Constraints
DOI 10.1007/s10601-011-9107-6

STR2: optimized simple tabular reduction for table constraints

Christophe Lecoutre

Abstract Table constraints play an important role within constraint programming. Recently, many schemes or algorithms have been proposed to propagate table constraints and/or to compress their representation. In this paper, we describe an optimization of simple tabular reduction (STR), a technique proposed by J. Ullmann to dynamically maintain the tables of supports when generalized arc consistency (GAC) is enforced/maintained. STR2, the new refined GAC algorithm we propose, allows us to limit the number of operations related to validity checking and search of supports. Interestingly enough, this optimization makes simple tabular reduction potentially r times faster where r is the arity of the constraint(s). The results of an extensive experimentation that we have conducted with respect to random and structured instances indicate that STR2 is usually around twice as fast as the original STR, two or three times faster than the approach based on the hidden variable encoding, and can be up to one order of magnitude faster than previously state-of-the-art (generic) GAC algorithms on some series of instances. When comparing STR2 with the more recently developed algorithm based on multi-valued decision diagrams (MDDs), we show that both approaches are rather complementary.

Keywords Table constraints · Extensional constraints · Generalized arc consistency (GAC)

1 Introduction

Arc Consistency (AC) plays a central role in Constraint Programming (CP). It is a property of constraint networks that can be used to identify and remove

This paper is an extension of [24].

C. Lecoutre (✉)
CRIL–CNRS UMR 8188, Université Lille-Nord de France, Artois,
rue de l'université, SP 16, 62307 Lens, France
e-mail: lecoutre@cril.fr

Published online: 30 March 2011

some inconsistent values, i.e. values which cannot lead to any solution. It is an essential component of the Maintaining Arc Consistency (MAC) algorithm, which is commonly used to solve binary instances of the Constraint Satisfaction Problem (CSP). It is also at the heart of stronger consistencies that have recently received some attention such as, e.g., singleton arc consistency [3, 25], weak k-singleton arc consistency [42] and conservative dual consistency [26].

For non-binary constraints, which arise naturally in many applications, Generalized AC (GAC) replaces AC. Instead of using GAC extensions of generic AC algorithms, efficiency may be improved by exploiting the semantics/structure of the constraints. Indeed, enforcing GAC is NP-hard [4] and the best worst-case time complexity [2] that can be obtained with a generic GAC algorithm is $O(erd^r)$ where e denotes the number of constraints, d the greatest domain size and r the greatest constraint arity.

This paper is concerned with efficient GAC algorithms for table constraints. Here the word *table* means the same thing as *extensional* except that table constraints are usually non-binary. A table constraint is defined by explicitly listing the tuples that are either allowed or disallowed for the variables of its scope. In the former case, the table constraint is said to be *positive* while in the latter case, it is said *negative*. Table constraints are also sometimes referred to as *ad-hoc* (non-binary) constraints [11].

Table constraints arise naturally in configuration problems where they represent available combinations of options. For some applications, compatibilities (or incompatibilities) between resources, e.g. persons or machines, can be expressed in tables. For example, for use in the selection of k persons to form a working group, a table may enumerate possible associations according to certain abilities while taking into account a (subjective) agreement criterion. Another example is that in some puzzles, e.g. crosswords, non-binary constraints can only be expressed extensionally. Tabular data may also come from databases: the results of database queries are sometimes expressed as tables that have large arity. It is well known (e.g. see [20]) that there are strong theoretical connections between relational database theory and constraint satisfaction.

Table constraints are important in constraint programming because they are easily handled by end-users of constraint systems. For simplicity reasons, an inexperienced user sometimes specifies extensional constraints although some of these should preferably be intensional. It is crucial to handle such extensional constraints as efficiently as possible, ideally as though their semantics were known. Furthermore, because any constraint can theoretically be expressed in tabular form (although this may lead to a time and space explosion), tables provide a universal way of representing constraints.

Some recent research articles have focused on theoretical and practical aspects of table constraints. As a result, there are many new ways to represent table constraints and to enforce generalized arc consistency on them. One line of research aims to combine the two concepts of validity and acceptability of tuples of values, using indexing structures [17, 28, 30]. Another line focuses on compact representations using data structures such as tries [17], multi-valued decision diagrams [10, 12, 13], compressed tables [23] and deterministic finite automatas [35]. Significant formal and practical results have been obtained with respect to the very classical schemes. Another recent proposal, called simple tabular reduction (STR) [41], significantly differs from previous methods: the principle is to dynamically maintain tables in order to only keep supports.

Our contribution in this paper is two-fold. First, we introduce an optimization of STR which allows us to limit the number of validity checking and support search operations. Interestingly, we show that the new refined GAC algorithm we propose, called STR2, makes simple tabular reduction potentially r times faster where r is the arity of the constraint(s). It means that the algorithm we propose is particularly adapted to table constraints of large arity. Second, we present the results of an extensive experimentation that we have conducted including both random and structured CSP instances. These results confirm that when GAC is maintained on table constraints during search, STR2 usually outperforms algorithms previously identified as state-of-the-art.

The paper is organized as follows. After some technical background in Section 2, we present in Section 3 the classical and recent GAC algorithms for table constraints. Then, we introduce STR in Section 4 as well as its refined version, STR2, in Section 5. Next, in Section 6, we experimentally show the good behavior of STR2, when compared to STR and classical GAC schemes (including GAC-valid+allowed, a robust algorithm). Some natural variants of STR2 are presented in Section 7. Finally, before concluding, in Section 8, we experimentally compare STR2 with binary encoding approaches as well as the compression-based approach based on multi-valued decision diagrams (MDDs).

2 Background

A (discrete) constraint network (CN) P is composed of a finite set of n variables, denoted by $vars(P)$, and a finite set of e constraints, denoted by $cons(P)$. Each variable x has an associated domain, denoted by $dom(x)$, that contains the finite set of values that can be assigned to x. The maximum domain size for a given CN will be denoted by d. Each constraint c involves an ordered set of variables, called the *scope* of c and denoted by $scp(c)$. It is 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 the size of $scp(c)$, and will usually be denoted by r. A *binary* constraint involves exactly 2 variables, and a *non-binary constraint* strictly more than 2 variables.

