Extending Compact-Table to Negative and Short Tables

Hélène Verhaeghe[1] and **Christophe Lecoutre**[2] 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 very useful for modeling combinatorial constrained problems, and thus play an important role in Constraint Programming (CP). During the last decade, many algorithms have been proposed for enforcing the property known as Generalized Arc Consistency (GAC) on such constraints. A state-of-the art GAC algorithm called Compact-Table (CT), which has been recently proposed, significantly outperforms all previously proposed algorithms. In this paper, we extend this algorithm in order to deal with both short supports and negative tables, i.e., tables that contain universal values and conflicts. Our experimental results show the interest of using this fast general algorithm.

Introduction

Table constraints, also called extension(al) constraints, explicitly express for the variables they involve, either the allowed combinations of values, called *supports*, or the forbidden combinations of values, called *conflicts*. Table constraints can theoretically encode any kind of restrictions and are consequently very important in Constraint Programming (CP). Indeed, as especially claimed by people from industry (e.g., IBM and Google), table constraints are often required when modeling combinatorial constrained problems in many application fields. The design of filtering algorithms for such constraints has generated a lot of research effort, see (Bessiere and Régin 1997; Lhomme and Régin 2005; Lecoutre and Szymanek 2006; Gent et al. 2007; Ullmann 2007; Lecoutre 2011; Lecoutre, Likitvivatanavong, and Yap 2015; J.-B. Mairy and Deville 2014; Perez and Régin 2014; Wang et al. 2016; Demeulenaere et al. 2016).

On classical tables, i.e., sequences of ordinary tuples, the algorithmic progresses that have been made over the years for maintaining the property called GAC (Generalized Arc Consistency) are quite impressive. Roughly speaking, an algorithm such as Compact-Table (Demeulenaere et al. 2016) is about one order of magnitude faster than the best algorithm(s) proposed a decade ago (Lhomme and Régin 2005; Lecoutre and Szymanek 2006; Gent et al. 2007; Ullmann 2007). Unfortunately, table constraints admit practical boundaries because the memory space required to represent them may grow exponentially with their arity. To reduce space complexity, researchers have focused on various forms of compression. For example, tries (Gent et al. 2007), Multi-valued Decision Diagrams (MDDs) (Cheng and Yap 2010; Perez and Régin 2014) and Deterministic Finite Automaton (DFA) (Pesant 2004) are general structures used to represent table constraints in a compact way, so as to facilitate filtering process.

Cartesian product is another classical mechanism to represent compactly large sets of tuples. This is the approach followed by works on compressed tuples (Katsirelos and Walsh 2007; Régin 2011; Xia and Yap 2013) and short supports and tuples (Jefferson and Nightingale 2013). A short tuple allows the presence of universal values, denoted by the symbol *, meaning that some variables can take any values from their domains. Other forms of compact representation are obtained by means of sliced tables (Gharbi et al. 2014) and smart tables (Mairy, Deville, and Lecoutre 2015).

Compact-Table (CT) is a state-of-the-art GAC algorithm for *positive* (ordinary) table constraints, i.e., constraints defined by tables containing (uncompressed) supports. In this paper, we extend CT in order to be able to deal with:

- negative tables (i.e., tables containing conflicts),
- and/or short tuples (i.e., tuples containing the symbol *).

Technical Background

A *constraint network* (CN) N is composed of a set of n variables and a set of e constraints. Each *variable* x has an associated domain, denoted by $dom(x)$, 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*, denoted by $rel(c)$, 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 defined explicitly by listing (in a table) the tuples that are allowed by c or the tuples that are disallowed by c. In the former case, the table constraint is said to be *positive* whereas in the latter case, it is said *negative*.

Let $\tau = (a_1, a_2, \ldots, a_r)$ be a tuple of values associated with an ordered set of variables $vars(\tau) = \{x_1, x_2, \ldots, x_r\}$. The ith value of τ is denoted by $\tau[i]$ or

	x	y	z
τ_{1a}	c	a	a
τ_{1b}	c	b	a
τ_{1c}	c	c	a
τ_2	a	b	c
τ_3	b	c	b

(a) An ordinary table

	x	y	z
τ_1	c	$*$	a
τ_2	a	b	c
τ_3	b	c	b

(b) A short table

Figure 1: Equivalence between ordinary and short tables

$\tau[x_i]$, and τ is *valid* iff $\forall i \in 1..r, \tau[i] \in dom(x_i)$. τ is a *support* (resp., a *conflict*) on a constraint c such that $vars(\tau) = scp(c)$ iff τ is a valid tuple allowed (resp., disallowed) by c. If τ is a support (resp., a conflict) on a constraint c involving a variable x and such that $\tau[x] = a$, we say that τ is a *support for* (resp., a *conflict for*) (x, a) on c.

