Compact-Table: Efficiently Filtering Table Constraints with Reversible Sparse Bit-Sets

Jordan Demeulenaere[1], Renaud Hartert[1], Christophe Lecoutre[2], Guillaume Perez[3], Laurent Perron[4], Jean-Charles Régin[3], Pierre Schaus[1]

[1]UCLouvain, Belgium, [2]CRIL, University of Artois [3]University of Nice, France, [4]Google, France,

Abstract. In this paper, we describe Compact-Table (CT), a bitwise algorithm to enforce Generalized Arc Consistency (GAC) on table constraints. Although this algorithm is the default propagator for table constraints in or-tools and OscaR, two publicly available CP solvers, it has never been described so far. Importantly, CT has been recently improved further with the introduction of residues, resetting operations and a data-structure called reversible sparse bit-set, used to maintain tables of supports (following the idea of tabular reduction): tuples are invalidated incrementally on value removals by means of bit-set operations. The experimentation that we have conducted with OscaR shows that CT outperforms state-of-the-art algorithms STR2, STR3, GAC4R, MDD4R and AC5-TC on standard benchmarks.

1 Introduction

Table constraints, also called extension(al) constraints, explicitly express the allowed combinations of values for the variables they involve as sequences of tuples, which are called tables. Table constraints can theoretically encode any kind of constraints and are amongst the most useful ones in Constraint Programming (CP). Indeed, they are often required when modeling combinatorial problems in many application fields. The design of filtering algorithms for such constraints has generated a lot of research effort, see [1,19,17,9,29,13,16,10,26].

Over the last decade, many developments have thus been achieved for enforcing the well-known property called Generalized Arc Consistency (GAC) on binary and/or non-binary extensionally defined constraints. Among successful techniques, we find:

- bitwise operations that allow performing parallel operations on bit vectors. Already exploited during the 70's [28,23], they have been applied more recently to the enforcement of arc consistency on binary constraints [3,18].
- residual supports (residues) that store the last found supports of each value. Initially introduced for ensuring optimal complexity [2], they have been shown efficient in practice [14,20,15] when used as simple sentinels.
- tabular reduction, which is a technique that dynamically maintains the tables of supports. Based on the structure of sparse sets [4,12], variants of Simple Tabular Reduction (STR) have been proved to be quite competitive [29,13,16].

– resetting operations that saves substantial computing efforts in some particular situations. They have been successfully applied to the algorithm GAC4 [26].

In this paper, we introduce a very efficient GAC algorithm for table constraints that combines the use of bitwise operations, residual supports, tabular reduction, and resetting operations. It is called Compact-Table (CT), and originates from or-tools, the Google solver that won the latest MiniZinc Challenges. It is important to note that or-tools does not implement many global constraints, but heavily relies on table constraints instead, with CT as embedding propagator. Through the years, CT has reached a good level of maturity because it has been continuously improved and extended with many cutting edge ideas such as those introduced earlier. Unfortunately, the core algorithm of CT has not been described in the literature so far[1] and is thus seldom used as a reference for practical comparisons. The first version of CT implemented in or-tools, with a bit-set representation of tables, dates back to 2012, whereas the version of CT presented in this paper is exactly the last one implemented in OscaR [25].

Outline. After presenting related works in Section 2, we introduce some technical background in Section 3. Then, we recall in Section 4 usual state restoration mechanisms implemented in CP solvers, and describe reversible sparse bit-sets in Section 5. In Section 6, we describe our algorithm CT. Before concluding, we present in Section 7 the results of an experimentation we have conducted with CT and its contenders on a large variety of benchmarks.

2 Related Work

Propagators for table constraints are filtering procedures used to enforce GAC. Given the importance of table constraints, it is not surprising that much research has been carried out in order to find efficient propagators. This section briefly describes the most efficient ones.

Generic Algorithms. On the one hand, GAC3 is a classical general-purpose GAC algorithm [21] for non-binary constraints. Each call to this algorithm for a constraint requires testing if each value is still supported by a valid tuple accepted by the constraint. Several improvements to fasten the search for a support gave birth to variants such as GAC2001 [2] and GAC3^{rm} [15]. Unfortunately, the worst-case time complexity of all these algorithms grows exponentially with the arity of the constraints. On the other hand, GAC4 [24] is a value-based algorithm, meaning here that for each value, it maintains a set of valid tuples supporting it. Each time a value is removed, all supporting tuples are removed from the associated sets, which allows us to identify values without any more supports. GAC4R is a recent improvement of GAC4 [26], which recomputes the sets of supporting tuples from scratch when it appears to be less costly than updating them based on the removed values.

[1] Note that some parts of this paper were published in a Master Thesis report [6].

AC5 Instantiations. In [10], Mairy et al. introduce several instantiations of the generic AC5 algorithm for table constraints, the best of them being AC5-TCOptSparse. This algorithm shares some similarities with GAC4 since it precomputes lists of supporting tuples which allows us to retrieve efficiently new supports by iterating over these lists. Note that a reversible integer is used to indicate the current position of a support in each list. This algorithm is implemented in Comet, and has been shown to be efficient on ternary and quaternary constraints.