A solution to a constraint network is an assignment of a value to each variable such that all the constraints are satisfied. A constraint network is said to be *satisfiable* iff it admits at least one solution. The Constraint Satisfaction Problem (CSP) is the NP-hard task of determining whether a given constraint network is satisfiable or not. Thus, a CSP instance is defined by a constraint network which is solved either by finding a solution or by proving unsatisfiability. In many cases, a CSP instance can be solved by using a combination of search and inferential simplification [14].

A central example of inferential simplification is enforcement of GAC (Generalized Arc Consistency), which removes certain inconsistent values. Values are inconsistent iff they cannot occur in any solution. In some cases, enforcement of GAC can prove that no solutions exist, without any search. Before giving a technical definition of GAC, we introduce the notion of support. Given an ordered set $\{x_1, \ldots, x_i, \ldots, x_r\}$ of r variables and a r-tuple $\tau = (a_1, \ldots, a_i, \ldots, a_r)$ of values, the individual value a_i will be denoted by $\tau[x_i]$.

Definition 1 Let c be an r-ary constraint. An r-tuple τ is *valid* on c iff $\forall x \in scp(c)$, $\tau[x] \in dom(x)$. The set of valid tuples on c is $val(c) = \Pi_{x \in scp(c)} dom(x)$.

Recall that a tuple τ is *allowed* by a constraint c iff $\tau \in rel(c)$. Supports are defined as follows.

Definition 2 Let c be an r-ary constraint. An r-tuple τ is a *support* on c iff τ is a valid tuple on c which 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; we also say that (x, a) is supported by c. To simplify discourse in this paper, we define *v-values* and *c-values*. A v-value of a constraint network P is a variable-value pair (x, a) such that $x \in vars(P)$ and $a \in dom(x)$. A c-value of a constraint network P is a constraint-variable-value triplet (c, x, a) such that $c \in cons(P)$, $x \in scp(c)$ and $a \in dom(x)$.

Definition 3 Let P be a constraint network.

- A v-value (x, a) of P is *generalized arc-consistent* on P iff for every constraint c of P involving x, there exists a support for (x, a) on c.
- A constraint c is *generalized arc-consistent* iff $\forall x \in scp(c), \forall a \in dom(x)$, there exists a support for (x, a) on c.
- A constraint network P is *generalized arc-consistent* iff every constraint of P is generalized arc-consistent.

A v-value (x, a) that is generalized arc-consistent is also said to be *GAC-consistent*. If a v-value (x, a) is not generalized arc-consistent, it is said to be *GAC-inconsistent*. It is easy to see that a GAC-inconsistent value cannot occur in any solution and is therefore (globally) inconsistent. Enforcing GAC means making the constraint network GAC-consistent by removing GAC-inconsistent values (from domains). Many algorithms are available for enforcing GAC (or for enforcing AC when the constraints are binary) [2].

In this paper, we shall also refer to MAC (Maintaining Arc Consistency) which is a complete algorithm considered to be among the most efficient generic approaches for the solution of CSP instances. MAC [37] explores the search space depth-first, backtracks when dead-ends occur, and enforces (generalized) arc consistency after each decision taken (variable assignment or value refutation) during search. The *level* of a node v in the search tree developed by MAC is the number of variable assignments performed along the path leading from the root of the search tree to v. A *past* variable is (explicitly) assigned whereas a *future* variable is not (explicitly) assigned. Finally, we emphasize that when GAC is enforced at a given step of the search, values are only removed from domains of future variables.

A positive table constraint is a constraint given in extension and defined by a set of allowed tuples. The set of allowed tuples associated with a positive table constraint c is denoted by *table*$[c]$. This set is represented by an array of tuples indexed from 1 to *table*$[c]$*.length* which denotes the size of the table (i.e. the number of allowed tuples).

To record this set, the worst-case space complexity is $O(tr)$ where $t = table[c].length$ and r is the arity of c. Sometimes, we are interested in the list of allowed tuples that include a v-value (x, a). We can provide for every c-value (c, x, a) the *sub-table* $table[c, x, a]$ of allowed tuples involving (x, a), from $table[c]$. This is an array whose indices ranges from 1 to $table[c, x, a].length$ such that any element $table[c, x, a][i]$ gives the position (index) in $table[c]$ of the i^{th} allowed tuple involving (x, a). Thus sub-tables are indexing structures.

3 GAC algorithms for table constraints

3.1 Classical GAC schemes

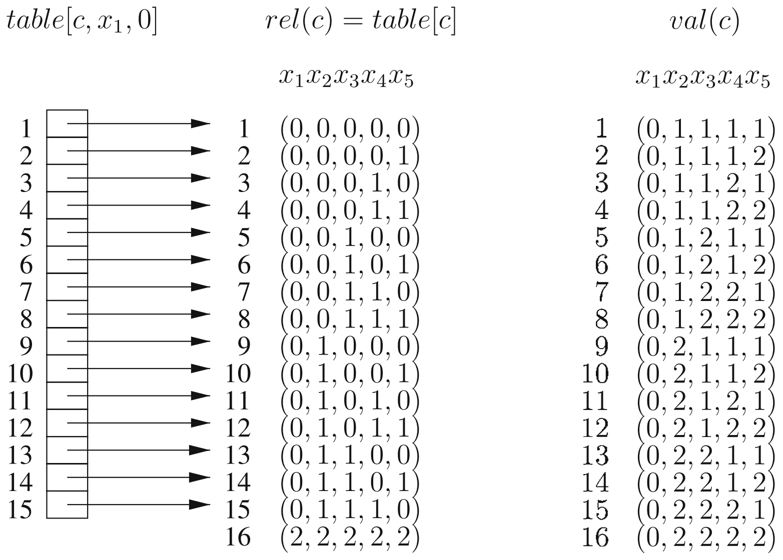
There are two different ways in which a GAC algorithm can seek a support. The support-seeking scheme called *GAC-valid* iterates over valid tuples until an allowed one is found. The other natural support-seeking scheme, which is called *GAC-allowed*, iterates over allowed tuples until a valid one is found. Roughly speaking, GAC-valid and GAC-allowed correspond respectively to GAC-scheme-predicate and GAC-scheme-allowed in [6].