Generalized Arc Consistency (GAC) is a well-known domain-filtering consistency defined as follows: a constraint c is GAC iff $\forall x \in scp(c), \forall a \in dom(x)$, there exists at least one support for (x, a) on c. A CN N is GAC iff every constraint of N is GAC. Enforcing GAC is the task of removing from domains all values that have no support on some constraint(s). Many algorithms have been devised for establishing GAC according to the nature of the constraints.

A very useful form of compression for tables is based on the concept of short tuples (Jefferson and Nightingale 2013). A *short* tuple allows some variables to be left out, meaning that these variables can take any values from their domains, which is represented by the symbol *. As an illustration, Figure 1 shows on the left an ordinary table, and on the right an equivalent short table, i.e., a table containing short tuples. Here, assuming that $dom(y) = \{a, b, c\}$, the short tuple $\tau_1 = (c, *, a)$ represents the three *ordinary* tuples $\tau_{1a} = (c, a, a)$, $\tau_{1b} = (c, b, a)$ and $\tau_{1c} = (c, c, a)$, and we say that these three tuples are *subsumed* by τ_1. A short tuple τ is valid iff $\forall i \in [1; r], \tau[i] = * \texttt{ or } \tau[i] \in dom(x_i)$.

Compact-Table (CT) Algorithm

Compact-Table (CT) is a state-of-the-art algorithm for enforcing GAC on positive table constraints (Demeulenaere et al. 2016). It first appeared in Or-Tools, the Google solver that has been very competitive at the latest MiniZinc Challenges, and is now implemented in constraint solvers OscaR, AbsCon and Choco. CT benefits from well-established techniques: bitwise[1] operations (Bliek 1996; Lecoutre and Vion 2008), residual supports (Lecoutre, Boussemart, and Hemery 2003; Likitvivatanavong et al. 2004; Lecoutre and Hemery 2007), tabular reduction (Ullmann 2007; Lecoutre 2011; Lecoutre, Likitvivatanavong, and Yap 2015) and resetting operations (Perez and Régin 2014). This section briefly describes the algorithm.

CT, applied to a positive table constraint c, introduces a bitset called `currTable` that keeps track at every node of the search tree built by a backtrack algorithm that maintains GAC the tuples in the table of c that are currently valid: the ith bit of `currTable` is set to 1 iff the ith tuple τ_i of the table of c is currently valid. To help updating dynamically this structure, a bitset called `supports`$[x, a]$ is computed initially (and never updated) for every value (x, a) of c. Each bit at position i indicates if the ith tuple τ_i of the table of c contains (x, a), i.e., is such that $\tau_i[x] = a$. An illustration is given by Figure 2.

	x	y	z
τ_1	c	a	a
τ_2	a	b	c
τ_3	b	c	b

(a) A positive table with 3 tuples

τ_1	τ_2	τ_3
1	1	1

(b) Bitset `currTable`

	τ_1	τ_2	τ_3
$[x, a]$	0	1	0
$[x, b]$	0	0	1
$[x, c]$	1	0	0
$[y, a]$	1	0	0
$[y, b]$	0	1	0
$[y, c]$	0	0	1
		...	

(c) Bitsets `supports`

Figure 2: Bitsets introduced for CT

In this paper, we present a simplified form of CT, Algorithm 1. The main method to call for enforcing GAC on a positive table constraint c (assuming that c is represented by a programming object) is enforceGAC(). Its principle is to update first the current table, filtering out (indices of) tuples that have become invalid, and to check afterwards whether each value has still a support.

When the algorithm is called, we assume that we get for each variable x in the scope of c (simply denoted by scp) the set of values Δ_x that have been removed since the last invocation of the algorithm. This allows us to choose in Method updateTable() between iterating over either the values in the current domain of x or the values in Δ_x, so as to update the bitset `currTable`. An illustration of these two updating modes is given by Figure 3: we suppose here that $\Delta_y = \{b\}$, and we can observe that choosing the incremental update saves some operations compared to the reset-based one. Note that the variable `mask` in Method updateTable() is a local bitset used to update `currTable` through bitwise operations.

