Extending Compact-Table to Basic Smart Tables

Hélène Verhaeghe[1], Christophe Lecoutre[2], Yves Deville[1], and Pierre Schaus[1]

[1]UCLouvain, ICTEAM, Place Sainte Barbe 2, 1348 Louvain-la-Neuve, Belgium,
$\{firstname.lastname\}@uclouvain.be$
[2]CRIL-CNRS UMR 8188, Université d'Artois, F-62307 Lens, France, $lecoutre@cril.fr$

Abstract. Table constraints are instrumental in modelling combinatorial problems with Constraint Programming. Recently, Compact-Table (CT) has been proposed and shown to be as an efficient filtering algorithm for table constraints, notably because of bitwise operations. CT has already been extended to handle non-ordinary tables, namely, short tables and/or negative tables. In this paper, we introduce another extension so as to deal with basic smart tables, which are tables containing universal values ($*$), restrictions on values ($\neq v$) bounds ($\leq v$ or $\geq v$) and sets ($\in S$). Such tables offer the user a better expressiveness and permit to deal efficiently with compressed tuples. Our experiments show a substantial speedup when compression is possible (and a very limited overhead otherwise).

Keywords: Table Constraints, Filtering, Compression, Compact-Table, Bitset

1 Introduction

Table constraints, also known as extension(al) constraints, express on sequences of variables the combinations of values that are allowed (*supports*) or forbidden (*conflicts*). Lots of efforts [1, 5, 7, 9, 13, 16, 17, 19, 24, 28, 30] have been made during the last decade to enhance the filtering process of such constraints, in order to establish the property known as Generalized Arc Consistency (GAC). Motivation behind this excitement comes from the fact that tables can theoretically encode any other kind of constraints. They are thus used in many application fields, as stated in the industry.

The last big improvement in this domain has been the introduction of Compact-Table [5], an algorithm that advantageously combines tabular reduction and bitwise operations (a related algorithm, independently proposed in the literature, is STRBit [30]). Quite interestingly, Compact-Table (CT) has been shown to be about one order of magnitude faster than the best algorithm(s) developed during the last decade.

Unfortunately, tables have a major drawback: the memory space required to store them, which may grow exponentially with the number of columns (arity). To address this issue, various compression techniques have already been studied. Some are based on using particular data structures, like Multi-valued Decision Diagrams (MDDs) [4, 24], tries [7] and Deterministic Finite Automata (DFA) [25]. Other approaches attempt to keep a table-like structure, which is made compact by reasoning on Cartesian products and some intentional forms of column restrictions, like short tuples [10], compressed tuples [11, 31], sliced tables [8] and smart tables [21].

In this paper, we show how CT can be extended for basic smart tables, i.e., tables that may contain universal values ($*$), restrictions ($\neq$), bounds ($\leq$ and $\geq$) and sets ($\in S$).

2 Technical Background

A *constraint network* (CN) is composed of a set of n variables and a set of e constraints. Each *variable* x has an associated domain ($dom(x)$) that contains the finite set of values that can be assigned to it. Each *constraint* c involves an ordered set of variables, called the *scope* of c and denoted by $scp(c)$, and is semantically defined by a *relation* ($rel(c)$) which contains the set of tuples allowed for the variables involved in c. The *arity* of a constraint c is $|scp(c)|$. For simplicity, a variable-value pair (x, a) such that $x \in scp(c)$ and $a \in dom(x)$ is called a *value* (of c). A *table constraint* c is a constraint such that $rel(c)$ is explicitly defined by listing (in a table) the tuples that are allowed[1] by c.

Let $\tau = (a_1, \ldots, a_r)$ be a tuple of values associated with an ordered set of variables $vars(\tau) = \{x_1, \ldots, x_r\}$. The ith value of τ is denoted by $\tau[i]$ or $\tau[x_i]$, and τ is *valid* iff $\forall i \in 1..r, \tau[i] \in dom(x_i)$. τ is a *support* on a constraint c iff $vars(\tau) = scp(c)$ and τ is a valid tuple allowed by c. If τ is a support on a constraint c involving a variable x and such that $\tau[x] = a$, we say that τ is a *support for* the value (x, a) of c. Enforcing Generalized Arc Consistency (GAC) on a constraint c means removing all values without any support on c.