Unfortunately, visiting only the lists of valid tuples or the lists of allowed tuples can be quite expensive. This is why many alternatives, presented later, have been developed. In the following example, which illustrates potential drawbacks of both classical schemes, a constraint c involves r variables $x_1, ..., x_r$ such that the domain of each variable is initially $\{0, 1, 2\}$. Suppose that exactly 2^{r-1} tuples are allowed by c: these correspond to the binary representation of all values between 0 and $2^{r-1} - 2$ together with the tuple $(2, 2, ..., 2, 2)$, as illustrated in Fig. 1a with $r = 5$. Suppose also that, due to propagation caused by other constraints, the domains of all variables have been reduced to $\{1, 2\}$ except for the variable x_1 whose domain has been reduced to $\{0\}$. After this propagation, there are exactly 2^{r-1} valid tuples that can be built for c, as illustrated in Fig. 1b with $r = 5$.

Now consider checking whether there is a support for $(x_1, 0)$ on c. Using GAC-valid, the time complexity of determining that $(x_1, 0)$ has no support on c is $\Omega(2^{r-1})$ because 2^{r-1} valid tuples are processed. GAC-allowed has also time complexity $\Omega(2^{r-1})$ because it reviews $2^{r-1} - 1$ allowed tuples to prove that $(x_1, 0)$ has no support on c. The behavior of both schemes is unsatisfactory because it is immediate that $(x_1, 0)$ is GAC-inconsistent.

In [28], a refinement that combines GAC-valid and GAC-allowed without any additional data structure is introduced: visits to lists of valid and allowed tuples are alternated. The idea is to jump over sequences of valid tuples containing no allowed tuple and to jump over sequences of allowed tuples containing no valid tuple.

For example, when seeking a support for $(x_1, 0)$ on the constraint c in Fig. 1, this refined scheme starts by finding, in $O(r)$, the first valid tuple $\tau = (0, 1, ..., 1, 1)$. Next, the first allowed tuple τ' greater than or equal to τ is sought. When dichotomic search is used here, this involves $log_2(2^{r-1})$ comparisons of tuples, which is $O(r^2)$ because comparing two tuples is $O(r)$. As no such tuple exists for $(x_1, 0)$, $(x_1, 0)$ is proven to be GAC-inconsistent. Note that this refined scheme, which is called GAC-valid+allowed, is able to skip a number of tuples that grows exponentially with the arity of the constraints, but in a manner different to that of indexing approaches



$table[c, x_1, 0]$	$rel(c) = table[c]$	$x_1x_2x_3x_4x_5$	$val(c)$	$x_1x_2x_3x_4x_5$
1	1	$(0,0,0,0,0)$	1	$(0,1,1,1,1)$
2	2	$(0,0,0,0,1)$	2	$(0,1,1,1,2)$
3	3	$(0,0,0,1,0)$	3	$(0,1,1,2,1)$
4	4	$(0,0,0,1,1)$	4	$(0,1,1,2,2)$
5	5	$(0,0,1,0,0)$	5	$(0,1,2,1,1)$
6	6	$(0,0,1,0,1)$	6	$(0,1,2,1,2)$
7	7	$(0,0,1,1,0)$	7	$(0,1,2,2,1)$
8	8	$(0,0,1,1,1)$	8	$(0,1,2,2,2)$
9	9	$(0,1,0,0,0)$	9	$(0,2,1,1,1)$
10	10	$(0,1,0,0,1)$	10	$(0,2,1,1,2)$
11	11	$(0,1,0,1,0)$	11	$(0,2,1,2,1)$
12	12	$(0,1,0,1,1)$	12	$(0,2,1,2,2)$
13	13	$(0,1,1,0,0)$	13	$(0,2,2,1,1)$
14	14	$(0,1,1,0,1)$	14	$(0,2,2,1,2)$
15	15	$(0,1,1,1,0)$	15	$(0,2,2,2,1)$
	16	$(2,2,2,2,2)$	16	$(0,2,2,2,2)$

(a) The list of allowed tuples (b) The list of valid tuples

Fig. 1 Constraint c is such that $scp(c) = \{x_1, x_2, x_3, x_4, x_5\}$ and $rel(c) = table[c]$ contains 2^4 (allowed) tuples, as shown. Currently, $dom(x_1) = \{0\}$ and $\forall i \in 2..5, dom(x_i) = \{1, 2\}$, so $val(c)$ contains 2^4 (valid) tuples

presented in the next section. GAC-valid+allowed can be implemented using binary search or instead using tries.

It is important to note that any GAC algorithm can be used within these schemes. It may be the basic GAC3 [32, 33], but it may also be other coarse-grained generalized arc consistency algorithms such as e.g. GAC2001 [7] and GAC3^{rm} [27].

3.2 Indexing and compression

In this section, we present recent approaches to enforce generalized arc consistency on table constraints and/or to compress their representation.

Some of these approaches associate auxiliary functions/structures with tables. The idea is to associate with each tuple of each table some pointers to next tuples involving particular values. This is an index structure for use in seeking supports. The first indexing approach [30] combines both the concept of "acceptability" (the fact that a tuple is accepted by a constraint) and the concept of validity (the fact that each value in a tuple is valid). A function, called nextIn, indicates for each c-value (c, x, a) and each tuple τ in $table[c]$, the smallest tuple in $table[c]$ that is greater than or equal to τ (according to the lexicographic order) and that contains (x, a). A data structure, called nextDiff here, in a second indexing-based approach [17] allows us to find for each positive table constraint c, for each tuple τ in $table[c]$ and for each variable $y \in scp(c)$, the next tuple in $table[c]$ with a value for y different from $\tau[y]$. Although attractive, these indexing approaches are considered to be outperformed by compression-based approaches [13, 17].