Once the current table has been updated, Method filterDomains() tests if each value has still a support by means of a simple bitwise intersection. For example, if `currTable` is `1 0 1`, we can infer that the value (x, a) can be removed because `supports`$[x, a]$ is `0 1 0` and

```
1 0 1  &  0 1 0  =  0 0 0
```

Of course, many improvements, not detailed here due to lack of space, permit a very efficient filtering process. Limiting some operations to subsets of variables (denoted by $\texttt{S}^{\texttt{val}}$ and $\texttt{S}^{\texttt{sup}}$) or exploiting so-called residues has been proved to be effective. Also, it is very important to note that each bitset is a non-trivial data structure. Basically, each bitset `bs` is defined by an array `bs.words` of computer 64-bit words, with `bs.length` indicating the number of words. Im-

[1] Exploiting bit vectors becomes a more and more popular topic in CP. See, e.g., (Van Kessel and Quimper 2012; Michel and Van Hentenryck 2012; Wang, Søndergaard, and Stuckey 2016).

Algorithm 1: Class ConstraintCT

```
1  Method updateTable()
2      foreach variable x ∈ scp do
3          mask ← 0
4          if |Δ_x| < |dom(x)| then
5              foreach value a ∈ Δ_x do
6                  mask ← mask | supports[x, a]
7              mask ← ~mask
8          else
9              foreach value a ∈ dom(x) do
10                 mask ← mask | supports[x, a]
11         currTable ← currTable & mask
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     if currTable = 0 then
20         return Backtrack
21     filterDomains()
```

portantly, an index is used to identify at any moment the subset of non-zero words in the bitset `currTable`, i.e., the subset of words that contain at least one bit set to 1. By means of a sparse-set structure (Briggs and Torczon 1993; le Clément de Saint-Marcq et al. 2013), which permits efficient reversibility when backtracking, all bitwise operations can be performed with respect to only these non-zero words. For example, suppose that the table of c initially contains $6,400$ tuples, forming an array of 100 64-bit words, and that at given node of the search tree, there is only one non-zero word in `currTable`. Then, all operations at Lines 3, 6, 7, 10, 11, 15 and 19 in Algorithm 1 are processed while dealing with only one word. Details can be found in (Demeulenaere et al. 2016). Finally, note that 0 used at Lines 3 and 19 means computer words with all bits set to 0.

Dealing with Short Tables: CT*

Interestingly, CT can be easily adapted to deal with positive tables containing short tuples without an increasing of the worst-case time complexity, which is $\mathcal{O}(rd\frac{t}{w})$ where r denotes the constraint arity, d the size of the largest domain, t the number of tuples and w the size of the computer words (e.g., $w = 64$). To handle short tables, a small modification is required: instead of using for each value (x, a) of c the bitset `supports`$[x, a]$ in both update strategies (see Lines 6 and 10 in Algorithm 1), we need two separate related bitsets. For the reset-based update, we use the bitset `supports`$[x, a]$ whose ith bit indicates if (x, a) is *accepted* by the ith tuple τ_i of the table of c, i.e., if $\tau_i[x] = a \vee \tau_i[x] = *$. For the incremental update, we use the bitset `supports`$^*[x, a]$ whose ith bit indicates if (x, a) is *strictly accepted* by the ith tuple

	τ_1	τ_2	τ_3
(A) = `currTable`	1	1	1
(B) = `supports`$[y, b]$	0	1	0
(C) = ~(B)	1	0	1
(A) & (C)	1	0	1

(a) Incremental update

	τ_1	τ_2	τ_3
(A) = `currTable`	1	1	1
(B) = `supports`$[y, a]$	1	0	0
(C) = `supports`$[y, c]$	0	0	1
(D) = (B) | (C)	1	0	1
(A) & (D)	1	0	1

(b) Reset-based update

Figure 3: Updating `currTable` from $\Delta_y = \{b\}$. (A) & (C) on top, as well as (A) & (D) on bottom, allow us to compute the new value of `currTable`.