Simple Tabular Reduction. STR1 [29] and STR2 [13] are coarse-grained algorithms that globally enforce GAC by traversing the constraint tables while dynamically maintaining them: each call to the algorithm for a constraint removes the invalid tuples from its table. The improvements brought in STR2 avoid unnecessary operations by considering only relevant subsets of variables when checking the validity of a tuple, and collecting supported values. Contrary to its predecessors, STR3 [16] is a fine-grained (or value-based) algorithm. For each value, it initially computes a static array of tuples supporting it, and keeps a reversible integer *curr* that indicates the position of the last valid tuple in the array. STR3 also maintains the set of valid tuples. STR3 is shown to be complementary to STR2, being more efficient when the tables are not reduced drastically during search.

Compressed Representations. Other algorithms gamble on the compression of tables to reduce the time needed to ensure GAC. The most promising data structure allowing a more compact representation is the Multi-valued Decision Diagram (MDD) [27]. Two notable algorithms using MDDs as main data structure are `mddc` [5] and MDD4R [26]. The former does not modify the decision diagram and performs a depth-first search of the MDD during propagation to detect which parts of the MDD are consistent or not. MDD4R dynamically maintains the MDD by deleting nodes and edges that do not belong to a solution. Each value is matched with its corresponding edges in the MDD, so, when a value has none of its edges present in the MDD, it can be removed.

3 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 $|scp(c)|$, i.e., the number of variables involved in c. A (positive) *table constraint* c is a constraint such that $rel(c)$ is defined explicitly by listing the tuples that are allowed by c.

Example 1. The constraint $x \neq y$ with $x \in \{1, 2, 3\}$ and $y \in \{1, 2\}$ can be alternatively defined by the table constraint c such that $scp(c) = \{x, y\}$ and $rel(c) = \{(1, 2), (2, 1), (3, 1), (3, 2)\}$. We also write:

$$\langle x, y\rangle \in T \quad \text{with} \quad T = \langle(1, 2), (2, 1), (3, 1), (3, 2)\rangle$$

Let $\tau = (a_1, a_2, \dots, a_r)$ be a tuple of values associated with an ordered set of variables $X = \{x_1, x_2, \dots, x_r\}$. The ith value of τ is denoted by $\tau[i]$ or $\tau[x_i]$. The tuple τ is valid iff $\forall i \in 1..r, \tau[i] \in dom(x_i)$. An r-tuple τ is a *support* on the r-ary constraint c iff τ is a valid tuple that is 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* (x, a) on c. Generalized Arc Consistency (GAC) is a well-known domain-filtering consistency defined as follows:

Definition 1. *A constraint c is* generalized arc consistent *(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 a constraint. Many algorithms have been devised for establishing GAC according to the nature of the constraints. For table constraints, STR [29] is such an algorithm: it removes invalid tuples during search of supports using a sparse set data structure which separates valid tuples from invalid ones. This method of seeking supports improves search time by avoiding redundant tests on invalid tuples that have already been detected as invalid during previous GAC enforcements. STR2 [13], an optimization of STR, limits some basic operations concerning the validity of tuples and the identification of supports, through the introduction of two important sets called S^{sup} and S^{val} (described later).

4 Reversible Objects and Implementation Details

Trail and Timestamping. The issue of storing related states of the solving process is essential in CP. In many solvers[2], a general mechanism is used for doing and undoing (on backtrack) the current state. This mechanism is called a trail and it was first introduced in [8] for implementing non-deterministic search. A trail is a stack of pairs $(location, value)$ where *location* stands for any piece of memory (e.g., a variable), which can be restored when backtracking. Typically, at each search node encountered during the solving process, the constraint propagation algorithm is executed. A same filtering procedure (propagator) can thus be executed several times at a given node. Consequently, if one is interested in storing some information concerning a filtering procedure, the value of a same memory location can be changed several times. However, stamping that is part of the "folklore" of programming [11] can be used to avoid storing a same memory location on the trail more than once per search node. The idea behind timestamping is that only the final state of a memory location is relevant for

[2] One notable exception is Gecode, a copy-based solver.

its restoration on backtrack. The trail contains a general time counter that is incremented at each search node, and a timestamp is attached to each memory location indicating the time at which its last storage on the trail happened. If a memory location changes and its timestamp matches the current time of the trail then there is no need to store it again. CP solvers generally expose some "reversible" objects to the users using this trail+timestamping mechanism. The most basic one is the reversible version of primitive types such as int or long values. In the following, we denote by `rint` and `rlong` the reversible versions of `int` and `long` primitive types.

Reversible Sparse Sets. Reversible primitive types can be used to implement more complex data structures such as reversible sets. It was shown in [12] how to implement a reversible set using a single `rint` that represents the current size (limit) of the set. In this structure, which is called reversible sparse set, an array of size n is used to store the permutation from 0 to $n-1$. All values in this permutation array at indices smaller than or equal to a variable *limit* are considered as part of the set, while the others are considered as removed. When iterating on current values of the set (with decreasing indices from *limit* to 0), the value at the current index can be removed in $O(1)$ by just swapping it with the value stored at *limit* and decrementing *limit*. Making a sparse set reversible just requires managing a single `rint` for *limit*. On backtrack, when the limit is restored, all concerned removed values are restored in $O(1)$.