Over the years, there have been many developments about compact forms of tables. Ordinary tables contain *ordinary* or *ground* 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[2] are composed of *smart* tuples, which are tuples containing expressions (column constraints) of one the following forms: $*$, <op> v, $\in S$, $\notin S$, <op> x_j and <op> $x_j + v$; v being a value, S a set of values, and <op> an operator in $\{<, \leq, =, \neq, \geq, >\}$. Finally, a *basic* smart table is a restricted form of smart table where column constraints are unary, that is where smart tuples are of the form $*$, <op> v, $\in S$ and $\notin S$. For example, in the smart tuple $(\neq 1, 2, > 1)$, x_1 must be different from 1, x_2 must be equal to 2 ('= 2' being trivially simplified in '2') and x_3 must be greater than 1. In term of expressiveness, one can observe that basic smart tables are equivalent to compressed tables, meaning that any compressed tuple can be represented by a basic smart tuple (this is immediate), and any basic smart tuple can be represented by a compressed tuple. However, they allow more compact representation (having '$\neq 2$' is shorter than '$\{\ldots, 1, 3, 4, 5, \ldots\}$').

3 CT on Ordinary and Short Tables

This section briefly introduces Compact-Table (CT), a state-of-the-art filtering algorithm [5] initially introduced for enforcing GAC on positive (ordinary) table constraints. It first appeared in Or-Tools [22], the solver developed at Google, and is now implemented in OscaR [23], AbsCon and Choco [26]. CT benefits from well-established

[1] We only deal with positive forms of table constraints in this paper.

[2] For simplicity, we consider here a slightly simpler form of smart table constraints than in [21].

	x	y	z
τ_1	$\neq a$	$*$	c
τ_2	c	$\leq b$	$\neq a$
τ_3	$< c$	b	$\neq b$
τ_4	$> b$	$\geq b$	$*$

(a) Table

	supports				supports*				supportsMin				supportsMax			
	τ_1	τ_2	τ_3	τ_4	τ_1	τ_2	τ_3	τ_4	τ_1	τ_2	τ_3	τ_4	τ_1	τ_2	τ_3	τ_4
(x,a)	0	0	1	0	0	0	0	0	1	1	1	1	1	0	1	0
(x,b)	1	0	1	0	0	0	0	0	1	1	1	1	1	0	1	0
(x,c)	1	1	0	1	0	1	0	0	1	1	0	1	1	1	1	1
(y,a)	1	1	0	0	0	0	0	0	1	1	1	1	1	1	0	0
(y,b)	1	1	1	1	0	0	1	0	1	1	1	1	1	1	1	1
(y,c)	1	0	0	1	0	0	0	0	1	0	0	1	1	1	1	1
							...									

(b) Bitsets

Fig. 1: Bitsets supports, supports*, supportsMin and supportsMax

techniques: bitwise operations [2, 18], residual supports [14, 15, 20], tabular reduction [13, 16, 28], reversible sparse sets [27] and resetting operations [24].

The core structure of CT, when applied to a constraint c, is a reversible sparse bitset, called currTable, responsible for keeping track of the current supports of c: the ith bit of currTable is set to 1 iff the ith tuple τ_i of the table of c is currently valid. It is updated by means of precomputed bitsets: for each value (x, a) of c, supports$[x, a]$ is the bitset that identifies the set of tuples that are initially supports of (x, a) on c.

Algorithm 1 presents a simplified version of CT, which consists of two main steps. First, updateTable(), iterates for each variable x involved in c over either the set Δ_x of values that have been removed from the domain of x since the last invocation of the algorithm (Line 5) or the current domain of x (Line 9), and use the appropriate bitsets supports to update currTable using bitwise operations (Lines 6 and 10). The test at Line 4 ensures minimizing the number of operations that must be performed during the update. Secondly, filterDomains(), iterates over every value (x, a) of c (Lines 13 and 14) and use the corresponding bitset supports$[x, a]$ to verify whether the value is still supported or not (Line 15). It should be noted that the bitwise operations on the bitsets currtable and mask are only performed on the active (i.e., non null) words of currtable.