	τ_1	τ_2	τ_3
(x, a)	0	1	0
(x, b)	0	0	1
(x, c)	1	0	0
(y, a)	1	0	0
(y, b)	1	1	0
(y, c)	1	0	1
	...		

(a) Bitsets `supports`

	τ_1	τ_2	τ_3
(x, a)	0	1	0
(x, b)	0	0	1
(x, c)	1	0	0
(y, a)	0	0	0
(y, b)	0	1	0
(y, c)	0	0	1
	...		

(b) Bitsets `supports`*

Figure 4: Bitsets `supports` and `supports`*

τ_i of the table, i.e., if $\tau_i[x] = a$. This means that for each occurrence of * in a short tuple, the corresponding bits are always set to 0 in the bitsets `supports`*. Figure 4 shows an illustration, where bitsets `supports` and `supports`* are given for the short table depicted in Figure 1b.

Proposition 1 *Algorithm 1, applied to a positive short table constraint enforces GAC if Line 6 is replaced by:*
`mask` $\leftarrow$ `mask` | `supports`$^*[x, a]$

Proof: This holds because a short tuple τ such that $\tau[y] = *$, is valid for any value remaining in $dom(y)$. ■

At this stage, it is worthwhile to mention that a recently published algorithm (Wang et al. 2016), called STRbit, also exploits bit vectors. However, the data structures employed are quite different, as for example, the main table VAL is not shrunk dynamically contrary to `currTable` as well as the bitsets BIT_SUP playing the role of supports. A variant called STRbit-C can be used on compressed tuples, which can be seen as encompassing short tuples. However the data structures are quite sophisticated, which makes the handling of short tuples non trivial. Besides, STRbit and its variants have been developed exclusively on positive tables, contrary to what we show in the next two sections for CT.

Dealing with Negative Tables: CT$_{neg}$

The modifications brought to CT for dealing with negative tables, i.e., tables containing disallowed tuples, are discussed now. We keep working with the bitset `currTable` that indicates which tuples from the initial table of c are still valid, and we introduce bitsets `conflicts` that are computed exactly the same way as bitsets `supports` were. If the table in Figure 2a would be assumed to be negative, then the bitsets in Figure 2c would be those for `conflicts`. Simply, as the context is different, the meaning is different: instead of permanently updating the table of supports in `currTable` by means of bitsets `supports`, we permanently update the table of conflicts in `currTable` by means of bitsets `conflicts`.

For filtering, the basic idea is to count for each value (x, a) of c how many valid tuples containing (x, a) are in the current table of c (hence, representing the number of conflicts for (x, a) on c) and to compare this number with the number of valid tuples containing (x, a). When these two numbers are equal, it simply means that all valid tuples containing (x, a) correspond to conflicts, and consequently that no support for (x, a) on c exists. Computing, in the context of a constraint c, the number of valid tuples for any value in the domain of a variable x is simple. This is:

$$\Pi_{y \in scp(c): y \neq x} |dom(y)| \qquad (1)$$

Algorithm 2: Class ConstraintCT$_{neg}$

1 **Method** updateTable()
2   **foreach** *variable* $x \in$ `scp` **do**
3   `mask` $\leftarrow 0$
4   **if** $|\Delta_x| < |dom(x)|$ **then**
5   **foreach** *value* $a \in \Delta_x$ **do**
6   `mask` $\leftarrow$ `mask` $|$ `conflicts`$[x, a]$
7   `mask` $\leftarrow$ ~`mask`
8   **else**
9   **foreach** *value* $a \in dom(x)$ **do**
10   `mask` $\leftarrow$ `mask` $|$ `conflicts`$[x, a]$
11   `currTable` $\leftarrow$ `currTable` & `mask`

12 **Method** filterDomains()
13   **foreach** *variable* $x \in$ `scp` **do**
14   **foreach** *value* $a \in dom(x)$ **do**
15   **if** nb1s(`currTable` & `conflicts`$[x, a]$) $= \Pi_{y \in \texttt{scp}: y \neq x} |dom(y)|$ **then**
16         $dom(x) \leftarrow dom(x) \setminus \{a\}$
17   `currTable` $\leftarrow$ `currTable` & ~`conflicts`$[x, a]$

18 **Method** enforceGAC()
19   updateTable()
20   **if** nb1s(`currTable`) $= \Pi_{x \in \texttt{scp}} |dom(x)|$ **then**
21   **return** *Backtrack*
22   filterDomains()

