Efficient Reification of Table Constraints

Minh Thanh Khong *, Yves Deville *, Pierre Schaus *, Christophe Lecoutre †
* ICTEAM, Université catholique de Louvain, Louvain-la-Neuve, Belgium
{minh.khong, yves.deville, pierre.schaus}@uclouvain.be
† CRIL-CNRS UMR 8188, Université d'Artois, F-62307 Lens, France
lecoutre@cril.fr

***Abstract*—Reifying a constraint c consists in associating a Boolean variable b with c such that c is satisfied if and only if b is true, which can be denoted by $c^{reif} : c \Leftrightarrow b$. Reification is useful for logically combining constraints and counting how many reified constraints can be satisfied. Since table constraints play an important role within constraint programming, in this paper, we are interested in their reification. We introduce a filtering algorithm that allows us to establish generalized arc consistency on reified table constraints, with no spatial overhead. We also propose a flexible approach that can generally reify any subsets of constraints. We show the practical interest of our work on the Max-CSP problem and a variation of the subgraph isomorphism problem.**

I. Introduction

The reification of a constraint c is used to reflect its truth value into a Boolean variable b. Consequently, reifying a constraint c involves replacing c with its reified form $c^{reif} : c \Leftrightarrow b$, with now the possibility of c being unsatisfied: b is true if and only if the constraint c is satisfied. Reified constraints are useful for applying logical connectives between constraints and/or expressing that a certain number of constraints must hold, e.g., by summing up the Boolean (interpreted as zero-one) variables associated with the reified constraints [1].

Table constraints, i.e., constraints defined by explicitly listing the allowed (or disallowed) combinations of values for the variables in their scopes, play an important role in constraint programming. Indeed, they can be seen as a general-purpose service, offered by constraint solvers, for expressing any kind of constraints, with the required space consumption as only limit to this approach. Table constraints can be useful to combine efficiently some parts of problems (e.g., merging highly related constraints), and appear naturally in many domains such as configuration and databases. Many algorithms have been proposed over the years to filter table constraints [2]–[9], or some of their compact forms [10]–[13].

In recent years, a number of works have been proposed for the reification of global constraints [1], [14], [15], not including table constraints. In this paper, we study the reification of table constraints, mainly by describing an algorithm to establish Generalized Arc Consistency (GAC), following the technique of Simple Tabular Reduction (STR) [5], [6]. One interesting outcome of our work is that the door is open to reify any kind of constraints, just by reformulating them as table constraints, provided that space memory consumption is not an issue.

We also propose a flexible approach to reify dynamically any subset of constraints as a table constraint when a certain threshold is reached. Reflecting the truth of this conjunction (subset) of constraints into a Boolean variable b is thus such that b is true if and only if all constraints in the subset are satisfied.

We introduce two applications of our work. First, we show how the Max-CSP problem (maximizing the number of satisfied constraints of a given Constraint Satisfaction Problem) can be solved efficiently when table constraints of high arity are involved. Second, we show how a variant of the Subgraph Isomorphism Problem (SIP) can be modeled easily, and solved efficiently, with our flexible reification approach.

II. Technical Background

A Constraint Satisfaction Problem (CSP) $P = (X, D, C)$ is composed of an ordered set of n variables, $X = \{x_1, \ldots, x_n\}$, a set of domains $D = \{dom(x_1), \ldots, dom(x_n)\}$ where $dom(x_i)$ is the set of possible values of the variable x_i and a set of e constraints, $C = \{c_1, \ldots, c_e\}$. Each constraint c involves an ordered set of variables, called the scope of c and denoted by $scp(c)$. Each constraint c is defined by a relation, denoted by $rel(c)$, which contains the allowed combinations of values for $scp(c)$. The arity of a constraint c is the size of $scp(c)$, and will usually be denoted by r.