In [29], it was shown how CT can be extended to (positive) short tables. CT* just requires the introduction of a second pool of bitsets: for each value (x, a) of c, supports*$[x, a]$ is the bitset that identifies the set of tuples τ that are *explicit* supports of (x, a), i.e., such that $\tau[x] = a$. This means that any occurrence of * in a short tuple implies that the corresponding bits are always set to 0 in the bitsets supports* (unlike bitsets supports). An illustration is given by Fig.1 where the bits for τ_1 in bitsets supports and supports* are different because $\tau_1[y] = *$. CT* is obtained from CT by simply replacing Line 6 of Algo.1 with:

mask $\leftarrow$ mask $\mid$ supports*$[x, a]$

The complexity of CT* is $\mathcal{O}(rd\frac{t}{w})$ where r denotes the arity, d the size of the largest domain, t the number of tuples and w the size of the computer words (e.g., $w = 64$).

Algorithm 1: Class ConstraintCT

```
1  Method updateTable()
2      foreach variable x ∈ scp do
3          mask ← 0
4          if |Δ_x| < |dom(x)| then                    // Incremental Update
5              foreach value a ∈ Δ_x do
6                  mask ← mask | supports[x, a]   // Use supports*[x, a] in CT*
7              mask ← ~mask                          // ~: bitwise negation
8          else                                       // Reset-based Update
9              foreach value a ∈ dom(x) do
10                 mask ← mask | supports[x, a]          // | : bitwise or
11         currTable ← currTable & mask                 // & : bitwise and
12 Method filterDomains()
13     foreach variable x ∈ scp do
14         foreach value a ∈ dom(x) do
15             if currTable & supports[x, a] = 0 then
16                 dom(x) ← dom(x) \ {a}
17 Method enforceGAC()
18     updateTable()
19     filterDomains()
```

4 CT on Basic Smart Tables

A basic smart table is composed of smart tuples containing expressions of the following forms: $*$, <op> v, $\in S$ and $\notin S$, with <op> $\in \{<, \leq, =, \neq, \geq, >\}$. As CT* already covers the forms $*$ and $= v$, we need to further extend CT* to handle the remaining forms of smart tuples in a basic smart table. The resulting algorithm is called CT^{bs}.

4.1 Handling $\neq v$

In reality, CT* can already handle expressions of the form $\neq v$. We simply have to slightly extend the semantics of bitsets supports*. Any occurrence of * or $\neq v$ in a tuple implies that the corresponding bits are set to 0 in supports* (unlike supports). The complexity is obviously unchanged. An example is depicted with $\tau_1[x]$ in Fig.1. Correctness is proved by showing that the table is always properly updated. Let us cover all possible cases for an expression $\neq v$ at column x. When $|dom(x)| = 0$, the solver detect the failure through the variables. For the case of the reset-based update, as support precisely depicts the acceptance of values by tuples, this is necessarily correct. In the incremental update, we will necessary have $|dom(x)| \geq 2$. This comes from the structure of the algorithm when the full version, as described in [5], is used. In this case, the tuple is always support for the given variable and as the corresponding bits in supports* are set to 0 by construction, the tuple is not removed from currTable.

4.2 Handling <op> v, with <op> $\in \{<, \leq, \geq, >\}$

First, note that it is sufficient to focus on expressions of the form $\geq v$ and $\leq v$ since $> v$ and $< v$ are equivalent to $\geq v+1$ and $\leq v-1$. We first introduce two additional arrays of bitsets: `supportsMin` for $\leq v$ and `supportsMax` for $\geq v$. For each value (x, a) of c, the ith bit of `supportsMin`[x, a] (resp., `supportsMax`[x, a]) is 1 iff $\tau_i[x]$ allows at least one value $\geq a$ (resp., $\leq a$). An example of the different bitsets for each of the operators $<, \leq, \geq, >$ can be found in Fig.1. We assume the ordering $a < b < c$ on domains.