We now briefly introduce four different approaches to the reduction of space required by tables. Roughly speaking, significant reduction of space turns out to reduce running time for enforcing generalized arc consistency. The key success factor is basically the compression ratio achieved when tables are represented by sophisticated data structures such as tries, multi-valued decision diagrams, compressed tables or deterministic finite automata.

In addition to the nextDiff indexing approach mentioned earlier, Gent et al. [17] have used tries to represent and propagate table constraints. A *trie* [16] is a rooted tree used to store and retrieve strings over an alphabet. A trie can represent a large dictionary because a trie has only one node for each common prefix. The table of an r-ary constraint c can be represented by a trie in which successive levels are associated with successive variables in the scope of c. At each level, the alphabet is the domain of the associated variable. At the leaf level we have a special terminal node $\boxed{\text{t}}$. All root-to-leaf paths are of uniform length since all tuples are composed of exactly r elements. In [17], the authors propose to specifically exploit tries to look for supports.

Starting with a trie, which is an arc-labelled rooted tree that eliminates prefix redundancy, we can eliminate shared suffixes [10, 12] to obtain a *multi-valued decision diagram* (MDD), which is an arc-labelled *directed acyclic graph* (DAG). In the special case where all domains are binary we obtain a *binary decision diagram* (BDD) instead of an MDD. An MDD has at least one root node and has exactly two terminal nodes. One of these is $\boxed{\text{t}}$. Although there is a clear advantage of using MDDs in terms of space complexity, enforcing generalized arc consistency requires new filtering procedures that must be shown to be effective. Available algorithms [10, 13] that enforce generalized arc consistency using MDDs are not revision-based. This means that instead of seeking a support for each value in turn, GAC is enforced globally on each constraint. A depth-first exploration of the MDD identifies all values that must be removed from domains in order to enforce GAC.

The use of so-called *compressed tuples* [23] can also reduce the amount of memory required for tables. A compressed tuple can be defined as follows: A *compressed tuple* Γ for an r-ary constraint c is an r-tuple $(D_1, \ldots, D_r)$ such that $D_1 \times \cdots \times D_r \subseteq \Pi_{x \in scp(c)} dom(x)$. Informally, a compressed table is *minimal* iff it is not possible to merge two compressed tuples from the table. Minimal disjoint compressed tables can be generated by a method [23] based on constructing decision trees. Because the problem of constructing a decision tree with minimum average branch length is NP-hard, Katsirelos and Walsh have heuristically selected at each construction step a decision used to expand the tree. A fine-grained implementation of GAC based on compressed tables is proposed in [23].

Finally, let us mention the global constraint called regular [35]: the sequence of values taken by the successive variables in the scope of this constraint must belong to a given regular language. For such constraints, a deterministic finite automaton (DFA) can be used to determine whether or not a given tuple is accepted. This is an attractive approach when constraint relations can be naturally represented by regular expressions in a known regular language. For example, in rostering problems, regular expressions can represent valid patterns of activities. Working with constraints defined by a DFA, Pesant's filtering algorithm [35] enforces generalized arc consistency by means of a two-stage forward-backward exploration. This two-stage process constructs a layered directed multi-graph and collects the set of states that support each v-value (x, a).

4 Simple tabular reduction

To enforce GAC on positive table constraints, *simple tabular reduction* (STR) is another approach introduced by Ullmann [41] which significantly differs from previous methods in that it dynamically maintains the tables of allowed tuples. More precisely, whenever a value is removed from the domain of a variable, all tuples that have become invalid are removed from tables. This facilitates identification and removal of values that are no longer GAC-consistent. GAC is enforced while removing invalid tuples; only supports are kept in tables.

Although STR can be applied stand-alone, we now present it in the more general context of a backtrack search algorithm. Indeed, an important feature of STR is the cheap restoration of its structures when backtracking occurs. The principle of STR is to split each table into different sets such that each tuple is a member of exactly one set. One of these sets contains all tuples that are currently valid (and are therefore supports): tuples in this set constitute the content of the *current table*. Any tuple of the current table of a constraint c is called a *current tuple* of c. Other sets contain tuples removed at different levels of search.

The following arrays provide access to the disjoint sets within $table[c]$:

- $position[c]$ is an array of size $t = table[c].length$ that provides indirect access to the tuples of $table[c]$. At any given time the values in $position[c]$ are a permutation of $\{1, 2, \ldots, t\}$. The i^{th} tuple of c is $table[c][position[c][i]]$.
- $currentLimit[c]$ is the position of the last current tuple in $table[c]$. The current table of c is composed of exactly $currentLimit[c]$ tuples. The values in $position[c]$ at indices ranging from 1 to $currentLimit[c]$ are positions of the current tuples of c.
- $levelLimits[c]$ is an array of size $n + 1$ such that $levelLimits[c][p]$ is the position of the first invalid tuple of $table[c]$ removed when the search was at level p (the level corresponds to the number of instantiated or past variables). $levelLimits[c][p] = -1$ if none was removed at level p. If p is the current search level and $levelLimits[c][p] \neq -1$, all tuples removed at level p can be accessed using indices at locations in array $position[c]$ ranging from $currentLimit[c] + 1$ to $levelLimits[c][p]$.