Given a sequence $\langle x_1, \ldots, x_r \rangle$ of r variables, a r-tuple τ on this sequence of variables is a sequence of values $\langle a_1, \ldots, a_r \rangle$, where the individual value a_i is also denoted by $\tau[x_i]$ or, when there is no ambiguity, $\tau[i]$. Let c be an r-ary constraint. An r-tuple τ is *valid* on c iff $\forall x \in scp(c), \tau[x] \in dom(x)$, and τ is *allowed* by c iff $\tau \in rel(c)$ (we also say that c accepts τ). A *support* on c is a valid tuple on c that is also accepted by c. A constraint c is *generalized arc-consistent* (GAC) iff $\forall x \in scp(c)$, $\forall a \in dom(x)$, there exists a *support* of (x, a) on c, i.e., a valid tuple τ on c such that τ is accepted by c and $\tau[x] = a$. A solution to a CSP is the assignment of a value to each variable such that all the constraints are satisfied.

The set of valid tuples on a constraint c is $val(c) = \Pi_{x \in scp(c)} dom(x)$. The ordered set of variables involved in a set of constraints $\mathcal{C}$ is denoted by $vars(\mathcal{C})$; we have $vars(\mathcal{C}) = \cup_{c \in \mathcal{C}} scp(c)$. The set of valid tuples on a set of constraints $\mathcal{C}$ is $val(\mathcal{C}) = \Pi_{x \in vars(\mathcal{C})} dom(x)$. A constraint c is said to be *entailed* (resp. *disentailed*) if any tuple τ in $val(c)$ is accepted (resp., not accepted) by c; in other words, c is always satisfied (resp. violated). A positive (resp. negative)

table constraint is a constraint whose semantics is defined in extension by listing the set of *allowed* (resp. *forbidden*) tuples. This table (set) is denoted by $table^{init}(c)$.

The reification of a constraint c is obtained by associating a Boolean variable b with c. We then obtain a reified constraint $c^{reif} : c \Leftrightarrow b$. The operational semantics of a filtering algorithm (propagator) for the reification of such a constraint is given by the following rules:

- if b is set to 1, then c must hold.
- if b is set to 0, then the negation of c must hold.
- if c becomes entailed, then b is set to 1.
- if c becomes disentailed, then b is set to 0.

To deal with a reified constraint, we need a propagator for c, a propagator for $\neg c$, and we have to detect when c becomes entailed or disentailed. The observation below shows when a table constraint becomes entailed and disentailed.

Observation 1. *Let $table(c)$ be the set of current supports of c, i.e., we have $table(c) = table^{init}(c) \cap val(c)$. We have:*

- *c is entailed iff $|table(c)| = |val(c)|$,*
- *c is disentailed iff $|table(c)| = 0$.*

When a table constraint becomes *entailed*, all valid tuples on c are supports of c. Example 1 shows an entailed constraint.

Example 1. *Given a positive table constraint c such that $scp(c) = \{x_1, x_2, x_3\}$ and $table(c)$ is:*

x_1	x_2	x_3
0	*0*	*0*
0	*1*	*0*
1	*0*	*0*
1	*1*	*0*

If $dom(x_1) = dom(x_2) = \{0, 1\}$ and $dom(x_3) = \{0\}$, then c is entailed because $|table(c)| = 4 = |val(c)| = |dom(x_1) \times dom(x_2) \times dom(x_3)|$.

III. Reifying Stand-alone Table Constraints

In this section, we present an algorithm for enforcing GAC on a given reified table constraint. The principle is to update the table of the constraint at each call of the filtering algorithm, so as to remove the tuples that have become invalid, and then to check the possibility of entailment or disentailment. For managing the table, we use the well-known technique called STR (Simple Tabular Reduction) [5], [6], [8].

A. Data Structures

The table associated with a table constraint c is denoted by $c.table$. This table is represented by an array of tuples indexed from 1 to $c.table.length$ that denotes the size of the table (i.e., the number of tuples). If the table is positive (resp., negative), $c.positive$ is true (resp., false). The worst-case space complexity to represent a table constraint c is $O(rt)$ where $t = c.table.length$ and $r = |scp(c)|$.