To handle <op> v, with <op> $\in \{\leq, \geq\}$, Lines 4-7 (incremental update) in Algo.1 must be replaced by the lines given in Algo.2. Note that `min` (resp. `max`) denotes the smallest (resp. largest) value of $dom(x)$, whereas `minChanged`() (resp. `maxChanged`()) is a method that return true when `min` (resp. `max`) have changed since the last call of the algorithm. Line 1 is slightly modified to compensate the overhead induced by the two operations. Because Lines 5-8 handle all the values that are less than and greater than `min` and `max`, we only consider at Line 3 the values $a \in \Delta_x$ such that $dom(x).\texttt{min} < a < dom(x).\texttt{max}$. Note that the semantics of $\texttt{supports}^*$ is unchanged: only *explicit* supports of (x, a) are considered, meaning that we have $\texttt{supports}^*[x, a] = 0$ when $\tau[x] = *$, $\tau[x] \neq b$, $\tau[x] \geq b$ or $\tau[x] \leq b$ for any value b.

Correctness is shown for $\leq v$, considering all cases at column x for tuple τ. The case $|dom(x)| = 0$ is as trivial as in the last section. For the case of the reset-based update, as `supports` precisely depicts the acceptance of values by tuples, this is necessarily correct. Finally in the incremental update (Algo.2), due to the constructions of the bitsets, i.e., the bit for τ in $\texttt{supports}^*$ (resp. `supportsMax`) is always set to 0 (resp. 1), updating depends only on `supportsMin`. By definition, if $dom(x).\texttt{min}$ is $\leq v$, meaning still supported, the bit for τ in `supportsMin` is 1, keeping τ in `currTable`. If $dom(x).\texttt{min} > v$, bit for τ is 0, removing τ. Time complexity remains $\mathcal{O}(rd\frac{t}{w})$.

Algorithm 2: Incremental Update for CT^{bs}

```
1 if |Δ_x| + 2 < |dom(x)| then
2     foreach value a ∈ Δ_x such that dom(x).min < a < dom(x).max do
3         mask ← mask | supports*[x, a]
4     mask ← ~mask
5     if dom(x).minChanged() then
6         mask ← mask & supportsMin[x, dom(x).min]
7     if dom(x).maxChanged() then
8         mask ← mask & supportsMax[x, dom(x).max]
```

4.3 Handling $\in S$ (and $\notin S$)

There is no easy way to handle expressions of the form $\in S$ using incremental update (on bitsets). We then propose to systematically execute reset-based update as they do in [30] for passing from STRbit to STRbit-C. More precisely, as soon as a variable is

involved in an expression of the form $\in S$ in one of the tuples of the basic smart table, a reset-based update is forced in Algo.1 (Lines 9-10). We do not present the (rather immediate) code. Dealing with $\notin S$ can be conducted similarly.

5 Compression

Computing the smallest short table (compression with only $*$) is known to be NP-complete [10]. Not surprisingly computing the smallest basic smart table is also NP-complete. Indeed minimizing the size of a DNF formula (NP-complete [12]) can be reduced in polynomial time to minimizing the size of a basic smart table.

We introduce a heuristic compression algorithm to generate a basic smart table from a given (ordinary) table. It focuses on column constraints of the form $\leq v$ and $\geq v$. Other forms can be obtained by post-processing: i) expressions $\leq dom(x).\texttt{max}$ or $\geq dom(x).\texttt{min}$ can be replaced by $*$, and ii) two tuples that are identical except on a column where we have respectively $\leq v-1$ and $\geq v+1$ can be merged by simply using $\neq v$. Expressions $\in S$ and $\notin S$ were not considered in this heuristic to avoid costly set operations.

The compression algorithm proceeds in r steps, r being the arity of the table. At each step, the algorithm handles two tables: the c-table (compressed table) and the r-table (residual table). The union of c-table and r-table is always equivalent to the initial table. At step i, each tuple of the c-table has exactly i column constraints of the form $\leq v$ or $\geq v$. When $i=0$, c-table is the initial table and r-table is empty. After step r, the resulting table of the algorithm is the union of c-table and r-table. The computation at a given step is the following. From the tuples in c-table, several abstract tuples are generated, used to introduce new tuples with one more column constraint of the form $\leq v$ or $\geq v$. The new tuples that cover at least two tuples in c-table are gathered in a new c-table used in the next step. The uncovered tuples in c-table are added to r-table.

