results when the solver is given $10,000$ seconds to run. The best results are displayed in bold face. The best bound (bnd) found within the allowed time is indicated for each model, as well as the CPU times required to reach it. When optimality is proved, the bound is displayed in italic shape (and the CPU time indicates the total time). First, one can observe that building MDDs from segmented tables is only effective for small values of n, although we tested various orders (and chose the most rele-

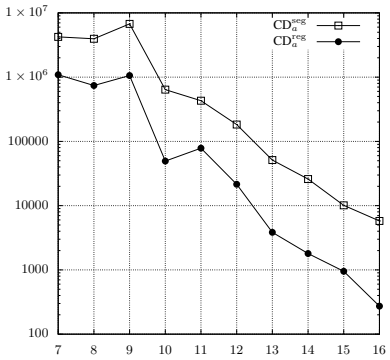

Figure 5: Number of wrong decisions (logarithmic y axis) taken in $10,000$ seconds by CD_a^{reg} and CD_a^{seg} against n (x axis).

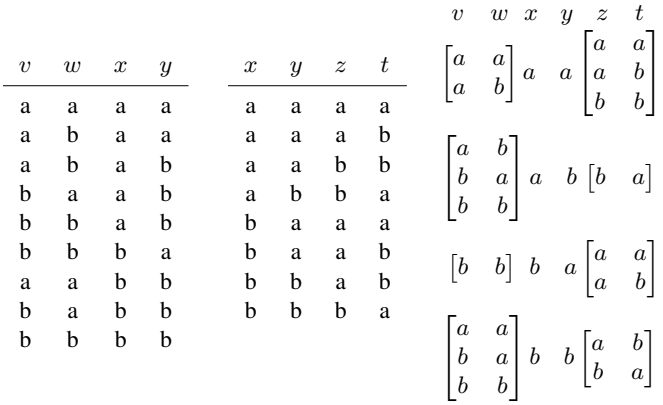

Figure 6: On the left, two ordinary table constraints sharing the variables (x, y). On the right, an equivalent table segmented constraint.

vant one) to apply merging operations: the size of the combined reduced MDDs was quite moderate, but from $n = 8$, the size of the generated intermediate diagrams provoked a memory-out (MO). The model CD_a^{reg} does not suffer from this drawback, and even obtains quite good results for intermediate values of n (8 and 9). However, for higher values ($n \geq 10$), CD_a^{seg} and CD_b^{seg} clearly outperform the other models. Besides, we found that the good results of CD_a^{reg} for some values of n are opportunistically due to the versatility of the heuristic (different search trees are built because of minor differences in constraint weighting). Indeed, we performed an experiment with the classical heuristic `dom` [15] that guarantees that CD_a^{reg} and CD_a^{seg} perform the very same search exploration. Figure 5 shows which parts of the search trees (measured by the number of wrong decisions [4]) are explored by CD_a^{reg} and CD_a^{seg} in $10,000$ seconds, with n ranging from 7 to 16. CD_a^{seg} is between 5 and 20 times faster than CD_a^{reg}; note that we use a logarithmic scale for the y axis.

We insist that Problem CD is very challenging, and cannot be tackled by classical (ordinary) table constraints. This is why the comparison is primarily performed against MDD/regular constraints. Comparison with other forms of compressed tables is further discussed now. On the one hand, using short tables is possible, and this comparison has been made (see Model CD_b^*, rightmost column, in Table 1, and also the results obtained by all solvers at the 2018 XCSP3 competition). The algorithm behind this model is STR2 [20] which is faster than STR3 [21] and CT [12] on this particular problem (CD); actually, it appears that CT is less efficient than STR2 on Problem CD due to the huge size of domains. On the other hand, in general, compressing dictionaries with sliced tables or compressed tuples is not very effective (for example, this is shown for sliced tables in Tables 3 and 4 in [13]). Concerning Problem CD, there is absolutely no clue about how to represent rows and columns with sliced or compressed tables without any memory explosion (similarly to ordinary tables).

7 Perspectives of Segmented Tables

We have just shown how segmented tables can be the right representation choice for a specific problem. Indeed, it is rather simple and natural to express constraints of Problem CD with segmented tables. And, this modeling approach has been shown to be quite efficient in practice. However, the reader must be aware that segmented tables are not appropriate for every problem where some table constraints are involved. And it is far from being obvious how to automatically convert ordinary tables into compressed segmented ones.