STR-based algorithms have been developed for filtering table constraints. The latest developments combining the principle of STR with bit vectors have been shown to be state-of-the-art [9], [16]. For simplicity, we only present the algorithm in the spirit of STR1 [5], i.e., without any optimizations. STR-based algorithms are efficient, in particular, because their data structures permit a cheap restoration upon backtracking. The principle of STR is to split a table into different sets such that each tuple is member of exactly one set ; this corresponds to the use of a sparse set [17], [18]. One of these sets contains all tuples that are currently valid: tuples in this set constitute the content of the *current table*, and are the current valid tuples of the constraint. Other sets contain tuples removed at different levels of search. For simplicity, data structures related to backtracking are not detailed in this work (see [6]).

The following arrays provide access to the disjoint sets of valid and invalid tuples within $c.table$:

- $c.position$ is an array of size $t = c.table.length$ that provides indirect access to the tuples of $c.table$. At any given time, the values in $c.position$ are a permutation of $\{1, 2, ..., t\}$. The i^{th} tuple of c is $c.table[c.position[i]]$. For simplicity, this tuple is denoted by $\tau_{c,i}$.
- $c.limit$ is the position of the last current tuple in $c.table$. The current table of c is composed of exactly $c.limit$ tuples. The values in $c.position$ at indices ranging from 1 to $c.limit$ are positions of the current tuples of c.

B. Tab-Reif

We now describe Tab-Reif, an algorithm for enforcing GAC on a given reified table constraint $c^{reif} : c \Leftrightarrow b$. Note that it can applied whatever the table constraint is positive or negative.

Algorithm 1: Tab-Reif($c^{reif} : c \Leftrightarrow b$)

```
1  if dom(b) = {1} then
2      replace c^reif by c                          // we post c
3  else if dom(b) = {0} then
4      replace c^reif by ¬c                         // we post ¬c
5  else if dom(b) = {0,1} then
6      i ← 1
7      while i ≤ c.limit do
8          if ∀x ∈ scp(c), τ_{c,i}[x] ∈ dom(x) then
9              i ← i + 1
10         else
11             swap tuples τ_{c,i} and τ_{c,c.limit}
12             c.limit ← c.limit − 1
       // Check entailment/disentailment
13     if c.limit = 0 then
14         dom(b) ← c.positive ? {0} : {1}
15         discard c^reif
16     else if c.limit = |val(c)| then
17         dom(b) ← c.positive ? {1} : {0}
18         discard c^reif
```

The first operation is to test whether b is assigned or not. If it is the case, we need to post the propagator for ensuring c or $\neg c$, and there will be no further call to the propagator of c^{reif}: c^{reif} is replaced by c or $\neg c$ (but c^{reif} will be restored when

backtracking). Note that any filtering can be employed such as STR2 [6] or CT [9] for positive table constraints, and STR-Ni [19] or CT$_{neg}$ [20] for negative table constraints. If b is not assigned, we have to check for entailment or disentailment of c (lines 13-18), after having updated the current table (lines 6-12). When entailment or disentailment is proved, b can be assigned and the reified constraint c^{reif} can be discarded (but will be restored when backtracking).

Proposition 1. *Tab-Reif enforces GAC on any given reified table constraint.*

Proof. When b is assigned, Tab-Reif ensures GAC on constraint c if b is 1 or $\neg c$ if b is 0 (assuming the propagators for c and $\neg c$ enforce GAC). Otherwise, Tab-Reif guarantees to filter $dom(b)$ when it is possible, according to Observation 1. □

The worst-case time complexity of Tab-Reif is $O(rt)$, and can even be decreased by using some optimizations proposed for STR2 or CT. Of course, when c^{reif} has been replaced, the worst-case time complexity is that of the employed propagators (e.g., CT and CT$_{neg}$).