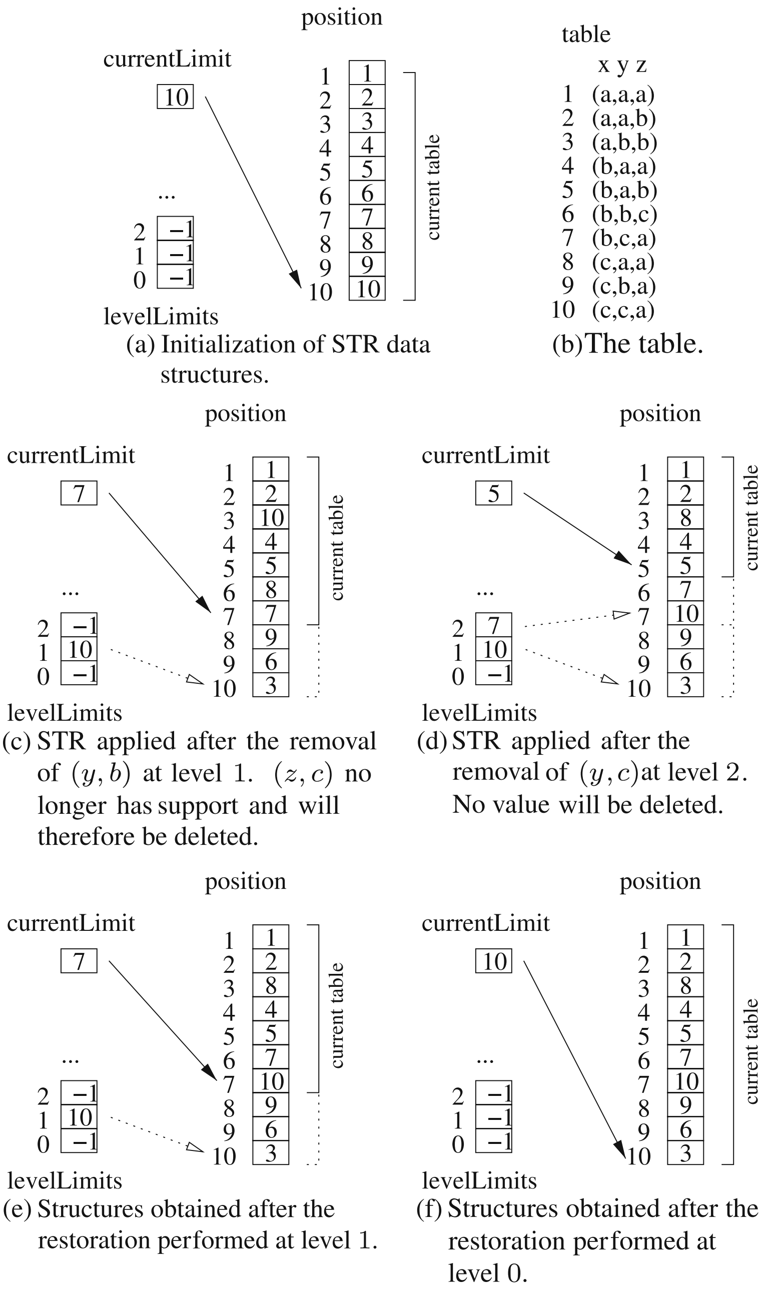
Note that the array $levelLimits[c]$ is indexed from 0 to n (although we usually have array indexing from 1). If the search is preceded by preprocessing then we find at level 0 the tuples removed after the initial call to STR during preprocessing (i.e. before search). The structure $levelLimits$ is not required if there is no search. The structures introduced here,[1] following [9], are simpler than those presented in [24, 41] but complexities remain the same.

To illustrate their use, the following example has a positive table constraint c_{xyz} such that:

- $scp(c_{xyz}) = \{x, y, z\}$
- $rel(c_{xyz}) = \{$
  $(a, a, a), (a, a, b), (a, b, b), (b, a, a), (b, a, b),$
  $(b, b, c), (b, c, a), (c, a, a), (c, b, a), (c, c, a)$
  $\}$

[1] I would like to thank Hadrien Cambazard for suggesting me such a simplification.

Figure 2a shows the STR data structures initialized for the ternary constraint c_{xyz}, with its table given in Fig. 2b. Now suppose that at level 1 (that is to say, after a first variable assignment), (y, b) is deleted by propagation (using other constraints) and



(a) Initialization of STR data structures.

table

	x y z
1	(a,a,a)
2	(a,a,b)
3	(a,b,b)
4	(b,a,a)
5	(b,a,b)
6	(b,b,c)
7	(b,c,a)
8	(c,a,a)
9	(c,b,a)
10	(c,c,a)

(b)The table.

(c) STR applied after the removal of (y, b) at level 1. (z, c) no longer has support and will therefore be deleted.

(d) STR applied after the removal of (y, c)at level 2. No value will be deleted.

(e) Structures obtained after the restoration performed at level 1.

(f) Structures obtained after the restoration performed at level 0.

Fig. 2 Illustration of STR on a ternary positive table constraint c_{xyz}

STR is applied on c_{xyz}. Tuples at position 3, 6 and 9 in $table[c_{xyz}]$ are no longer valid: their locations in array $position[c_{xyz}]$ are swapped with locations of three valid tuples. Locations of tuples that are not valid are now at the end of the array $position[c_{xyz}]$. $levelLimits[c_{xyz}][1]$ is initialized with the old value of $currentLimits[c_{xyz}]$, namely 10, as shown in Fig. 2c. Moreover, (z, c) is deleted because it is no longer supported by any current tuple of c_{xyz}. After a second variable assignment, the removal of (y, c) by propagation and the application of STR, the situation is as shown in Fig. 2d. Suppose now that the search backtracks to level 1. By modifying two pointers (in constant time), we can restore the structures so that tuples removed at level 2 are now included in the current table, as shown in Fig. 2e. Finally, if the search algorithm backtracks to level 0, we obtain the situation shown in Fig. 2f. Tuples in the current table in Fig. 2f are not ordered as initially in Fig. 2a, but for STR this is not a problem. This idea of semantic backtracking, which does not require exact restoration, was used in CLP(R) [22, 43].