Nevertheless, we think that segmented tables are really a useful modeling tool with some promising perspectives. Firstly, for certain constraints, one may identify some relevant patterns for segmentation. For example, let us consider a global constraint `allDifferent` with scope $\langle x_1, \ldots, x_r \rangle$ and $dom(x_i) = \{1, \ldots, r\}$; a permutation is enforced. This constraint can be translated into a segmented table composed of exactly $\binom{r}{r/2}$ segmented tuples. Each segmented tuple is formed by a first sub-table (all permutations of the $r/2$ selected values) followed by a second sub-table (all permutations of the $r/2$ non selected values). This means that for $r = 10$, there are 252 segmented tuples, the size of each one being equivalent to 120 tuples of arity r. Compared to the $3,628,600$ ordinary tuples, these $30,240$ equivalent "tuples" are far less memory expensive. Contrary to the constraint `allDifferent`, imposing some additional restrictions on (some of) these r variables can be envisioned by transforming further the segmented table. Note that the same kind of segmentation can be performed on other constraints (e.g., `sum`).

Secondly, segmented tables allow us to merge easily constraints having non-trivial intersections (i.e., sharing at least two variables). For example, in Figure 6, we have on the left two 4-ary ordinary table constraints whose scopes share the variables x and y. Merging these two constraints gives the segmented table constraint depicted on the right; for each instantiation of x and y, we collect two sub-tables from the two constraints. This means that enforcing GAC on this constraint is equivalent to enforcing Pairwise consistency [16] on the original pair of constraints. This opens the door to an approach for enforcing pairwise consistency (totally or partially) without any additional structures [22] or variables [25].

8 Conclusion

We have introduced the concept of segmented tables that represent a very general form of expressing constraints. Indeed, they generalize compressed tables for which 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 algorithm. Interestingly, segmented tables could 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. Because of their segmented structure, developing efficient parallel filtering algorithms for segmented table constraints seems also a promising avenue. Finally, segmented tables are a new powerful modeling tool, as shown in the paper with a challenging problem, and some promising perspectives; one of them being related to pairwise consistency.

ACKNOWLEDGEMENTS

This work has been supported by the project CPER Data from the region "Hauts-de-France".