IV. Reifying Subsets of Constraints

We propose now to reify dynamically any subset of constraints (not necessarily, table constraints). More specifically, we introduce an algorithm that can be used to reify any non-empty subset $\mathcal{C}$ of constraints of a given CSP $P = (X, D, C)$; $\mathcal{C} \subseteq C$ is called a *sub-model* of P. The CSP to be solved is then $\mathcal{P} = (X, D, C \setminus \mathcal{C} \cup \{\mathcal{C}^{reif}\})$ where $\mathcal{C}^{reif}$ is the reified submodel of P: we have $\mathcal{C}^{reif} : \mathcal{C} \Leftrightarrow b$. We propose to reformulate dynamically the reified sub-model $\mathcal{C}^{reif}$ as a reified table constraint c^{reif} when the size of the Cartesian product of the domains of the variables involved in $\mathcal{C}$ is less than a given threshold L (in order to avoid combinatorial explosion). We chose this metric for the threshold as it is an upper bound on the number of tuples of the generated reified table constraint. This upper bound can be improved by estimating the number of solutions for a subset of constraints [21], but this is out of scope of this paper.

Algorithm 2 is the algorithm we propose for filtering the reified sub-model $\mathcal{C}^{reif} : \mathcal{C} \Leftrightarrow b$. Note that this algorithm does not guarantee GAC, notably because filtering is delayed until the number of valid tuples corresponding to the variables involved in $\mathcal{C}$ is less than the specified integer limit L. If the variable b is assigned to 1, we can simply replace $\mathcal{C}^{reif}$ by all constraints in C (an alternative, not considered in this paper, is to only keep lines 3-16 of Algorithm 2). Otherwise, when the test at Line 3 evaluates to true, a table T is built (lines 4–7). To perform this operation, the current state of $\mathcal{P}$ is stored, the constraint $\mathcal{C}^{reif}$ is discarded and all constraints in $\mathcal{C}$ added to $\mathcal{P}$. Then, all tuples in $val(\mathcal{C})$ that are compatible with $\mathcal{P}$ while considering constraint propagation ϕ are collected. More precisely, for each tuple $\tau \in val(\mathcal{C})$, $\mathcal{P}|_\tau$ denotes the CSP $\mathcal{P}$ with the additional variable assignments $x = \tau[x], \forall x \in vars(\mathcal{C})$. The test $\phi(\mathcal{P}|_\tau) \neq \bot$ indicates if running constraint propagation (denoted by ϕ) on $\mathcal{P}|_\tau$ does not lead to a conflict (domain wipeout denoted by $\bot$). At Line 7, the state of $\mathcal{P}$ is restored. If entailment or disentailment of $\mathcal{C}$ is proved (lines 8 and 11), b can be assigned and the reified sub-model $\mathcal{C}^{reif}$ can be discarded (but will be restored when backtracking). Otherwise, we can build a positive table constraint $vars(\mathcal{C}) \in T$ and post its reified form. It is important to note that the table T can be obtained by branching on the variables in $vars(\mathcal{C})$ and using propagation at each step. Also, we assume a trailed-based solver able to undo the changes on the state of the CSP between the `store` and `restore` instructions.

Algorithm 2: Mod-Reif(L: Integer, $\mathcal{C}^{reif} : \mathcal{C} \Leftrightarrow b$)

```
1  if dom(b) = {1} then
2      replace C^reif by C   // we post const. in C
3  else if |val(C)| < L then
4      store state P
5      replace C^reif in P by C
6      T ← {τ ∈ val(C) : φ(P|_τ) ≠ ⊥}
7      restore state P          // P is as in Line 2
8      if |T| = 0 then
9          dom(b) ← {0}
10         discard C^reif
11     else if |T| = |val(C)| then
12         dom(b) ← {1}
13         discard C^reif
14     else
15         let c^reif : vars(C) ∈ T ⇔ b
16         replace C^reif by c^reif        // we post c^reif
```

The way a table constraint is constructed in Mod-Reif has some similarities with works about solving submodels on the fly [22] and autotabling [23]. Sub-models are converted into table constraints to speed-up the solving process. However, in our case, we are interested in reification.