Algorithm 1: STR(c: constraint): set of variables

```
Input: $c$ is a constraint (of the constraint network $P$ to be solved)
Output: the set of variables in $scp(c)$ with reduced domain
1  foreach variable $x \in scp(c) \mid x \notin past(P)$ do
2      $gacValues[x] \leftarrow \emptyset$
3  $i \leftarrow 1$
4  while $i \leq currentLimit[c]$ do
5      $index \leftarrow position[c][i]$
6      $\tau \leftarrow table[c][index]$
7      if isValidTuple($c, \tau$) then
8          foreach variable $x \in scp(c) \mid x \notin past(P)$ do
9              if $\tau[x] \notin gacValues[x]$ then
10                 $gacValues[x] \leftarrow gacValues[x] \cup \{\tau[x]\}$
11         $i \leftarrow i + 1$
12     else
13         removeTuple($c, i, |past(P)|$)        // $currentLimit[c]$ decremented
   // domains are now updated and $X_{evt}$ computed
14 $X_{evt} \leftarrow \emptyset$
15 foreach variable $x \in scp(c) \mid x \notin past(P)$ do
16     if $gacValues[x] \subset dom(x)$ then
17         $dom(x) \leftarrow gacValues[x]$
18         if $dom(x) = \emptyset$ then
19             throw INCONSISTENCY
20         $X_{evt} \leftarrow X_{evt} \cup \{x\}$
21 return $X_{evt}$
```

Algorithm 2: isValidTuple(c: constraint, τ: tuple): Boolean

```
Input: $c$ is a constraint
Input: $\tau$ is a tuple whose validity must be checked
Output: $true$ iff $\tau$ is valid on $c$
1  foreach variable $x \in scp(c)$ do
2      if $\tau[x] \notin dom(x)$ then
3          return false
4  return true
```

Algorithm 3: removeTuple(c: constraint, i, p: integers)

```
Input: $c$ is a constraint
Input: $i$ is the position of the tuple to be removed
Input: $p$ is the current level (number of past variables)
1 if $levelLimits[c][p] = -1$ then
2     $levelLimits[c][p] \leftarrow currentLimit[c]$
3 $tmp \leftarrow position[c][i]$
4 $position[c][i] \leftarrow position[c][currentLimit[c]]$
5 $position[c][currentLimit[c]] \leftarrow tmp$
6 $currentLimits[c] \leftarrow currentLimit[c] - 1$
```

Algorithm 4: restoreTuples(c: constraint, p: integer)

```
Input: $c$ is a constraint
Input: $p$ is the level at which tuples must be restored
1 if $levelLimits[c][p] \neq -1$ then
2     $currentLimit[c] \leftarrow levelLimits[c][p]$
3     $levelLimits[c][p] \leftarrow -1$
```

Corresponding to each variable x, we provide a set $gacValues[x]$ [40] that will contain all values in $dom(x)$ which are proved to have a support when GAC is enforced on a constraint c. For STR, Algorithm 1 is a filtering procedure that establishes generalized arc consistency on positive table constraints. The loops at lines 1, 8 and 15 only iterate over uninstantiated variables because it is only possible (and it is sufficient) to remove values from domains of these variables; the set $past(P)$ denotes the set of variables of the constraint network P (to be solved) explicitly instantiated by the search algorithm MAC. The sets $gacValues$ are emptied at lines 1 and 2 of Algorithm 1 because no value is initially guaranteed to be GAC-consistent. Then the loop at lines 4–13 successively processes all current tuples of the table of c. When a tuple τ is proved to be valid (see Algorithm 2), we know that it is necessarily a support since it is (by definition) allowed; values that have been proved to be GAC-consistent are collected at lines 8 to 10. In constant time at line 13 an invalid tuple τ is removed (see Algorithm 3), from the current table: actually it is located at the end of the current table before the value of $currentLimit[c]$ is decremented. If this tuple is the first removed at the current level p, then the current limit is recorded in $levelLimits[c][p]$. Note that τ is effectively removed without actually being moved in memory. After all current tuples have been considered, unsupported values are removed (lines 14 to 21): these are the values in $dom(x) \setminus gacValues[x]$. The test $gacValues[x] \subset dom(x)$ at line 16 is (in our context) equivalent to $|gacValues[x]| \neq |dom(x)|$, which is performed in constant time provided that the size of sets are managed. If the domain of a variable x becomes empty then an exception is thrown at line 19. Otherwise, the set of variables reduced by STR is returned so that these "events" can be propagated to other constraints.

For a given constraint c, the worst-case time complexity of STR, Algorithm 1, is $O(r'd + rt')$ where $r' = |scp(c) \setminus past(P)|$ denotes the number of uninstantiated variables in $scp(c)$ and t' denotes the size of the current table of c. The loops at lines 1, 4 and 15 are $O(r')$, $O(rt')$ and $O(r'd)$, respectively. The worst-case space complexity of STR is $O(n + rt)$ per constraint since $levelLimits$ is $O(n)$, $table$ is $O(rt)$ and $position$ is $O(t)$.

It is well known that values must be restored to domains when backtracking occurs. After this restoration, tuples that were invalid may now be valid. If a tuple τ was removed from the current table of c at level p, then τ must be restored to the current table of c when the search backtracks to level $p-1$. In our implementation, tuples are restored by calling Algorithm 4 which puts the set of invalid tuples removed at the given level into the current table, at the tail end. Restoration is achieved in constant time (for each constraint) without traversing either set and without moving any tuple in memory [41].

Simple Tabular Reduction dynamically maintains the list (table) of current supports for each constraint. This makes it conceptually related to other approaches that maintain a compact form of current supports such as those based on automata [35] or decision diagrams [10, 13]. All these approaches aim at globally enforcing generalized arc consistency by traversing a unique data structure. However, STR is the only one that deals with the original list of allowed tuples. STR is also related to the algorithm GAC4 [34] that can be considered as a filtering algorithm for table constraints because all allowed tuples are explicitly stored during an initialization phase. The data structures used by GAC4 are a propagation queue containing deleted values (not yet processed) and for each c-value (c, x, a) a set containing the current supports for (x, a) on c. A major difference between GAC4 and STR is that the former is guided by (deleted) values whereas the latter globally enforces generalized arc consistency. Finally, there exist several binary encodings of non-binary constraint networks : the dual encoding [15], the hidden variable encoding [36] and the double encoding [39]. In these different encodings, each non-binary constraint c becomes a *dual* variable x_c whose domain is the set of tuples allowed by c. Maintaining the list of supports of c in STR is thus equivalent to maintaining the list of valid values of x_c in a binary encoding. Because of this similarity, we have decided to conduct an experimentation (presented in Section 8.2).