Domains and Deltas. In OscaR [25], the implementation of domains relies on reversible sparse sets. One advantage of implementing domains with this structure is that one can easily retrieve the set of values removed from a domain between any two calls to a given filtering procedure. All we need to store in the filtering procedure is the last size of the domain. The delta set (set of values removed between the two calls) is composed of all the values located between the current size and the last recorded size. More details on this cheap mechanism to retrieve the delta sets can be found in [12].

5 Reversible Sparse Bit-Sets

This section describes the class `RSparseBitSet` that is the main data structure for our algorithm to maintain the supports. In what follows, when we refer to an array t, t[0] denotes the first element (indexing starts at 0) and t.length the number of its cells (size).

The class `RSparseBitSet`, which encapsulates four fields and 6 methods, is given in Algorithm 1. One important field is **`words`**, an array of p 64-bit words (actually, reversible long integers), which defines the current value of the bit-set: the ith bit of the jth word is 1 iff the $(j-1) \times 64 + i$th element of the (initial) set is present. Initially, all words in this array have all their bits at 1, except for the last word that may involve a suffix of bits at 0. For example, if we want to handle a set initially containing 82 elements, then we build an array with $p = \lceil 82/64 \rceil = 2$ words that initially looks like:

Algorithm 1: Class RSparseBitSet

```
words: array of rlong                                   // words.length = p
index: array of int                                     // index.length = p
limit: rint
mask: array of long                                      // mask.length = p
Method isEmpty(): Boolean
    return limit = −1
Method clearMask()
    foreach $i$ from 0 to limit do
        offset ← index[$i$]
        mask[offset] ← $0^{64}$
Method reverseMask()
    foreach $i$ from 0 to limit do
        offset ← index[$i$]
        mask[offset] ← ~mask[offset]                       // bitwise NOT
Method addToMask(m: array of long)
    foreach $i$ from 0 to limit do
        offset ← index[$i$]
        mask[offset] ← mask[offset] | $m$[offset]          // bitwise OR
Method intersectWithMask()
    foreach $i$ from limit downto 0 do
        offset ← index[$i$]
        $w$ ← words[offset] & mask[offset]                 // bitwise AND
        if $w \neq$ words[offset] then
            words[offset] ← $w$
            if $w = 0^{64}$ then
                index[$i$] ← index[limit]
                index[limit] ← offset
                limit ← limit − 1
Method intersectIndex(m: array of long): int
    /* Post: returns the index of a word where the bit-set
       intersects with m, -1 otherwise                             */
    foreach $i$ from 0 to limit do
        offset ← index[$i$]
        if words[offset] & $m$[offset] $\neq 0^{64}$ then
            return offset
    return −1
```

`words`: 11111111111111111111111111111111 11111111111111111100000000000000

Because, in our context, only non-zero words (words having at least one bit set to 1) are relevant when processing operations on the bit-set, we rely on the sparse-set technique by managing in an array `index` the indices of all words: the indices of all non-zero words are in `index` at positions less than or equal to the value of a variable `limit`, and the indices of all zero-words are in `index` at positions strictly greater than `limit`. For our example, we initially have:

> `words`: 11111111111111111111111111111111 11111111111111111100000000000000
> `index`: 0 1
> `limit` : 1

If we suppose now that the 66 first elements of our set above are removed, we obtain:

> `words`: 00000000000000000000000000000000 00111111111111111100000000000000
> `index`: 1 0
> `limit`: 0

The class invariant describing the state of a reversible sparse bit-set is the following:

- `index` is a permutation of $[0, \ldots, p-1]$, and
- $\texttt{words}[\texttt{index}[i]] \neq 0^{64} \Leftrightarrow i \leq \texttt{limit}, \forall i \in 0..p-1$

Note that the reversible nature of our object comes from 1) an array of reversible long (denoted `rlong`) (instead of simple longs) to store the bit words, and 2) the reversible prefix size of non-zero words by using a reversible int (`rint`).

A `RSparseBitSet` also contains a kind of local temporary array, called `mask`. Is is used to collect elements with Method addToMask(), and can be cleared and reversed too. A `RSparseBitSet` can only be modified by means of the method intersectWithMask() which is an operation used to intersect with the elements collected in `mask`. An illustration of the usage of these methods is given in next example.

`words`	1	0	1	0	1	1	1	1
addToMask	1	1	1	0	1	0	0	0
addToMask	0	0	0	1	0	0	0	1
`mask`	1	1	1	1	1	0	0	1
intersectWithMask	1	0	1	0	1	0	0	1

Fig. 1: `RSparseBitSet` example

Example 2. Figure 1 illustrates the use of Methods addToMask() and intersectWithMask(). We assume that the current state of the bit-set is given by the value of `words`, and that clearMask() has been called such that `mask` is initially empty. Then two bit-sets are collected in `mask` by calling addToMask(). The value of `mask` is represented after these two operations. Finally intersectWith-Mask() is executed and the new value of the bit-set `words` is given at the last row of Figure 1.