More formally, at a given step, we define an *abstract tuple* as a tuple taken from the current c-table with one of its literal value $x=a$ replaced by the symbol '?'. At step i, there are thus $(r-i)\cdot t_c$ possible abstract tuples, with t_c the size of c-table. An abstract tuple can be matched against so-called strictly compatible (resp. compatible) tuples. A basic smart tuple τ is strictly compatible (resp. compatible) with an abstract tuple ρ iff for each $1 \leq j \leq r$, the form of $\tau[j]$ is strictly compatible (resp. compatible) with the form of $\rho[j]$. Compatibility of forms is intuitive: a value v is compatible with the same value v and also with '?', the form $\leq v$ (resp. $\geq v$) is compatible with $\leq w$ (resp. $\geq w$) provided that $w \geq v$ (resp. $w \leq v$). Strict compatibility requires compatibility and $w=v$.

We denote by S^ρ_c (resp., S^ρ_{sc}) the sets of tuples from the current c-table that are compatible (resp., strictly compatible) with ρ, an abstract tuple. Note that the computation of these two sets can be done in $O(r.t_c)$ and that we have $S^\rho_{sc} \subset S^\rho_c$. Given $S^\rho_c = \{\tau_1, \ldots, \tau_k\}$, we denote by V^ρ set of values $\{\tau_1[j], \ldots, \tau_k[j]\}$ where j is the column index of ? in ρ. If, given the domain of x_j, a subset of V^ρ can be represented by $x_j \leq v$ (or $x_j \geq v$), then a new basic smart tuple ρ' is generated, where ρ' is the tuple ρ with ? replaced by $\leq v$ (or $\geq v$). The corresponding tuples in S_{sc} can be removed as they are covered by the new smart tuple. However, the tuples only present in S_c can-

not be removed. In practice, a new basic smart tuple is only introduced if it ensures a reduction of the table (i.e., at least two tuples can be removed). As t_c is $O(t)$, the total complexity of the compression algorithm is $O(r^3t^2)$.
Example. Let us consider the abstract tuple $\rho = (1, ?, \leq 1)$. In the following set of basic smart tuples $\{\tau_1 = (1, 0, \leq 1), \tau_2 = (1, 1, \leq 2), \tau_3 = (1, 2, \leq 1)\}$, the tuples τ_1 and τ_3 are strictly compatible with ρ, the tuple τ_2 is only compatible with ρ. The new smart tuple $(1, \leq 2, \leq 1)$ is then generated, allowing us to remove both τ_1 and τ_3. The tuple τ_2 is necessary to generate this new tuple, but cannot be removed from the table.

6 Experimental Results

We have selected from the XCSP3 website [3] the instances that exclusively contain positive table constraints. This benchmark includes a large variety of series.

Compression of ordinary tables into basic smart tables. The compression ratio is defined as $\frac{t'}{t}$, where t and t' respectively denote the numbers of tuples in the initial and compressed tables. Using the algorithm described in Sect.5, we obtain the results displayed in Fig.2. As we expected, dense tables (i.e., tables with a high number of tuples compared to the Cartesian product of domains) lead to good compression. This can be observed in particular with the series PigeonsPlus that contains really dense instances (making them highly compressible), and also the series Renault that contains instances with a wide range of tables (many of them being well compressed). On the other hand, the series Kakuro contains very sparse tables that cannot be compressed at all.

Practical efficiency. To assess the efficiency of CT^{bs}, notably the interest of using the different forms of expressions, tables have been compressed using our algorithm in three different related ways: (1) compression with $\leq$ and $\geq$, (2) compression with $\leq$ and $\geq$ followed by a post-processing to detect $*$ and $\neq$ and (3) compression with $\leq$ and $\geq$ followed by a transformation into set restrictions (e.g., $\leq v$ is written as $\{i : i \leq v\}$).

Figure 3 shows the performance profile [6] for CT^{bs} with these three related compression approaches and also for standard CT on uncompressed tables. A point (x, y) on the plot indicates the percentage of instances that can be solved within a time-limit that is at most x times the time taken by the best algorithm. The performance profile was based only on instances showing enough compression (rate ≤ 0.9) and requiring at least 2 seconds of solving time. With a timeout set to 10 min, only 60 instances matched out these criteria out of the $4,000$ tested instances.