5 STR2

It is possible to improve STR in two directions, which yields an optimized approach called STR2. First, as soon as all values in the domain of a variable have been detected GAC-consistent, it is futile to continue to seek supports for values of this variable. We therefore introduce a set, S^{sup}, of uninstantiated variables in $scp(c)$ whose domain contains at least one value for which a support has not yet been found. In STR2, Algorithm 5, which is an optimized version of STR, lines 1 and 7 initialize S^{sup} to be the same as $scp(c) \setminus past(P)$. If $|gacValues[x]| = |dom(x)|$ at line 19 then all values of $dom(x)$ are supported, so line 20 removes x from S^{sup}. Efficiency is gained by iterating only over variables in S^{sup} at lines 16 and 25.

The second direction of improvement avoids unnecessary validity operations. At the end of an invocation of STR for constraint c, we know that for every variable $x \in scp(c)$, every tuple τ such that $\tau[x] \notin dom(x)$ has been removed from the current table of c. If there is no backtrack and $dom(x)$ does not change between this invocation and the next invocation, then at the time of the next invocation it is certainly true that $\tau[x] \in dom(x)$ for every tuple τ in the current table of c. In this case, there is no need to check whether $\tau[x] \in dom(x)$; efficiency is gained by omitting this check. We implement this optimization by means of a set S^{val}, which is the set of uninstantiated variables whose domain has been reduced since the previous invocation of STR2. Initially, this set also contains the last assigned variable, denoted by $lastPast(P)$ here, if it belongs to the scope of the constraint c. After any variable assignment $x = a$, some tuples may become invalid due to the removal of values from $dom(x)$. The last assigned variable is the only instantiated variable for which validity operations must be performed. Algorithm 6 checks validity only for variables in S^{val}. The set S^{val} is initialized at lines 2 through 4 of Algorithm 5. At line 8 of this algorithm, $|dom(x)|$ is the size of the current domain of x while $lastSize[c][x]$ is the size of the domain of x, the last time the specific constraint c was processed (see lines 10 and 30); initially we have $lastSize[c][x] = -1$ for every arc (c, x), i.e. each pair composed of a constraint c and a variable x in $scp(c)$. If these two values differ at line 8 then $dom(x)$ has changed since the previous invocation of Algorithm 5 for the specific constraint c. In this case, x is included in S^{val} at line 9. This is how the membership of S^{val} is determined.

Algorithm 5: STR2(c: constraint): set of variables

```
Input: $c$ is a constraint (of the constraint network $P$ to be solved)
Output: the set of variables in $scp(c)$ with reduced domain
1  $S^{sup} \leftarrow \emptyset$
2  $S^{val} \leftarrow \emptyset$
3  if $lastPast(P) \in scp(c)$ then
4      $S^{val} \leftarrow S^{val} \cup \{lastPast(P)\}$
5  foreach variable $x \in scp(c) \mid x \notin past(P)$ do
6      $gacValues[x] \leftarrow \emptyset$
7      $S^{sup} \leftarrow S^{sup} \cup \{x\}$
8      if $|dom(x)| \neq lastSize[c][x]$ then
9          $S^{val} \leftarrow S^{val} \cup \{x\}$
10         $lastSize[c][x] \leftarrow |dom(x)|$
11 $i \leftarrow 1$
12 while $i \leq currentLimit[c]$ do
13     $index \leftarrow position[c][i]$
14     $\tau \leftarrow table[c][index]$
15     if $isValidTuple(c, S^{val}, \tau)$ then
16         foreach variable $x \in S^{sup}$ do
17             if $\tau[x] \notin gacValues[x]$ then
18                 $gacValues[x] \leftarrow gacValues[x] \cup \{\tau[x]\}$
19                 if $|gacValues[x]| = |dom(x)|$ then
20                     $S^{sup} \leftarrow S^{sup} \setminus \{x\}$
21         $i \leftarrow i + 1$
22     else
23         removeTuple($c, i, |past(P)|$)                // $currentLimit[c]$ decremented
   // domains are now updated and $X_{evt}$ computed
24 $X_{evt} \leftarrow \emptyset$
25 foreach variable $x \in S^{sup}$ do
26     $dom(x) \leftarrow gacValues[x]$
27     if $dom(x) = \emptyset$ then
28         throw INCONSISTENCY
29     $X_{evt} \leftarrow X_{evt} \cup \{x\}$
30     $lastSize[c][x] \leftarrow |dom(x)|$
31 return $X_{evt}$
```

Algorithm 6: isValidTuple(c: constraint, S^{val}: variables, τ: tuple): Boolean

```
Input: $c$ is a constraint
Input: $S^{val}$ is the set of variables to be considered during the check
Input: $\tau$ is a tuple whose validity must be checked
Output: $true$ iff $\tau$ is valid on $c$
1 foreach variable $x \in S^{val}$ do
2     if $\tau[x] \notin dom(x)$ then
3         return false
4 return true
```

The worst-case time complexity of STR2 is $O(r'd + r''t')$, where $r' = |scp(c) \setminus past(P)|$ denotes the number of uninstantiated variables in $scp(c)$, $r'' = |S^{val}|$ denotes the number of variables for which validity operations must be performed, and t' denotes the size of the current table of c. Performing a validity check is now $O(r'')$ instead of $O(r)$, as can be seen in Algorithm 6, since only variables in S^{val} are checked. The loops at lines 5, 12 and 25 are $O(r')$, $O(r''t')$ and $O(r'd)$, respectively. Like STR, the worst-case space complexity of STR2 is $O(n + rt)$ per constraint since data structures inherited from STR are $O(n + rt)$, *lastSize* is $O(r)$; S^{sup} and S^{val} are also $O(r)$ but may be shared by all constraints.

