Extending Compact-Table to Negative and Short Tables

Hélène Verhaeghe\textsuperscript{1} and Christophe Lecoutre\textsuperscript{2} and Pierre Schaus\textsuperscript{1}
\textsuperscript{1}UCLouvain, ICTEAM, Place Sainte Barbe 2, 1348 Louvain-la-Neuve, Belgium, \{firstname.lastname\}@uclouvain.be
\textsuperscript{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 extensional 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 Régis 1997; Lhomme and Régis 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 Régis 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 Régis 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 Régis 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; Régis 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 \(c\) constraints. Each variable \(x\) has an associated domain, denoted by \(\text{dom}(x)\), 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 \(\text{sep}(c)\), and is semantically defined by a relation, denoted by \(\text{rel}(c)\), which contains the set of tuples allowed for the variables involved in \(c\). The arity of a constraint \(c\) is \(|\text{sep}(c)|\). For simplicity, a variable-value pair \((x,a)\) such that \(x \in \text{sep}(c)\) and \(a \in \text{dom}(x)\) is called a value (of \(c\)). A table constraint \(c\) is a constraint such that \(\text{rel}(c)\) 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 \(\text{vars}(\tau) = \{x_1, x_2, \ldots, x_r\}\). The \(i\)th value of \(\tau\) is denoted by \(\tau[i]\) or
\[
\begin{array}{|c|c|c|}
\hline
x & y & z \\
\hline
\tau_{1a} & e & a \\
\tau_{1b} & e & b \\
\tau_{1c} & e & c \\
\tau_2 & a & b \\
\tau_3 & b & c \\
\hline
\end{array}
\] (a) An ordinary table

\[
\begin{array}{|c|c|c|}
\hline
x & y & z \\
\hline
\tau_1 & c & a \\
\tau_2 & a & b \\
\tau_3 & b & c \\
\hline
\end{array}
\] (b) A short table

Figure 1: Equivalence between ordinary and short tables

\[\tau[x_i], \text{ and } \tau \text{ is valid iff } \forall i \in 1..r, \tau[i] \in \text{dom}(x_i). \tau \text{ is a support (resp., a conflict) on a constraint } c \text{ such that } \text{vars}(\tau) = \text{scp}(c) \text{ iff } \tau \text{ is a valid tuple allowed (resp., disallowed) by } c. \text{ If } \tau \text{ is a support (resp., a conflict) on a constraint } c \text{ involving a variable } x \text{ and such that } \tau[x] = a, \text{ we say that } \tau \text{ is a support for (resp., a conflict for) } (x, a) \text{ 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 \text{scp}(c), \forall a \in \text{dom}(x), \text{there exists at least one support for } (x, a) \text{ on } c. \text{ A CN } N \text{ is GAC iff every constraint of } N \text{ 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 the left an ordinary table, and on the right an equivalent short table, i.e., a table containing short tuples. Here, assuming that \(\text{dom}(y) = \{a, b, c\}, \) 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 \(\tau_1\). A short tuple \(\tau\) is valid iff \(\forall i \in [1..r], \tau[i] = * \lor \tau[i] \in \text{dom}(x_i)\).

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 \(\text{currTable}\) 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 \(i\)th bit of \(\text{currTable}\) is set to 1 if the \(i\)th tuple \(\tau_i\) of the table of \(c\) is currently valid. To help updating dynamically this structure, a bitset called \(\text{supports}[x, a]\) is computed (initially, and never updated) for every value \((x, a)\) of \(c\). Each bit at position \(i\) indicates if the \(i\)th tuple \(\tau_i\) of the table of \(c\) contains \((x, a)\), i.e., is such that \(\tau_i[x] = a\). An illustration is given by Figure 2.