Obtained results show that simple compression (1) brings a slight speedup compared to CT. Notice however that the computation time for an instance was reduced up to a factor of 7. Because post-processing (2) brought less than 3% of additional compression, it is not surprising that CT^{bs} with approaches (1) and (2) are close. As expected, handling tables with set restrictions only, approach (3), induces an overhead as no incremental updates can be performed. The overhead is however limited (at most a factor two). The computation time taken by Method `updateDomain()` in Algo.1 is not much reduced when using basic smart tables (mainly, because of the residue caching described in [5]). This explains why the observed speed-ups are not proportional to the compression ratios.

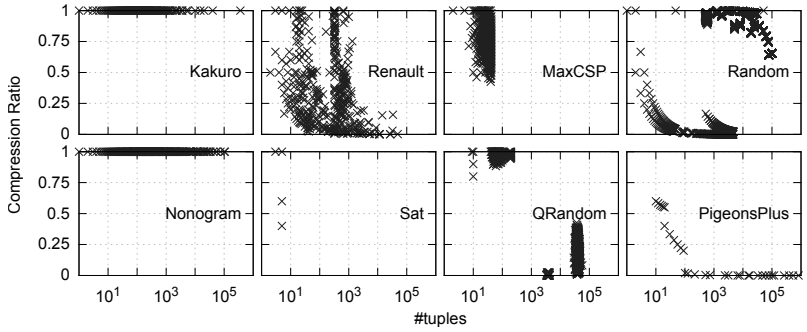

Fig. 2: Distribution of table compression ratios on 8 series of instances.

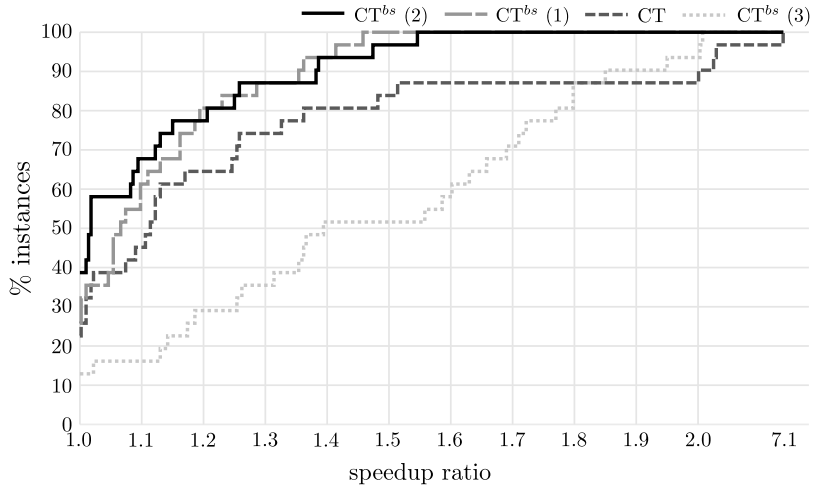

Fig. 3: Performance profile for CT^{bs} versus CT

7 Conclusion

In this paper, we have shown how to extend CT^*, the Compact-Table algorithm devised for (ordinary and) short tables, to basic smart tables. The new algorithm CT^{bs} benefits from the highly optimized mechanisms of CT and can be attractively applied to expressive forms of tables involving natural conditions on values ($*$, $\neq v$, $\leq v$, $\geq v$ and $\in S$).

We have also proposed a heuristic algorithm to generate basic smart tables. Our experimental results show both the usefulness of this form of compression and the good behavior of CT^{bs} compared to CT.

References

[1] Bessiere, C., Régin, J.C.: Arc consistency for general constraint networks: preliminary results. In: Proceedings of IJCAI'97. pp. 398–404 (1997)

[2] Bliek, C.: Wordwise algorithms and improved heuristics for solving hard constraint satisfaction problems. Tech. Rep. 12-96-R045, ERCIM (1996)

[3] Boussemart, F., Lecoutre, C., Piette, C.: XCSP3: An integrated format for benchmarking combinatorial constrained problems. Tech. Rep. arXiv:1611.03398, CoRR (2016), available from `http://www.xcsp.org`

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

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

[6] Dolan, E.D., Moré, J.J.: Benchmarking optimization software with performance profiles. Mathematical programming 91(2), 201–213 (2002)