When Method enforceGAC(), Algorithm 2, is called, the first step is to update the current table, exactly as it is done for positive table constraints, except that the bitsets `conflicts` are used instead of `supports`. After this step, one can possibly detect an inconsistency by computing the number of conflicts in the current table of c. When this number is equal to the number of valid tuples, it means that no more supports exist. Function nb1s(), Algorithm 3, permits to count the total number of bits set to 1 in `currTable` by executing an optimized bitwise statement such as "java.lang.Long.bitCount" (Warren 2002). Again, an optimization, which is not detailed here although used in our implementation, is to iterate over only non-zero words.

Algorithm 3: Function nb1s(bs: Bitset)

`cnt` $\leftarrow 0$
foreach $i \in 1..$`bs.length` **do**
  `cnt` $\leftarrow$ `cnt`$+$ Long.bitCount(`bs.words`$[i]$)
return `cnt`

For filtering domains, we verify whether values have still support or not. We call Function nb1s() on the bitwise intersection of `currTable` and `conflicts`$[$`x, a`$]$ so as to compute the number of conflicts for (x, a) on c. The rest of the algorithm is similar to CT, except that when a value is deleted, we have to update the current table at Line 17.

Proposition 2 *Algorithm 2, applied to a negative table constraint c enforces GAC.*

Proof: By means of Method updateTable() and statement at Line 17, we maintain the set of conflicts on c in `currTable`. At Line 15, we can detect if no more support exists for a given value (x, a), and delete it if necessary. ■

The worst-case time complexity ends up to be $\mathcal{O}(rd\frac{t}{w}k)$ which is the same as CT and CT* multiplied by k the cost of counting the active bits in a word ($k = \log(w)$ when using Long.bitCount or can even be $k = 1$ on some architectures).

Dealing with Negative Short Tables: CT$^*_{neg}$

We now show how we can extend CT to tables that are both short and negative. There is however one limitation[2]: there cannot be any overlapping between two short tuples. Two short tuples τ_1 and τ_2 *overlap* iff there is an ordinary tuple that is both subsumed by τ_1 and τ_2. For example, $(a, *, b)$ and $(*, a, b)$ overlap since they both subsume (a, a, b).

One difficulty is to count (efficiently) the number of tuples subsumed by short tuples. In order to speedup the counting operation, the idea is to group the tuples such that each computer word of the current table only refers to *-similar tuples. Two (ordinary or short) tuples are **-similar* iff they contain the same number of * and at the same positions. For example, $(a, *, b)$ and $(b, *, a)$ are *-similar. To make things clear, let us consider the negative short table depicted in Figure 5a. It contains 5 tuples, and one can observe the *-similarity of τ_2 with τ_3 (since they are both ordinary tuples), and of τ_1 with τ_5. We then split this table of 5 tuples into three groups. Importantly, in order to have only *-similar tuples in each computer word (important property for counting, as

[2]This is due to our need of counting tuples. Overlapping tuples made counting not trivial, and is let as a perspective of this work.

	x	y	z
τ_1	c	$*$	a
τ_2	a	b	c
τ_3	b	c	b
τ_4	a	a	$*$
τ_5	b	$*$	a

(a) A negative short table

	x	y	z
τ_2	a	b	c
τ_3	b	c	b
	$\perp$	$\perp$	$\perp$
	$\perp$	$\perp$	$\perp$
τ_1	c	$*$	a
τ_5	b	$*$	a
	$\perp$	$\perp$	$\perp$
	$\perp$	$\perp$	$\perp$
τ_4	a	a	$*$
	$\perp$	$\perp$	$\perp$
	$\perp$	$\perp$	$\perp$
	$\perp$	$\perp$	$\perp$

(b) Tuples of the table grouped by *-similarity on 4-bit words

τ_2	τ_3			τ_1	τ_5			τ_4			
1	1	0	0	1	1	0	0	1	0	0	0

(c) Restructured Bitset `currTable`

Figure 5: Restructuration of negative short tables

seen later), we propose a very simple procedure that consists in padding entries for each incomplete word with dummy tuples (i.e., tuples only containing a special value $\perp$ that is not present in the initial domains of the variables) until the word is complete. Assuming computer 4-bits words, on our example, we obtain 3 words as shown in Figure 5b. The restructured bitset `currTable` is shown in Figure 5c; note the presence of bits set to 0 to discard dummy tuples.