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

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

\[
1 \ 0 \ 1 \lor 0 \ 1 \ 0 = 0 \ 0 \ 0
\]

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 \(\text{scp}\) and \(\text{scp}^\text{up}\)) 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 \(bs.\text{words}\) of computer 64-bit words, with \(bs.\text{length}\) indicating the number of words. Im-

\[
\begin{array}{|c|c|c|}
\hline
\tau_1 & \tau_2 & \tau_3 \\
\hline
\tau_1 & c & a & a \\
\tau_2 & a & b & c \\
\tau_3 & b & c & b \\
\hline
\end{array}
\]

(a) A positive table with 3 tuples

\[
\begin{array}{|c|c|c|}
\hline
\tau_1 & \tau_2 & \tau_3 \\
\hline
\tau_1 & 1 & 1 & 1 \\
\hline
\end{array}
\]

(b) Bitset \(\text{currTable}\)

\[
\begin{array}{|c|c|c|}
\hline
\tau_1 & \tau_2 & \tau_3 \\
\hline
\tau_1 & 0 & 1 & 0 \\
\tau_2 & 0 & 0 & 1 \\
\tau_3 & 1 & 0 & 0 \\
\hline
\end{array}
\]

(c) Bitsets \(\text{supports}\)

Figure 2: Bitsets introduced for CT
importantly, 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. Formally, we 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 \( O(rd\frac{1}{w}) \) 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 \( \text{supports}[x, a] \) 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 \( \text{supports}[x, a] \) whose \( i \)th bit indicates if \((x, a)\) is accepted by the \( i \)th tuple \( \tau_i \) of the table of \( c \), i.e., if \( \tau_i[x] = a \lor \tau_i[x] = \ast \). For the incremental update, we use the bitset \( \text{supports}^{\ast}[x, a] \) whose \( i \)th bit indicates if \((x, a)\) is strictly accepted by the \( i \)th tuple

| \( \tau \) of the table, i.e., if \( \tau[x] = a \). This means that for each occurrence of \( \ast \) in a short tuple, the corresponding bits are always set to 0 in the bitsets \( \text{supports}^{\ast} \). Figure 4 shows an illustration, where bitsets \( \text{supports} \) and \( \text{supports}^{\ast} \) 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:

\[
\text{mask} \leftarrow \text{mask} \land \text{supports}^{\ast}[x, a]
\]

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

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.

---

**Algorithm 1: Class ConstraintCT**

```plaintext
10 Method updateTable()
11   foreach variable \( x \in \text{scp} \) do
12       mask \( \leftarrow 0 \)
13       if \(|\Delta|x| < |\text{dom}(x)|\) then
14           foreach value \( a \in \Delta_x \) do
15               mask \( \leftarrow \text{mask} \land \text{supports}[x, a] \)
16           mask \( \leftarrow \text{mask} \)
17       else
18           foreach value \( a \in \text{dom}(x) \) do
19               mask \( \leftarrow \text{mask} \land \text{supports}[x, a] \)
20       currTable \( \leftarrow \text{currTable} \land \text{mask} \)
21   end
22 Method filterDomains()
23   foreach variable \( x \in \text{scp} \) do
24       foreach value \( a \in \text{dom}(x) \) do
25           if currTable \& \text{supports}[x, a] = 0 then
26               \( \text{dom}(x) \leftarrow \text{dom}(x) \setminus \{a\} \)
27       currTable \( \leftarrow \text{currTable} \land \text{mask} \)
28   end
29 Method enforceGAC()
30   updateTable()
31   if currTable = 0 then
32       return Backtrack
33   filterDomains()
```

---

**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.

<table>
<thead>
<tr>
<th>( \tau )</th>
<th>( \tau_1 )</th>
<th>( \tau_2 )</th>
<th>( \tau_3 )</th>
</tr>
</thead>
<tbody>
<tr>
<td>( (A) )</td>
<td>currentTable</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>( (B) = \text{supports}[y, b] )</td>
<td>0</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>( (C) = \text{supports}[y, c] )</td>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>( (A) &amp; (C) )</td>
<td>1</td>
<td>0</td>
<td>1</td>
</tr>
</tbody>
</table>

(a) Incremental update

<table>
<thead>
<tr>
<th>( \tau )</th>
<th>( \tau_1 )</th>
<th>( \tau_2 )</th>
<th>( \tau_3 )</th>
</tr>
</thead>
<tbody>
<tr>
<td>( (A) = \text{currTable} )</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>( (B) = \text{supports}[y, a] )</td>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>( (C) = \text{supports}[y, c] )</td>
<td>1</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>( (A) &amp; (D) )</td>
<td>1</td>
<td>0</td>
<td>1</td>
</tr>
</tbody>
</table>