REFERENCES

[1] O. Akgun, I. Gent, C. Jefferson, I. Miguel, P. Nightingale, and A. Salamon, 'Automatic discovery and exploitation of promising subproblems for tabulation', in *Proceedings of CP'18*, (2018).
[2] K.R. Apt, *Principles of Constraint Programming*, Cambridge University Press, 2003.
[3] C. Bessiere, 'Arc consistency and arc consistency again', *Artificial Intelligence*, **65**, 179–190, (1994).
[4] C. Bessiere, B. Zanuttini, and C. Fernandez, 'Measuring search trees', in *Proceedings of ECAI'04 workshop on Modelling and Solving Problems with Constraints*, pp. 31–40, (2004).
[5] F. Boussemart, F. Hemery, C. Lecoutre, and L. Sais, 'Boosting systematic search by weighting constraints', in *Proceedings of ECAI'04*, pp. 146–150, (2004).
[6] F. Boussemart, C. Lecoutre, G. Audemard, and C. Piette, 'XCSP3: an integrated format for benchmarking combinatorial constrained problems', *CoRR*, **abs/1611.03398**, (2016).
[7] P. Briggs and L. Torczon, 'An efficient representation for sparse sets', *ACM Letters on Programming Languages and Systems*, **2**(1-4), 59–69, (1993).
[8] R. Bryant, 'Graph-based algorithms for Boolean function manipulation', *IEEE Transactions on Computers*, **35**(8), 677–691, (1986).
[9] V. Le Clément de Saint-Marcq, P. Schaus, C. Solnon, and C. Lecoutre, 'Sparse-sets for domain implementation', in *Proceeding of TRICS'13*, pp. 1–10, (2013).
[10] R. Dechter, *Constraint processing*, Morgan Kaufmann, 2003.
[11] J. Dekker, G. Bjordal, M. Carlsson, P. Flener, and J.-N. Monette, 'Auto-tabling for subproblem presolving in minizinc', *Constraints*, **22**(4), 512–529, (2017).
[12] J. Demeulenaere, R. Hartert, C. Lecoutre, G. Perez, L. Perron, J.-C. Régin, and P. Schaus, 'Compact-Table: efficiently filtering table constraints with reversible sparse bit-sets', in *Proceedings of CP'16*, pp. 207–223, (2016).
[13] N. Gharbi, F. Hemery, C. Lecoutre, and O. Roussel, 'Sliced table constraints: Combining compression and tabular reduction', in *Proceedings of CPAIOR'14*, pp. 120–135, (2014).
[14] T. Guns, A. Dries, S. Nijssen, G. Tack, and L. De Raedt, 'Miningzinc: A declarative framework for constraint-based mining', *Artificial Intelligence*, **244**, 6–29, (2017).
[15] R.M. Haralick and G.L. Elliott, 'Increasing tree search efficiency for constraint satisfaction problems', *Artificial Intelligence*, **14**, 263–313, (1980).
[16] P. Janssen, P. Jégou, B. Nouguier, and M.C. Vilarem, 'A filtering process for general constraint-satisfaction problems: achieving pairwise-consistency using an associated binary representation', in *Proceedings of IEEE Workshop on Tools for Artificial Intelligence*, pp. 420–427, (1989).
[17] C. Jefferson and P. Nightingale, 'Extending simple tabular reduction with short supports', in *Proceedings of IJCAI'13*, pp. 573–579, (2013).
[18] G. Katsirelos and T. Walsh, 'A compression algorithm for large arity extensional constraints', in *Proceedings of CP'07*, pp. 379–393, (2007).
[19] C. Lecoutre, *Constraint networks: techniques and algorithms*, ISTE/Wiley, 2009.
[20] C. Lecoutre, 'STR2: Optimized simple tabular reduction for table constraints', *Constraints*, **16**(4), 341–371, (2011).
[21] C. Lecoutre, C. Likitvivatanavong, and R. Yap, 'STR3: A path-optimal filtering algorithm for table constraints', *Artificial Intelligence*, **220**, 1–27, (2015).
[22] C. Lecoutre, A. Paparrizou, and K. Stergiou, 'Extending STR to a higher-order consistency', in *Proceedings of AAAI'13*, pp. 576–582, (2013).
[23] C. Lecoutre and O. Roussel, 'Proceedings of the 2018 XCSP3 competition', *CoRR*, **abs/1901.01830**, (2019).
[24] O. Lhomme, 'Arc-consistency filtering algorithms for logical combinations of constraints', in *Proceedings of CPAIOR'04*, pp. 209–224, (2004).
[25] C. Likitvivatanavong, W. Xia, and R. Yap, 'Decomposition of the factor encoding for CSPs', in *Proceedings of IJCAI'15*, pp. 353–359, (2015).
[26] J.-B. Mairy, Y. Deville, and C. Lecoutre, 'The smart table constraint', in *Proceedings of CPAIOR'15*, pp. 271–287, (2015).
[27] G. Perez and J.-C. Régin, 'Efficient operations on MDDs for building constraint programming models', in *Proceedings of IJCAI'15*, pp. 374–380, (2015).
[28] T. Le Provost and M. Wallace, 'Generalized constraint propagation over the CLP scheme', *Journal of Logic Programming*, **16**(3), 319–359, (1993).
[29] *Handbook of Constraint Programming*, eds., F. Rossi, P. van Beek, and T. Walsh, Elsevier, 2006.
[30] J. Ullmann, 'Partition search for non-binary constraint satisfaction', *Information Science*, **177**, 3639–3678, (2007).
[31] H. Verhaeghe, C. Lecoutre, Y. Deville, and P. Schaus, 'Extending Compact-Table to basic smart tables', in *Proceedings of CP'17*, pp. 297–307, (2017).
[32] H. Verhaeghe, C. Lecoutre, and P. Schaus, 'Extending Compact-Table to negative and short tables', in *Proceedings of AAAI'17*, pp. 3951–3957, (2017).
[33] R. Wang, W. Xia, R. Yap, and Z. Li, 'Optimizing Simple Tabular Reduction with a bitwise representation', in *Proceedings of IJCAI'16*, pp. 787–795, (2016).
[34] W. Xia and R. Yap, 'Optimizing STR algorithms with tuple compression', in *Proceedings of CP'13*, pp. 724–732, (2013).