V. Applications

We show the practical interest of our approach on two applications: Max-CSP and a SIP variant. We implemented the reification algorithms in OscaR [24], a constraint solver written in Scala.

A. Max-CSP

When a CSP is unsatisfiable (i.e., has no solution), it may be interesting to identify a complete instantiation that satisfies the greatest number of constraints. This is called the Maximal Constraint Satisfaction Problem (Max-CSP). During the two last decades, many works have been carried out to solve this problem (and its direct extension, WCSP); see e.g., [25]–[28].

Suppose that P is an unsatisfiable instance, and that we want to solve its Max-CSP problem version. One simple approach is to reify each constraint c_i of P (with the introduction of a Boolean variable b_i) and to convert the CSP into a COP (Constraint Optimization Problem) whose objective function is $max \sum_{i \in 1..e} b_i$. If P only contains table constraints, we can use our algorithm Tab-Reif. This is what we have made

with the series of n-ary table constraints (with $n > 2$) used at the Max-CSP 2008 competition[1].

On a cluster under Linux (CPUs clocked at 2.2 GHz, with 10GB of RAM), we have compared on the instances of these series the behavior of Tab-Reif (implemented in OscaR), and Toulbar2 version 9.8.0 (http://www7.inra.fr/mia/T/toulbar2/documentation.html), which is a well-known state-of-the-art solver dedicated to Max-CSP and WCSP. We have also considered the results obtained by the three best solvers (AbsCon, sugar++, and tbBTD) at the Max-CSP 2008 competition (times have been taken as such even if must be cautiously considered as the execution environment is clearly not the same). The timeout was set to $1,200$ seconds.

TABLE I
NUMBER OF SOLVED INSTANCES PER SERIES.

Series	#ins	Tab-Reif	toulbar2	AbsCon	sugar++	tbBTD
aim-50	8	8	8	8	8	8
aim-100	8	8	8	8	8	8
aim-200	8	8	8	6	8	6
bddLarge	20	0	0	2	0	0
bddSmall	15	0	15	15	0	15
dubois	13	6	11	2	13	13
pret	8	4	8	4	8	8
rand-10-20-10	20	20	0	0	0	0
rand-3-20-20	10	10	9	7	6	9
lexVg	15	4	0	0	2	0
wordsVg	15	2	0	0	1	0
modifiedRenault	17	13	0	5	17	17
jnhUnsat	15	12	15	9	15	13

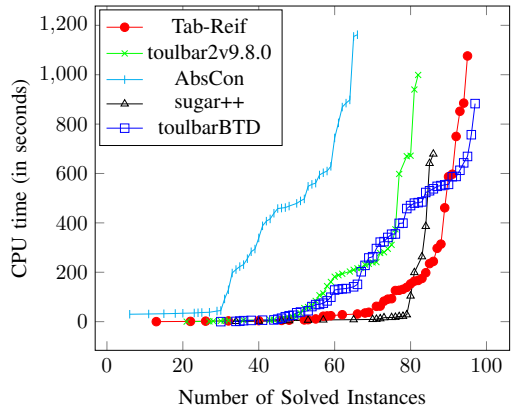

Fig. 1. Number of instances solved in a given amount of time.

Table I shows the number of instances (per series) solved by each of the five solvers. On some series, Tab-Reif is largely outperformed (notably, on bddLarge and bddSmall) while on some other series Tab-Reif is dominating (notably, on rand-10-20-10 and lexVg). Figure 1 is a cactus plot indicating the number of solved instances according to elapsed time. Tab-Reif has a rather good behavior, being dominated by toulbarBTD when 600 seconds have been reached. This experimentation aims at showing that reified table constraints can be a useful and efficient mechanism to solve MAX-CSP instances in certain circumstances.