We now describe the implementation of the methods in `RSparseBitSet`. Method isEmpty() simply checks if the number of non-zero words is different from zero (if the limit is set to -1, it means that all words are non-zero). Method clearMask() sets to 0 all words of `mask` corresponding to non-zero words of `words`, whereas Method reverseMask() reverses all words of `mask`. Method addToMask() applies a word by word logical bit-wise *or* operation. Once again, notice that this operation is only applied to words of `mask` corresponding to non-zero words of `words`. Method intersectMask() considers each non-zero word of `words` in turn and replaces it by its intersection with the corresponding word of `mask`. In case the resulting new word is zero, it is swapped with the last non-zero word and the value of `limit` is decremented. Finally, Method intersectIndex() checks if a given bit-set (array of longs) intersects with the current bit-set: it returns the index of the first word where an intersection can be proved, -1 otherwise.

6 Compact-Table (CT) Algorithm

As STR2 and STR3, Compact-Table (CT) is a GAC algorithm that dynamically maintains the set of valid supports regarding the current domain of each variable. The main difference is that CT is based on an object `RSparseBitSet`. In this set, each tuple is indexed by the order it appears in the initial table. Invalid tuples are removed during the initialization as well as values that are not supported by any tuple. The class `ConstraintCT`, Algorithm 2, allows us to implement any positive table constraint c while running the CT algorithm to enforce GAC.

6.1 Fields

As fields of Class `ConstraintCT`, we first find `scp` for representing the scope of c and `currTable` for representing the current table of c by means of a reversible sparse bit-set. If $\langle \tau_0, \tau_1, \ldots, \tau_{p-1} \rangle$ is the initial table of c, then `currTable` is a `RSparseBitSet` object (of initial size p) such that the value i is contained (is set to 1) in the bit-set if and only if the ith tuple is valid:

$$i \in \texttt{currTable} \Leftrightarrow \forall x \in scp(c), \tau_i[x] \in dom(x)$$

We also have three fields $\texttt{S}^{\texttt{val}}$, $\texttt{S}^{\texttt{val}}$ and `lastSizes` in the spirit of STR2. Indeed, as in [13], we introduce two sets of variables, called $\texttt{S}^{\texttt{val}}$ and $\texttt{S}^{\texttt{sup}}$. The set $\texttt{S}^{\texttt{val}}$ contains uninstantiated variables (and possibly, the last assigned variable) whose domains have been reduced since the previous invocation of the filtering algorithm on c. To set up $\texttt{S}^{\texttt{val}}$, we need to record the domain size of each modified variable x right after the execution of CT on c: this value is recorded in `lastSizes`$[x]$. The set $\texttt{S}^{\texttt{sup}}$ contains uninstantiated variables (from the scope of the constraint c) whose domains contain each at least one value for which a support must be found. These two sets allow us to restrict loops on variables to relevant ones.

We also have a field `supports` containing static data. During the set up of the table constraint c, CT also computes a static array of words $\texttt{supports}[x, a]$, seen as a bit-set, for each variable-value pair (x, a) where $x \in scp(c) \wedge a \in dom(x)$: the bit at position i in the bit-set is 1 if and only if the tuple τ_i in the initial table of c is a support for (x, a).

T	x	y	z
0	a	a	a
1	a	a	b
2	a	b	c
3	b	a	a
	a	c	b
4	a	b	b
5	b	a	b
6	b	b	a
7	b	b	b

(a) The indexed tuples

`currTable`	1	1	1	1	1	1	1	1
$\texttt{supports}[x, a]$	1	1	1	0	1	0	0	0
$\texttt{supports}[x, b]$	0	0	0	1	0	1	1	1
$\texttt{supports}[y, a]$	1	1	0	1	0	1	0	0
$\texttt{supports}[y, b]$	0	0	1	0	1	0	1	1
$\texttt{supports}[y, d]$	0	0	0	0	0	0	0	0
$\texttt{supports}[z, a]$	1	0	0	1	0	0	1	0
$\texttt{supports}[z, b]$	0	1	0	0	1	1	0	1
$\texttt{supports}[z, c]$	0	0	1	0	0	0	0	0

(b) The corresponding bit-sets

Fig. 2: Illustration of the data structures after the initialization of $\langle x, y, z \rangle \in T$. The tuple (a, c, b) will not be indexed and d will be removed from $dom(y)$.

Example 3. Figure 2 shows an illustration of the content of those bit-sets after the initialization of the following table constraint $\langle x, y, z \rangle \in T$, with:

- $dom(x) = \{a, b\}$, $dom(y) = \{a, b, d\}$, $dom(z) = \{a, b, c\}$
- $T = \langle (a, a, a), (a, a, b), (a, b, c), (b, a, a), (a, c, b), (a, b, b), (b, a, b), (b, b, a), (b, b, b) \rangle$

The tuple (a, c, b) is initially invalid because $c \notin dom(y)$, and thus will not be indexed. Value d will be removed from $dom(y)$ given that it is not supported by any tuple.

Finally, we have an array `residues` such that for each variable-value pair (x, a), $\texttt{residues}[x, a]$ denotes the index of the word where a support was found for (x, a) the last time one was sought for.