Once the bitset `currTable` has been restructured, counting can be advantageously achieved for a given computer word in conjunction with bit-wise operations. Indeed, the number of ordinary tuples subsumed by any (short) tuple referred to in a given word of `currTable` is necessarily the same. For example, assuming that $dom(y) = \{a, b, c\}$, τ_1 and τ_5, referred to in the second word of `currTable`, subsume exactly 3 ordinary tuples each. For simplicity, in what follows, we consider that `nbSubsumedTuples`(i) indicates the number of ordinary tuples subsumed by any (short) tuple referred to in the ith word of `currTable`. On our example, `nbSubsumedTuples`(2) returns 3. With this auxiliary function, which can benefit from a cache in practice, counting is now performed by Function nb1s*, Algorithm 4.

Similarly to CT*, we also need two separate related bitsets for each value (x, a) of the negative short table constraint c. For the reset-based update, we use the bitset `conflicts`$[x, a]$ whose ith bit indicates if the value (x, a) is accepted by the ith tuple τ_i of the table of c, i.e., if $\tau_i[x] = a \lor \tau_i[x] = *$. For the incremental update, we use the bitset `conflicts`$^*[x, a]$ whose ith bit indicates if (x, a) is strictly accepted by the ith tuple τ_i of the table, i.e., if $\tau_i[x] = a$. Of course, we need to take dummy tuples into account when building these structures.

Proposition 3 *Algorithm 2, applied to a negative short table constraint c enforces GAC if*

- *structures* `conflicts` *are replaced by structures* `conflicts`* *at Lines 6, 15 and 17,*
- *calls to Function* nb1s *are replaced by calls to Function* nb1s* *at Lines 15 and 20.*

Proof: First, with Function nb1s*, we count the right number of bits set to 1 because computer words only contain *-similar tuples. At Lines 6 and 17, we update `currTable` with `conflicts`* because * is a universal value, and consequently, a short tuple can never become invalid through *. Finally, when we count the number of bits set to 1 at Line 15, we use again `conflicts`* because we have to consider all possible values for each *. ■

The worst-case time complexity, similar to CT$_{neg}$, is $\mathcal{O}(rd\frac{t'}{w}k)$, with t' representing the number of tuples in the table, including the dummy ones.

Algorithm 4: Function nb1s*(bs: Bitset)

```
cnt ← 0
foreach i ∈ 1..bs.length do
    bc ← Long.bitCount(bs.words[i])
    cnt ← cnt + bc ∗ nbSubsumedTuples(i)
return cnt
```

Experimental Results

We have implemented all algorithms described in this paper, namely, CT, CT*, CT$_{neg}$ and CT$^*_{neg}$ in the Oscar solver (OscaR Team 2012), using 64-bit words (Long). Our implementation benefits from all optimization techniques described in (Demeulenaere et al. 2016), which were briefly discussed in the section about CT. Notably, we manage sparse sets in order to avoid handling zero computer words. All the results of our experiments are displayed using performance profiles (Dolan and Moré 2002). A performance profile is a cumulative distribution of the speedup performance of an algorithm $s \in S$ compared to other algorithms of S over a set I of instances: $\rho_s(\tau) = \frac{1}{|I|} \times |\{i \in I : r_{i,s} \leq \tau\}|$ where the performance ratio is defined as $r_{i,s} = \frac{t_{i,s}}{\min\{t_{i,s} | s \in S\}}$ with $t_{i,s}$ the time obtained with algorithm $s \in S$ on instance $i \in I$. A ratio $r_{i,s} = 1$ thus means that s is the fastest on instance i.

Unfortunately, to the best of our knowledge, there are no available benchmarks for positive and negative short tables. This can be explained by the fact that the first algorithm dedicated to positive short tables has only been published recently (Jefferson and Nightingale 2013), and that CT$^*_{neg}$ is the first algorithm in the literature that can deal with negative short tables. However, we expect that short tables will become popular in the near future because i) they represent a useful modeling tool, ii) they can be directly represented in format XCSP3 (Boussemart, Lecoutre, and Piette 2016), and iii) the algorithms proposed in this paper are very efficient.