[1]See https://www.cril.univ-artois.fr/CPAI08/results/results.php?idev=16

B. Variant of Subgraph Isomorphism Problem

1) Description: A *graph* $G = (N, E)$ consists of a node set N and an edge set $E \subseteq N \times N$ where an edge (u, v) is a pair of nodes. In this paper, we consider only undirected graphs, $(u, v) \in E \Rightarrow (v, u) \in E$. A graph $G_p = (N_p, E_p)$ is *isomorphic* to a graph $G_t = (N_t, E_t)$ if there exists a mapping $f : N_p \rightarrow N_t$ such that $(u, v) \in E_p \Leftrightarrow (f(u), f(v)) \in E_t$.

A *subgraph isomorphism problem* (SIP) between a pattern graph G_p and a target graph G_t consists in deciding whether G_p is isomorphic to some subgraph of G_t. We propose an extension of SIP (so-called eSIP) which handles pattern graphs containing two sub-patterns where only one must be part of the isomorphism. For a pattern graph $G_p = (N_p, E_p \cup E_1 \cup E_2)$ and a target graph $G_t = (N_t, E_t)$, eSIP consists in deciding whether there exists a subgraph of G_t that is isomorphic to either $(N_p, E_p \cup E_1)$ or $(N_p, E_p \cup E_2)$. An example is given by Figure 2.

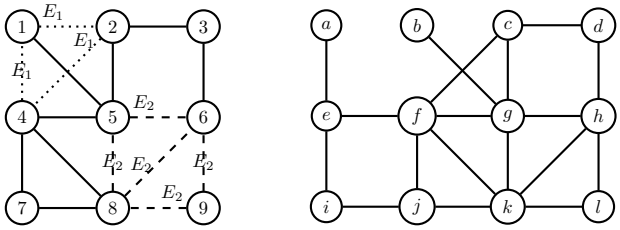

Fig. 2. An eSIP instance (pattern graph on the left, target graph on the right). Edges of E_1 (E_2) are dotted (dashed) line . A possible solution is $f = \{(1, b), (2, c), (3, d), (4, f), (5, g), (6, h), (7, j), (8, k), (9, l)\}$.

An eSIP can be formulated as a CSP as follows. A variable x_u is associated with every node u of the pattern graph with $dom(x_u) = N_t$. A global constraint *AllDifferent* is used to ensure that the matching is injective, and a set of binary constraints for edge matching: $\forall (u, v) \in E_p, (x_u, x_v) \in E_t$. We introduce two reified sub-models $\mathcal{C}_i^{reif} : \mathcal{C}_i \Leftrightarrow b_i$ for E_i s.t. $\mathcal{C}_i = \forall (u_i, v_i) \in E_i, (x_{u_i}, x_{v_i}) \in E_t, i = 1, 2$, then the constraint $b_1 \vee b_2$ ensures that $\mathcal{C}_1$ or $\mathcal{C}_2$ must be satisfied. This CSP can be solved straightforwardly by our flexible reification approach Mod-Reif.

2) Experimental Results: We ran our experiment on a Mac OS X with a 2.70GHz Intel Core i5 and 16GB of memory. To evaluate our algorithm, some classes of instances were chosen from the *vflib* database (see [29], [30] for more details). Each class *bvg-x-p-t* contains 10 instances with fixed-valence graphs where $x \in \{6, 9\}$ corresponds to the valence, $p \in \{40, 80\}$ corresponds to the number of nodes in the pattern graphs and $t \in \{200\}$ corresponds to the number of nodes in the target graphs. Each sub-model is generated randomly by extracting 20% of nodes and 90% of edges from the pattern graphs (these edges are removed in the pattern graph). In order to compare different approaches, a static search heuristics (lexico) is used.

We have compared different levels of reification: *static* means that reified sub-models are reformulated at the root of the search tree, while *dynamic* means that the reformulation is postponed in the search tree until the threshold L is reached. Different thresholds have been considered in our

TABLE II
AVERAGE RESOLUTION TIME (S) FOR STATIC AND DYNAMIC REIFICATION.