6.2 Methods

The main method in `ConstraintCT` is enforceGAC(). After the initialization of the sets $\texttt{S}^{\texttt{val}}$ and $\texttt{S}^{\texttt{sup}}$, CT updates `currTable` to filter out (indices of) tuples that are no more supports, and then considers each variable-value pair to check whether these values still have a support.

Algorithm 2: Class ConstraintCT

```
scp: array of variables                                  // Scope
currTable: RSparseBitSet                                 // Current table
$S^{val}$, $S^{sup}$                                     // Temporary sets of variables
lastSizes    // lastSizes[$x$] is the last size of the domain of $x$
supports     // supports[$x, a$] is the bit-set of supports for $(x, a)$
residues     // residues[$x, a$] is the last found support for $(x, a)$
Method updateTable()
    foreach variable $x \in S^{val}$ do
        currTable.clearMask()
        if $|\Delta_x| < |dom(x)|$ then                  // Incremental update
            foreach value $a \in \Delta_x$ do
                currTable.addToMask(supports[$x, a$])
            currTable.reverseMask()
        else                                             // Reset-based update
            foreach value $a \in dom(x)$ do
                currTable.addToMask(supports[$x, a$])
        currTable.intersectWithMask()
        if currTable.isEmpty() then
            break
Method filterDomains()
    foreach variable $x \in S^{sup}$ do
        foreach value $a \in dom(x)$ do
            index $\leftarrow$ residues[$x, a$]
            if currTable.words[index] & supports($x, a$)[index] $= 0^{64}$ then
                index $\leftarrow$ currTable.intersectIndex(supports[$x, a$])
                if index $\neq -1$ then
                    residues[$x, a$] $\leftarrow$ index
                else
                    $dom(x) \leftarrow dom(x) \setminus \{a\}$
        lastSize[$x$] $\leftarrow |dom(x)|$
Method enforceGAC()
    $S^{val} \leftarrow \{x \in$ scp $: |dom(x)| \neq$ lastSize[$x$]$\}$
    foreach variable $x \in S^{val}$ do
        lastSize[$x$] $\leftarrow |dom(x)|$
    $S^{sup} \leftarrow \{x \in$ scp $: |dom(x)| > 1\}$
    updateTable()
    if currTable.isEmpty() then
        return Backtrack
    filterDomains()
```

Updating the Current Table For each variable $x \in \mathtt{S}^{\mathtt{val}}$, i.e., each variable x whose domain has changed since the last time the filtering algorithm was called, updateTable() performs some operations. This method assumes an access to the set of values Δ_x removed from $dom(x)$. There are two ways of updating `currTable`, either incrementally or from scratch after resetting. Note that the idea of using resets has been proposed in [26] and successfully applied to GAC4 and MDD4, with the practical interest of saving computational effort in some precise contexts. This is the strategy implemented in updateTable(), by considering a reset-based computation when the size of the domain is smaller than the number of deleted values.

In case of an incremental update (line 10), the union of the tuples to be removed is collected by calling addToMask() for each bit-set (of supports) corresponding to removed values, whereas in case of a reset-based update (line 14), we perform the union of the tuples to be kept. To get a mask ready to apply, we just need to reverse it when it has been built from removed values. Finally, the (indexes of) tuples of `currTable` not contained in the mask, built from x, are directly removed by means of intersectWithMask(). When there is no more tuple in the current table, a failure is detected, and updateTable() is stopped by means of a loop break.

Filtering of Domains Values are removed from the domain of some variables during the search of a solution, which can lead to inconsistent values in the domain of other variables. As `currTable` is a reversible and dynamically maintained structure, the value of some bits changes from 1 to 0 when tuples become invalid (or from 0 to 1 when the search backtracks). On the contrary, the `supports` bit-sets are only computed at the creation of the constraint and are not maintained during search. It follows from the definition of those bit-sets that (x, a) has a valid support if and only if

$$(\mathtt{currTable} \cap \mathtt{supports}[x, a]) \neq \emptyset \qquad (1)$$

Therefore, each time a tuple becomes invalid, the constraint must check this condition for every variable value pair (x, a) such that $a \in dom(x)$, and remove a from $dom(x)$ if the condition is not satisfied any more. This operation is efficiently implemented in filterDomains() with the help of residues and the method intersectIndex().

Example 4. The same set of tuples as in Example 3 is considered. Suppose now that a was removed from $dom(x)$ (by another constraint) after the initialization. Given that the domain of x is reduced, when updateTable() is called by enforceGAC(), all tuples supporting a (because $\Delta_x = \{a\}$) will be invalidated. Figure 3a illustrates the intermediary bit-sets used to compute the new value $\mathtt{currTable}^{out}$ from $\mathtt{currTable}^{in}$ and $\mathtt{supports}[x, a]$. Then filterDomains() computes for each variable-value pair (x_i, a_i) (with $x_i \in \mathtt{S}^{\mathtt{sup}}$ and $a_i \in dom(x)$) the intersection of its associated set of supports with `currTable` as shown in Figure 3b. Given that the intersection for $\mathtt{supports}[z, c]$ and `currTable` is empty, c is removed from $dom(z)$.