(b) Reset-based update

**Figure 4:** Bitsets \( \text{supports} \) and \( \text{supports}^{\ast} \)

---
Dealing with Negative Tables: CT$_\text{neg}$

The modifications brought to CT for dealing with negative tables, i.e., tables containing disallowed tuples, are discussed now. We keep working with the bitmask 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:

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

Algorithm 2: Class ConstraintCT$_\text{neg}$

```java
Method updateTable()
1    foreach variable $x \in \text{scp}$ do
2        mask ← 0
3        if $|\Delta_y| < |\text{dom}(x)|$ then
4            foreach value $a \in \Delta_y$ do
5                mask ← mask $|$ conflicts[$x, a]
6            mask ← "mask"
7        else
8            foreach value $a \in \text{dom}(x)$ do
9                mask ← mask $|$ conflicts[$x, a]
10           currTable ← currTable $&$ mask
11       currTable ← currTable $&$ mask
```

Method filterDomains()

```java
foreach variable $x \in \text{scp}$ do
1        foreach value $a \in \text{dom}(x)$ do
2            if nbls(currTable $&$ conflicts[$x, a]) = $\Pi_{y \in \text{scp}, y \neq x} |\text{dom}(y)|$ then
3                dom($x$) ← dom($x$) \ {a}
4                currTable ← currTable $&$ conflicts[$x, a]
```

Method enforceGAC()

```java
updateTable()
18 if nbls(currTable) = $\Pi_{x \in \text{scp}} |\text{dom}(x)|$ then
19    return Backtrack
20 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 nbls(), 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.

For filtering domains, we verify whether values have still support or not. We call Function nbls() 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 $O(rd^2/k)$ which is the same as CT and CT$_\ast$ 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$_\text{neg}$

We now show how we can extend CT to tables that are both short and negative. There is however one limitation: there cannot be any overlapping between two short tuples. Two short tuples $\tau_1$ and $\tau_2$ overlap iff there is an ordinary tuple that is both subsumed by $\tau_1$ and $\tau_2$. 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 if 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 $\tau_2$ with $\tau_3$ (since they are both ordinary tuples), and of $\tau_1$ with $\tau_2$. 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

2This is due to our need of counting tuples. Overlapping tuples made counting not trivial, and is let as a perspective of this work.
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 ⊥ 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\}, τ₁ and τ₂, 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 i-th 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 i-th bit indicates if the value (x, a) is accepted by the i-th tuple τᵢ of the table of c, i.e., if τᵢ[x] = a \lor τᵢ[x] = e. For the incremental update, we use the bitset conflicts*[x, a] whose i-th bit indicates if (x, a) is strictly accepted by the i-th tuple τᵢ of the table, i.e., if τᵢ[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 ta-

**Algorithm 4**: Function nb1s*(bs: Bitset)

```plaintext
1  cnt ← 0
2  foreach i ∈ 1..bs.length do
3      bc ← Long.bitCount(bs.words[i])
4      cnt ← cnt + bc * nbSubsumedTuples(i)
5  return cnt
```

### Experimental Results

We have implemented all algorithms described in this paper, namely, CT, CT*, CT₁ₙₑgross and CT₁ₙₑgross in the Oscar solver (OscaR Team 2012), using 64-bit words (Long). Our implementation benefits from all optimization techniques described in (Demuulenaere 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 More 2002). A performance profile is a cumulative distribution of the speedup performance of an algorithm s ∈ S compared to other algorithms of S over a set I of instances: ρₛ(τ) = 1/τ × \{|i ∈ I : rᵢ,s ≤ τ\} where the performance ratio is defined as rᵢ,s = tᵢ,s/tᵢ,s with tᵢ,s the time obtained with algorithm s ∈ S on instance i ∈ I. A ratio rᵢ,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 CT₁ₙₑgross 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 Régis 2014).
**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 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* 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 observe that CT* is 2 times faster than CT on 20% of the instances, while saving memory space.

**Negative Tables**

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* 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* 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* and CT* 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