Class	Static	Dynamic			
		L1	L2	L3	check
bvg6-40-200	7.49	6.09	5.60	5.56	7.04
bvg9-40-200	27.75	9.19	9.05	21.31	27.84
bvg6-80-200	28.06	15.52	14.07	33.00	51.45

TABLE III
AVERAGE NUMBER OF FAILS FOR STATIC AND DYNAMIC REIFICATION.

Class		Static	Dynamic			
			L1	L2	L3	check
bvg6-40-200	#fail	12858	13543	13661	14497	16542
bvg9-40-200	#fail	9203	12228	14108	36557	72949
bvg6-80-200	#fail	6562	18112	16826	41887	65453

TABLE IV
AVERAGE NUMBER OF REFORMULATION CALLS AND TABLE SIZE FOR STATIC AND DYNAMIC REIFICATION.

Class		Static	Dynamic			
			L1	L2	L3	check
bvg6-40-200	#call	2	1108.1	1654.1	1628.9	2080.0
	size	360.0	1.10	0.73	0.67	0.64
bvg9-40-200	#call	2	1358.6	1371.6	50992	104520
	size	320.0	0.53	0.53	0.23	0.07
bvg6-80-200	#call	2	440.0	600.0	4422.4	16480
	size	460.0	0.00	0.00	0.00	0.00

experimentation: $L1 = 1,000,000$, $L2 = 10,000$, $L3 = 100$ and $check = 1$.

Table II shows that the dynamic approach usually provides better resolution times than the static approach. This is mainly due to the time taken to reformulate reified sub-models at the root node (when domains have not been reduced at all). Table III shows that as soon as the reformulation is performed, it can provide a better pruning. Hence, the number of fails increases when the threshold decreases. This can also make higher the number of reformulation calls (see Table IV). The thresholds *L1* and *L2* are good trade-offs.

VI. CONCLUSION

In this paper, we have presented a GAC algorithm for reified table constraints, which does not require any additional space: we keep dealing with the original tables of the constraints. We have also introduced a flexible reification approach for reifying any subsets of constraints, by generating dynamically reified table constraints. The practical interest of these algorithms have been shown on two problems. Interestingly, our algorithms benefit from recent algorithmic advances such as those proposed in Compact-Table.

ACKNOWLEDGMENTS

The first author is supported by the FRIA-FNRS (Fonds pour la Formation à la Recherche dans l'Industrie et dans l'Agriculture, Belgium). The fourth author is supported by the project CPER DATA from the "Hauts-de-France".

REFERENCES

[1] F. Fages and S. Soliman, "Reifying global constraints," HAL, Tech. Rep. RR-8084, 2012.

[2] C. Bessiere and J.-C. Régin, "Arc consistency for general constraint networks: preliminary results," in *Proceedings of IJCAI'97*, 1997, pp. 398–404.

[3] O. Lhomme and J.-C. Régin, "A fast arc consistency algorithm for n-ary constraints," in *Proceedings of AAAI'05*, 2005, pp. 405–410.

[4] I. Gent, C. Jefferson, I. Miguel, and P. Nightingale, "Data structures for generalised arc consistency for extensional constraints," in *Proceedings of AAAI'07*, 2007, pp. 191–197.

[5] J. Ullmann, "Partition search for non-binary constraint satisfaction," *Information Science*, vol. 177, pp. 3639–3678, 2007.

[6] C. Lecoutre, "STR2: Optimized simple tabular reduction for table constraints," *Constraints*, vol. 16, no. 4, pp. 341–371, 2011.

[7] J.-B. Mairy, P. Van Hentenryck, and Y. Deville, "Optimal and efficient filtering algorithms for table constraints," *Constraints*, vol. 19, no. 1, pp. 77–120, 2014.

[8] C. Lecoutre, C. Likitvivatanavong, and R. Yap, "STR3: A path-optimal filtering algorithm for table constraints," *Artificial Intelligence*, vol. 220, pp. 1–27, 2015.