`currTable`^{in}	1	1	1	1	1	1	1	1
`supports`$[x, a]$	1	1	1	0	1	0	0	0
`currTable`^{out}	0	0	0	1	0	1	1	1

(a) updateTable() invalidates tuples supporting (x, a)

`currTable`	0	0	0	1	0	1	1	1
`supports`$[x, b] \cap$ `currTable`	0	0	0	1	0	1	1	1
`supports`$[y, a] \cap$ `currTable`	0	0	0	1	0	1	0	0
`supports`$[y, b] \cap$ `currTable`	0	0	0	0	0	0	1	1
`supports`$[z, a] \cap$ `currTable`	0	0	0	1	0	0	1	0
`supports`$[z, b] \cap$ `currTable`	0	0	0	0	0	1	0	1
`supports`$[z, c] \cap$ `currTable`	0	0	0	0	0	0	0	0

(b) filterDomains() removes c from $dom(z)$

Fig. 3: Illustration of enforceGAC() after the removal of a from $dom(x)$.

6.3 Improvements

The algorithm in Section 6.2 can be improved to avoid unnecessary computations in some cases.

Filtering out bounded variables. The initialization of $\mathtt{S}^{\mathtt{val}}$ at line 32 can be only performed from unbound variables (and the last assigned variable), instead from the whole scope. We can maintain them in a reversible sparse set.

Last modified variable. It is not necessary to attempt to filter values out from the domain of a variable x if this was the only modified variable since the last call to enforceGAC(). Indeed, when updateTable() is executed, the new state of `currTable` will be computed from $dom(x)$ or Δ_x only. Because every value of x had a support in `currTable` the last time the propagator was called, we can omit filtering $dom(x)$ by initially removing x from $\mathtt{S}^{\mathtt{sup}}$.

7 Experiments

We experimented CT on 1,621 CSP instances involving (positive) table constraints (15GB of uncompressed files in format XCSP 2.1). This corresponds to a large variety of instances, taken from 37 series. For guiding search, we used binary branching with *domain over degree* as variable ordering heuristic and *min value* as value ordering heuristic. A timeout of 1,000 seconds was used for each instance. The tested GAC algorithms are CT, STR2 [13], STR3 [16], GAC4 [24,26], GAC4R [26], MDDR [26] and AC5TCRecomp [22]. All scripts, codes and benchmarks allowing to reproduce our experiments are available at `https://bitbucket.org/pschaus/xp-table`. The experiments were run on a 32-core machine (1400MHz cpu) with 100GB using Java(TM) SE Runtime Environment (build 1.8.0_60-b27) with 10GB of memory allocated (-Xmx option).

Performance Profiles. Let $t_{p,s}$ represent the time obtained with filtering algorithm $s \in S$ on instance $p \in P$. The performance ratio is defined as follows:

$r_{p,\,s} = \frac{t_{p,\,s}}{\min\{t_{p,\,s^*} | s^* \in S\}}$, where s^* denotes the fastest algorithm. A ratio $r_{p,\,s} = 1$ means that s was the fastest on p. The performance profile [7] is a cumulative distribution function of the performance of s compared to other algorithms:

$$\rho_s(\tau) = \frac{1}{|P|} \times |\{p \in P | r_{p,\,s} \leq \tau\}|$$

Our results are visually aggregated to form a performance profile in Figure 4 generated by means of the online tool `http://sites.uclouvain.be/performance-profile`. Note that we filtered out the instances that i) could not be solved within 1,000 seconds by all algorithms ii) were solved in less than 2 seconds by the slowest algorithm, and iii) required less than 500 backtracks. The final set of instances used to build the profile is composed of 227 instances. An interactive performance profile is also available at `https://www.info.ucl.ac.be/~pschaus/assets/publi/performance-profile-ct` to let the interested reader deactivate some family of instances to analyze the results more closely.

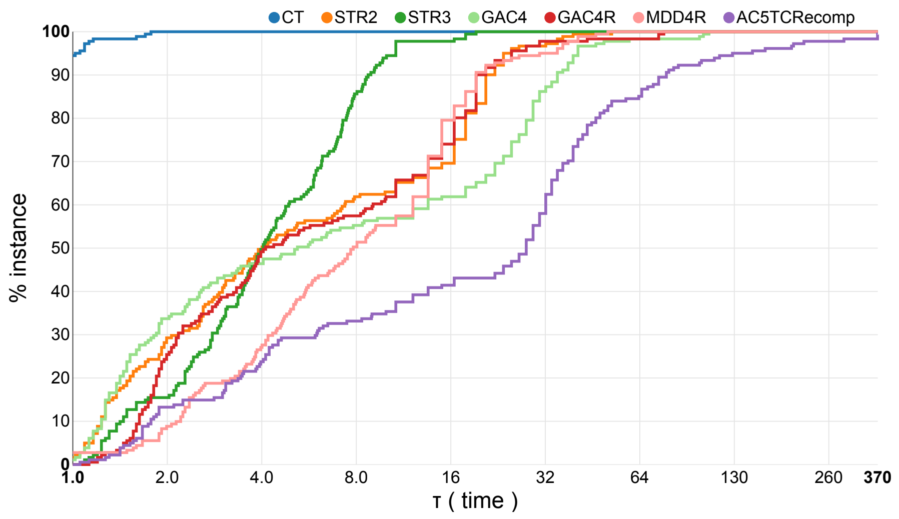

Fig. 4: Performance Profile

Table 1 reports the speedup statistics of CT over the other algorithms. A first observation is that CT is the fastest algorithm on 94.47% of the instances. Among all tested algorithms, AC5TCRecomp obtains the worse results. Then it is not clear which one among STR2, STR3, GAC4 and GAC4R is the second best algorithm. Based on the average speedup, STR3 seems to be the second best algorithm followed by STR2, MDD4R and GAC4R. Importantly, one can observe that the speedup of CT over the best of the other algorithms is about 3.77 on average.