Consequently, we have generated random tables, varying the tightness of the tables (ratio 'number of tuples in the table' over 'total number of possible tuples') following the discussion in (Perez and Régin 2014).

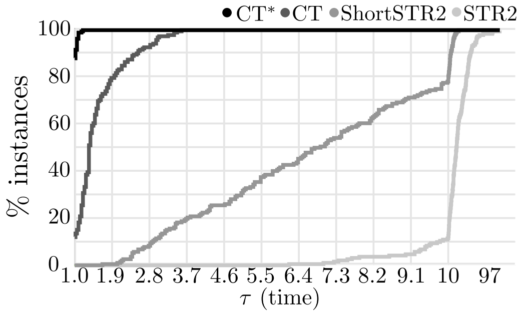

Figure 6: Results on Positive Short Tables

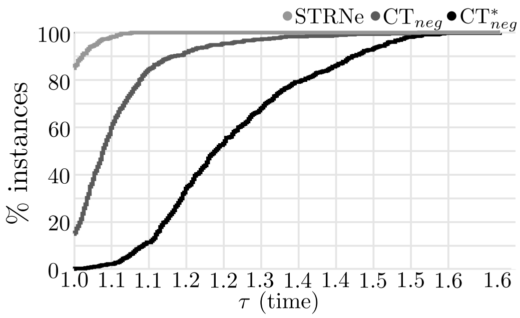

Figure 7: Results on Negative Short Tables – Small Domains

Positive Short Tables

The series we used contains 600 instances, each with 20 variables whose domain sizes range from 5 to 7, and 40 random positive short table constraints of arities 6 or 7, each table having a tightness comprised between 0.5% and 2% and a proportion of short tuples equal to 1%, 5%, 10% and 20%. Figure 6 shows the results obtained on these positive short tables, mainly comparing CT* and ShortSTR2 (Jefferson and Nightingale 2013). Clearly, CT* outperforms ShortSTR2 that is at least 7 times slower than CT* for 50% of the instances. We have also tested CT and STR2 (Lecoutre 2011) on these instances after converting short tables into ordinary tuples. Here, we can can observe that CT* is 2 times faster than CT on 20% of the instances, while saving memory space.

Negative Short Tables

On a first series, generated with the same parameters as above except that negative short tables replace positive short tables, Figure 7 shows that CT$_{neg}$ and CT$^*_{neg}$ are slightly outperformed (at most 1.4 and 1.6 times slower, respectively) by STRNe (Li et al. 2013), which is an adaptation of STR2 for negative tables; for CT$_{neg}$ and STRNe, note that short tables had to be converted into ordinary tables. The second series we used does not involve short tables and contains 45 (more difficult) instances, each with 10 variables whose domain size is 5, and 40 random negative table constraints of arity 6, each table having a tightness of 10%, 20%, . . . , 90%. Figure 8 shows the results we obtained with CT$_{neg}$ and STRNe. We also plot the curve for CT$^*_{neg}$ even if only ordinary tables are present, so as to observe the overhead introduced by the handling of the *-similarity groups. Clearly, CT$_{neg}$ outperforms STRNe that requires at least 3 times more time for around half of the hardest instances. Unlike the previous series that only contains satisfiable instances, about half of the instances of this series are unsatisfiable, making CT$_{neg}$ more suitable in general when the outcome of the problem is not known in advance.