[9] J. Demeulenaere, R. Hartert, C. Lecoutre, G. Perez, L. Perron, J.-C. Régin, and P. Schaus, "Efficiently filtering table constraints with reversible sparse bit-sets," in *Proceedings of CP'16*, 2016, pp. 207–223.

[10] G. Katsirelos and T. Walsh, "A compression algorithm for large arity extensional constraints," in *Proceedings of CP'07*, 2007, pp. 379–393.

[11] K. Cheng and R. Yap, "An MDD-based generalized arc consistency algorithm for positive and negative table constraints and some global constraints," *Constraints*, vol. 15, no. 2, pp. 265–304, 2010.

[12] N. Gharbi, F. Hemery, C. Lecoutre, and O. Roussel, "Sliced table constraints: Combining compression and tabular reduction," in *Proceedings of CPAIOR'14*, 2014, pp. 120–135.

[13] G. Perez and J.-C. Régin, "Improving GAC-4 for Table and MDD constraints," in *Proceedings of CP'14*, 2014, pp. 606–621.

[14] T. Feydy, Z. Somogyi, and P. Stuckey, "Half reification and flattening," in *Proceedings of CP'11*, 2011, pp. 286–301.

[15] N. Beldiceanu, M. Carlsson, P. Flener, and J. Pearson, "On the reification of global constraints," *Constraints*, vol. 18, no. 1, pp. 1–6, 2013.

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

[17] P. Briggs and L. Torczon, "An efficient representation for sparse sets," *ACM Letters on Programming Languages and Systems*, vol. 2, no. 1-4, pp. 59–69, 1993.

[18] V. le Clément de Saint-Marcq, P. Schaus, C. Solnon, and C. Lecoutre, "Sparse-sets for domain implementation," in *Proceeding of TRICS'13*, 2013, pp. 1–10.

[19] H. Li, Y. Liang, J. Guo, and Z. Li, "Making simple tabular reduction works on negative table constraints," in *Proceedings of AAAI'13*, 2013, pp. 1629–1630.

[20] H. Verhaeghe, C. Lecoutre, and P. Schaus, "Extending Compact-Table to negative and short tables," in *Proceedings of AAAI'17*, 2017, pp. 3951–3957.

[21] A. Favier, S. De Givry, and P. Jégou, "Exploiting problem structure for solution counting," in *International Conference on Principles and Practice of Constraint Programming*. Springer, 2009, pp. 335–343.

[22] C. Bessiere and J.-C. Régin, "Enforcing arc consistency on global constraints by solving subproblems on the fly," in *Proceedings of CP'99*, 1999, pp. 103–117.

[23] J. Dekker, G. Bjordal, M. Carlsson, P. Flener, and J.-N. Monette, "Auto-tabling for subproblem presolving in MiniZinc," *Constraints*, 2017.

[24] OscaR Team, "OscaR: Scala in OR," 2012, available from https://bitbucket.org/oscarlib/oscar.

[25] E. Freuder and R. Wallace, "Partial constraint satisfaction," *Artificial Intelligence*, vol. 58, no. 1-3, pp. 21–70, 1992.

[26] J. Larrosa and P. Meseguer, "Partition-Based lower bound for Max-CSP," *Constraints*, vol. 7, pp. 407–419, 2002.

[27] M. Cooper, S. de Givry, and T. Schiex, "Optimal soft arc consistency," in *Proceedings of IJCAI'07*, 2007, pp. 68–73.

[28] C. Lecoutre, N. Paris, O. Roussel, and S. Tabary, "Solving WCSP by extraction of minimal unsatisfiable cores," in *Proceedings of ICTAI'13*, 2013, pp. 915–922.

[29] M. D. Santo, P. Foggiae, C. Sansone, and M. Vento, "A large database of graphs and its use for benchmarking graph isomorphism algorithms," *Pattern Recognition Letters*, vol. 24, no. 8, pp. 1067–1079, 2003.

[30] C. Solnon, "Alldifferent-based filtering for subgraph isomorphism," *Artificial Intelligence*, vol. 174, no. 12-13, pp. 850–864, 2010.