Speedup	STR2	STR3	GAC4	GAC4R	MDD4R	AC5-TC	Best2
average	9.11	5.07	15.59	11.37	10.38	50.40	3.77
min	0.76	1.09	0.92	1.13	0.13	1.05	0.13
max	88.58	51.04	173.24	208.52	50.84	1850.82	15.99
std	10.64	4.36	19.67	18.57	9.46	134.13	2.87

Table 1: Speedup analysis of CT over the other algorithms. Column 'Best2' corresponds to a virtual second best solver (by considering the minimum time taken by all algorithms except CT).

Impact of Resetting Operations. In Algorithm 2, the choice of being incremental or not, when updating `currTable`, depends on the size of several sets and is thus dynamic. We propose to analyze two variants of Algorithm 2 when this choice is static:

– Full incremental (CTI): only the body of the 'if' at line 10 is executed (deltas are systematically used).
– Full re-computation (CTR) : only the body of the 'else' at line 14 is executed (domains are systematically used).

The performance profiles with these two variants are given in Figure 5, and the speedup table of the static versions over the dynamic one is given in Table 2.

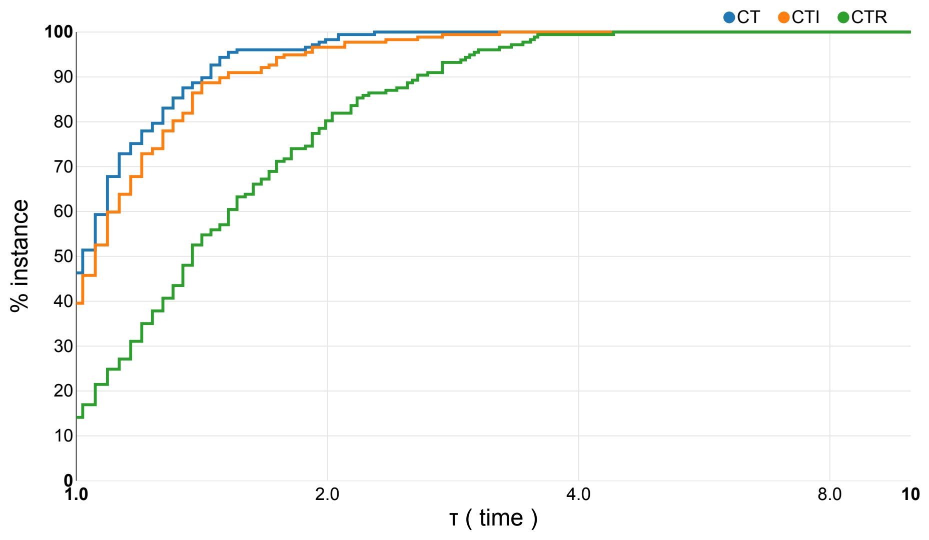

Fig. 5: Performance Profiles with dynamic (CT), recomputation (CTR) and incremental (CTI) strategies.

Speedup	CTI	CTR	Best
average	1.09	1.46	1.02
min	0.44	0.53	0.44
max	3.23	4.39	1.96
std	0.38	0.65	0.27

Table 2: Speedup analysis of the two static variants over CT.

As can be seen from both the performance profiles and the speedup table, the dynamic version using the resetting operations as introduced in [26] dominates the static ones. The average speedup is around 9% over CTI and 46% over CTR.

Contradiction with Previous Results. In [22], AC5TCRecomp was presented as being competitive with STR2. When we analyzed the code[3] of STR2 used in [22], it appeared that STR2 was implemented in Comet using built-in sets (triggering the garbage collection of Comet). We thus believe that the results and conclusions in [22] may over-penalize the performance of STR2. Our results also somehow contradicts the results in [26] where STR3 and STR2 were dominated by MDD4R and GAC4R. When analyzing the performance of the implementation of STR2 and STR3 used in [26] with or-tools, it appears that it is not as competitive as that in AbsCon (sometimes slower by a factor of 3). The results presented in [26] may thus also over-penalize the performances of STR2 and STR3.

One additional contribution of this work is a fined-tuned implementation of the best filtering algorithms for table constraints. The implementation of all these algorithms in OscaR was optimized, and checked to be close in performance to the ones by the original authors. For CT, STR2 and STR3, a comparison was made with AbsCon, and for CT, MDD4R and GAC4R, a comparison was made with or-tools. Our implementation required a development effort of 10 man-months in order to obtain an efficient implementation of each algorithm. It involved the expertise of several OscaR developers and a deep analysis of the existing implementations in AbsCon and or-tools. The implementation of all table algorithms used in this paper is open-source and available in OscaR release 3.1.0.