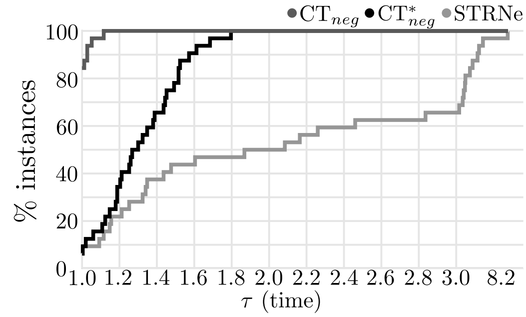

Figure 8: Results on Negative Tables

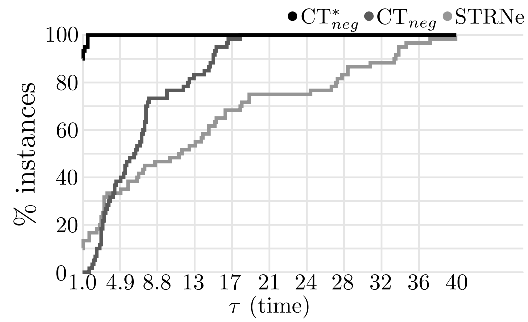

Figure 9: Results on Negative Short Tables – Large Domains

The third series contains 100 instances, each with 3 variables whose domain size is 100, and 40 random negative short table constraints of arity 3, each table having a tightness ranging from 0.5% to 2% and a proportion of short tuples equal to 5%, 10% and 20% (with no overlapping between short tuples). Here, we want to emphasize that CT$^*_{neg}$ can be very efficient, compared to STRNe, when the domain sizes and the number of short tuples are very large. This is visible in Figure 9. Roughly speaking, CT$^*_{neg}$ is about 10 times speedier on average.

Conclusion

In this paper, we have proposed three extensions of the state-of-the-art GAC algorithm for positive table constraints CT. The new algorithms, CT*, CT$_{neg}$ and CT$^*_{neg}$, can handle short tables, negative tables, and negative short tables, respectively. Exploiting bitwise operations, and notably efficient bitwise counting of bits set to 1 in computer words, these algorithms are particularly competitive, as shown by our experiments. We do believe that these algorithms will be adopted by constraint solver developers because short tables will become more and more popular, as they represent a natural and simple modeling mechanism.

References

Bessiere, C., and Régin, J.-C. 1997. Arc consistency for general constraint networks: preliminary results. In *Proceedings of IJCAI'97*, 398–404.

Bliek, C. 1996. Wordwise algorithms and improved heuristics for solving hard constraint satisfaction problems. Technical Report 12-96-R045, ERCIM.

Boussemart, F.; Lecoutre, C.; and Piette, C. 2016. XCSP3: An integrated format for benchmarking combinatorial constrained problems. Technical Report arXiv:1611.03398, CoRR. Available from `http://www.xcsp.org`.

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

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

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

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

Gent, I.; Jefferson, C.; Miguel, I.; and Nightingale, P. 2007. Data structures for generalised arc consistency for extensional constraints. In *Proceedings of AAAI'07*, 191–197.

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

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

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

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

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

Lecoutre, C., and Hemery, F. 2007. A study of residual supports in arc consistency. In *Proceedings of IJCAI'07*, 125–130.

Lecoutre, C., and Szymanek, R. 2006. Generalized arc consistency for positive table constraints. In *Proceedings of CP'06*, 284–298.

Lecoutre, C., and Vion, J. 2008. Enforcing arc consistency using bitwise operations. *Constraint Programming Letters* 2:21–35.

Lecoutre, C.; Boussemart, F.; and Hemery, F. 2003. Exploiting multidirectionality in coarse-grained arc consistency algorithms. In *Proceedings of CP'03*, 480–494.

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

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

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

Li, H.; Liang, Y.; Guo, J.; and Li, Z. 2013. Making simple tabular reduction works on negative table constraints. In *Proceedings of AAAI'13*, 1629–1630.

Likitvivatanavong, C.; Zhang, Y.; Bowen, J.; and Freuder, E. 2004. Arc consistency in MAC: a new perspective. In *Proceedings of CPAI'04 workshop held with CP'04*, 93–107.

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

Michel, L. D., and Van Hentenryck, P. 2012. Constraint satisfaction over bit-vectors. In *Proceedings of CP'12*, 527–543. Springer.

OscaR Team. 2012. OscaR: Scala in OR. Available from `https://bitbucket.org/oscarlib/oscar`.

Perez, G., and Régin, J.-C. 2014. Improving GAC-4 for Table and MDD constraints. In *Proceedings of CP'14*, 606–621.

Pesant, G. 2004. A regular language membership constraint for finite sequences of variables. In *Proceedings of CP'04*, 482–495.

Régin, J.-C. 2011. Improving the expressiveness of table constraints. In *Proceedings of the workshop ModRef'11 held with CP'11*.

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

Van Kessel, P., and Quimper, C.-G. 2012. Filtering algorithms based on the word-RAM model. In *Proceedings of AAAI'12*, 577–583.

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

Wang, W.; Søndergaard, H.; and Stuckey, P. J. 2016. A bit-vector solver with word-level propagation. In *Proceedings of CPAIOR'16*, 374–391. Springer.

Warren, H. 2002. *Hacker's Delight*. Addison-Wesley.

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