[7] Gent, I., Jefferson, C., Miguel, I., Nightingale, P.: Data structures for generalised arc consistency for extensional constraints. In: Proceedings of AAAI'07. pp. 191–197 (2007)

[8] Gharbi, N., Hemery, F., Lecoutre, C., Roussel, O.: Sliced table constraints: Combining compression and tabular reduction. In: Proceedings of CPAIOR'14. pp. 120–135 (2014)

[9] J.-B. Mairy, P.V.H., Deville, Y.: Optimal and efficient filtering algorithms for table constraints. Constraints 19(1), 77–120 (2014)

[10] Jefferson, C., Nightingale, P.: Extending simple tabular reduction with short supports. In: Proceedings of IJCAI'13. pp. 573–579 (2013)

[11] Katsirelos, G., Walsh, T.: A compression algorithm for large arity extensional constraints. In: Proceedings of CP'07. pp. 379–393 (2007)

[12] Khot, S., Saket, R.: Hardness of minimizing and learning dnf expressions. In: Foundations of Computer Science, 2008. FOCS'08. IEEE 49th Annual IEEE Symposium on. pp. 231–240. IEEE (2008)

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

[14] Lecoutre, C., Boussemart, F., Hemery, F.: Exploiting multidirectionality in coarse-grained arc consistency algorithms. In: Proceedings of CP'03. pp. 480–494 (2003)

[15] Lecoutre, C., Hemery, F.: A study of residual supports in arc consistency. In: Proceedings of IJCAI'07. pp. 125–130 (2007)

[16] Lecoutre, C., Likitvivatanavong, C., Yap, R.: STR3: A path-optimal filtering algorithm for table constraints. Artificial Intelligence 220, 1–27 (2015)

[17] Lecoutre, C., Szymanek, R.: Generalized arc consistency for positive table constraints. In: Proceedings of CP'06. pp. 284–298 (2006)

[18] Lecoutre, C., Vion, J.: Enforcing arc consistency using bitwise operations. Constraint Programming Letters 2, 21–35 (2008)

[19] Lhomme, O., Régin, J.C.: A fast arc consistency algorithm for n-ary constraints. In: Proceedings of AAAI'05. pp. 405–410 (2005)

[20] Likitvivatanavong, C., Zhang, Y., Bowen, J., Freuder, E.: Arc consistency in MAC: a new perspective. In: Proceedings of CPAI'04 workshop held with CP'04. pp. 93–107 (2004)

[21] Mairy, J.B., Deville, Y., Lecoutre, C.: The smart table constraint. In: Proceedings of CPAIOR'15. pp. 271–287 (2015)

[22] van Omme, N., Perron, L., Furnon, V.: or-tools user's manual. Tech. rep., Technical report, Google (2014), available from `https://github.com/google/or-tools`

[23] OscaR Team: OscaR: Scala in OR (2012), available from https://bitbucket.org/oscarlib/oscar
[24] Perez, G., Régin, J.C.: Improving GAC-4 for Table and MDD constraints. In: Proceedings of CP'14. pp. 606–621 (2014)
[25] Pesant, G.: A regular language membership constraint for finite sequences of variables. In: Proceedings of CP'04. pp. 482–495 (2004)
[26] Prud'homme, C., Fages, J.G., Lorca, X.: Choco3 documentation. TASC, INRIA Rennes, LINA CNRS UMR 6241 (2014)
[27] de Saint-Marcq, V.L.C., Schaus, P., Solnon, C., Lecoutre, C.: Sparse-sets for domain implementation. In: (TRICS) Workshop on Techniques for Implementing Constraint programming Systems (2013)
[28] Ullmann, J.: Partition search for non-binary constraint satisfaction. Information Science 177, 3639–3678 (2007)
[29] Verhaeghe, H., Lecoutre, C., Schaus, P.: Extending compact-table to negative and short tables. In: Proceedings of AAAI'17 (2017)
[30] Wang, R., Xia, W., Yap, R., Li, Z.: Optimizing Simple Tabular Reduction with a bitwise representation. In: Proceedings of IJCAI'16. pp. 787–795 (2016)
[31] Xia, W., Yap, R.: Optimizing STR algorithms with tuple compression. In: Proceedings of CP'13. pp. 724–732 (2013)