8 Conclusion

In this paper, we have shown that Compact-Table (CT) is a robust algorithm that clearly dominates state-of-the-art propagators for table constraints. CT benefits from well-tried techniques: bitwise operations, residual supports, tabular reduction and resetting operations. We believe that CT can be easily implemented using the reversible sparse bit-set data structure.

[3] available at `http://becool.info.ucl.ac.be`

References

1. C. Bessiere and J.-C. Régin. Arc consistency for general constraint networks: preliminary results. In *Proceedings of IJCAI'97*, pages 398–404, 1997.
2. C. Bessiere, J.-C. Régin, R. Yap, and Y. Zhang. An optimal coarse-grained arc consistency algorithm. *Artificial Intelligence*, 165(2):165–185, 2005.
3. C. Bliek. Wordwise algorithms and improved heuristics for solving hard constraint satisfaction problems. Technical Report 12-96-R045, ERCIM, 1996.
4. P. Briggs and L. Torczon. An efficient representation for sparse sets. *ACM Letters on Programming Languages and Systems*, 2(1-4):59–69, 1993.
5. K. Cheng and R. Yap. An MDD-based generalized arc consistency algorithm for positive and negative table constraints and some global constraints. *Constraints*, 15(2):265–304, 2010.
6. J. Demeulenaere. Efficient algorithms for table constraints. Technical report, Master Thesis, under the supervision of P. Schauss, UCLouvain, 2015.
7. E.D. Dolan and J.J. Moré. Benchmarking optimization software with performance profiles. *Mathematical programming*, 91(2):201–213, 2002.
8. R.W. Floyd. Nondeterministic algorithms. *Journal of the ACM*, 14(4):636–644, 1967.
9. I.P. Gent, C. Jefferson, I. Miguel, and P. Nightingale. Data structures for generalised arc consistency for extensional constraints. In *Proceedings of AAAI'07*, pages 191–197, 2007.
10. P. Van Hentenryck J.-B. Mairy and Y. Deville. Optimal and efficient filtering algorithms for table constraints. *Constraints*, 19(1):77–120, 2014.
11. D.E. Knuth. *The Art of Computer: Combinatorial Algorithms*, volume 4. Addison-Wesley, 2015.
12. V. le Clément de Saint-Marcq, P. Schaus, C. Solnon, and C. Lecoutre. Sparse-sets for domain implementation. In *Proceeding of TRICS'13*, pages 1–10, 2013.
13. C. Lecoutre. STR2: Optimized simple tabular reduction for table constraints. *Constraints*, 16(4):341–371, 2011.
14. C. Lecoutre, F. Boussemart, and F. Hemery. Exploiting multidirectionality in coarse-grained arc consistency algorithms. In *Proceedings of CP'03*, pages 480–494, 2003.
15. C. Lecoutre and F. Hemery. A study of residual supports in arc consistency. In *Proceedings of IJCAI'07*, pages 125–130, 2007.
16. C. Lecoutre, C. Likitvivatanavong, and R. Yap. STR3: A path-optimal filtering algorithm for table constraints. *Artificial Intelligence*, 220:1–27, 2015.
17. C. Lecoutre and R. Szymanek. Generalized arc consistency for positive table constraints. In *Proceedings of CP'06*, pages 284–298, 2006.
18. C. Lecoutre and J. Vion. Enforcing arc consistency using bitwise operations. *Constraint Programming Letters*, 2:21–35, 2008.
19. O. Lhomme and J.-C. Régin. A fast arc consistency algorithm for n-ary constraints. In *Proceedings of AAAI'05*, pages 405–410, 2005.
20. C. Likitvivatanavong, Y. Zhang, J. Bowen, and E.C. Freuder. Arc consistency in MAC: a new perspective. In *Proceedings of CPAI'04 workshop held with CP'04*, pages 93–107, 2004.
21. A.K. Mackworth. Consistency in networks of relations. *Artificial Intelligence*, 8(1):99–118, 1977.
22. J.-B. Mairy, P. van Hentenryck, and Y. Deville. An optimal filtering algorithm for table constraints. In *Proceedings of CP'12*, pages 496–511, 2012.

23. J.J. McGregor. Relational consistency algorithms and their application in finding subgraph and graph isomorphisms. *Information Sciences*, 19:229–250, 1979.
24. R. Mohr and G. Masini. Good old discrete relaxation. In *Proceedings of ECAI'88*, pages 651–656, 1988.
25. OscaR Team. OscaR: Scala in OR, 2012. Available from `https://bitbucket.org/oscarlib/oscar`.
26. G. Perez and J.-C. Régin. Improving GAC-4 for Table and MDD constraints. In *Proceedings of CP'14*, pages 606–621, 2014.
27. A. Srinivasan, T. Kam, S. Malik, and R.K. Brayton. Algorithms for discrete function manipulation. In *Proceedings of ICCAD'90*, pages 92–95, 1990.
28. J.R. Ullmann. An algorithm for subgraph isomorphism. *Journal of the ACM*, 23(1):31–42, 1976.
29. J.R. Ullmann. Partition search for non-binary constraint satisfaction. *Information Science*, 177:3639–3678, 2007.