We illustrate now how important is the difference in behavior that may occur, in practice, between the two algorithms. Let us consider a positive table constraint c such that $scp(c) = \{x_1, ..., x_r\}$ and the table initially includes:

```
(0,0,...,0)
(1,1,...,1)
...
(d-2,d-2,...,d-2)
(d-2,d-1,...,d-1)
(d-1,0,...,0)
...
```

In this example, the domain of each variable involved in c comprises all digits from 0 to $d-1$. In the table, each of the first $d-1$ tuples is a sequence that repeats the same digit (from 0 to $d-2$). The d^{th} tuple consists of the digit $d-2$ followed by a sequence of $d-1$. The $d+1^{th}$ tuple consists of the digit $d-1$ followed by a sequence of 0. Assume that $past(P) = \emptyset$ (no variable has been assigned) and that simple tabular reduction (either of the two algorithms) is applied to this constraint. No value is removed because all values are present in domains, and there exists a support for each value. Now, imagine that $(x_1, d-1)$ is deleted while propagating some other constraints, whereas all other values remain valid. If STR is applied again to this constraint, no value will be removed (since the constraint is still GAC-consistent as any remaining value has still a support), but some tuples (at least the $d+1^{th}$ one) will be eliminated. Interestingly, calling STR requires $O(r)$ constant time operations to deal with *gacValues* structures (loops starting at line 1 and 15), $O(rt)$ operations to perform validity checks, $O(rt)$ operations to check GAC values and $O(rd)$ operations to collect GAC values. On the other hand, calling STR2 requires $O(r)$ operations to deal with *gacValues* structures, $O(t)$ operations to perform validity checks (since $S^{val} = \{x_1\}$), $O(rd)$ operations to check GAC values (since $S^{sup} = \emptyset$ after the treatment of the first d tuples) and $O(rd)$ operations to collect GAC values. This leads to:

Observation 1 There exist situations where applying STR to an r-ary constraint is $O(rt + rd)$ whereas applying STR2 is $O(t + rd)$.

Most of the time, $d \ll t$ since $t \in O(d^r)$. In this case, Observation 1 shows that STR2 is potentially r times faster than STR. The higher the arity, the greater the possible benefit of using STR2.

Finally, there are two possible ways to cope with backtracking. One way is to reinitialize the array *lastSize*, filling it with the special value -1. The other way is to record the content of such an array at each depth of search, so that the original state of the array can be restored upon backtracking. This approach, which requires some additional data structures, is denoted by STR2+.

To manage decrementality, we need first to introduce an array *lastLevel* that indicates for each constraint c at which level (number of instantiated variables as given by $|past(P)|$) the constraint c was most recently enforced to be GAC-consistent. We also need an array *stack* that stores for each constraint c and each level p the value of the array *lastSize* at the end of the last invocation of STR2 for constraint c at level p. When the function restoreTuples, Algorithm 4, is called, it suffices to additionally execute the following instruction:

$lastLevel[c] \leftarrow min(lastLevel[c], p - 1)$

This temporarily updates the value of *lastLevel*[c] since we are about to backtrack to level $p - 1$. When we start executing the function STR2, Algorithm 5, we have to execute the following instructions:

```
$level \leftarrow |past(P)|$
for $i$ ranging from $lastLevel[c] + 1$ to $level$ do
    $stack[c][i] \leftarrow stack[c][lastLevel[c]]$
$lastSize[c] \leftarrow stack[c][level]$
$lastLevel[c] \leftarrow level$
```

Between *lastLevel*[c] + 1 and *level*, Algorithm 5 was never called for c, so this is the reason why we copy the value of *stack*[c][*lastLevel*[c]]. Most of the time, $level = lastLevel[c] + 1$ or $level = lastLevel[c]$. To benefit from decrementality, *lastSize*[c] is initialized with the value of *stack*[c][*level*], and *lastLevel*[c] is updated. STR2+ requires an additional structure that is $O(nr)$ per constraint.

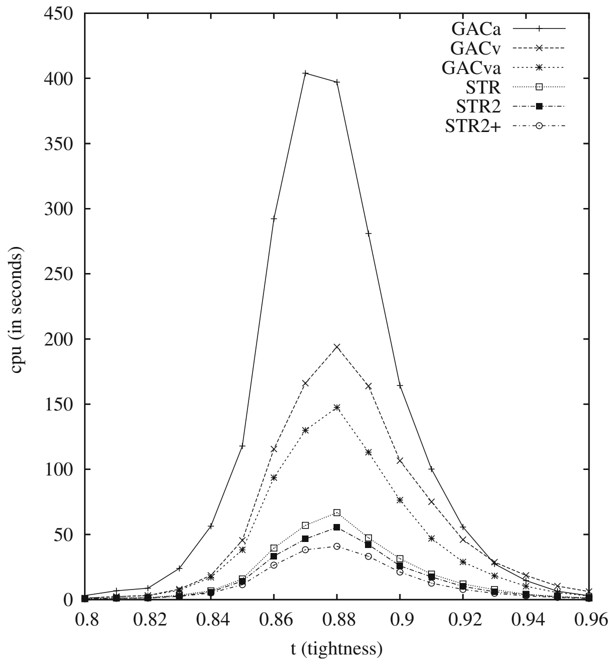
6 Experimental results

In order to show the practical interest of simple tabular reduction, and in particular the optimization we propose, we have conducted an experimentation (with our solver AbsCon) using a cluster of Xeon 3.0GHz with 1GiB of RAM under Linux, employing MAC with *dom/ddeg* [5] as variable ordering heuristic, and *lexico* as value ordering heuristic. In our implementation, using the adaptive heuristic *dom/wdeg* [8] does not guarantee to explore the same search tree when classical schemes are used and when simple tabular reduction schemes are used because the order in which constraints are propagated is different, which modifies the process of constraint weighting. This is the reason why we have discarded this heuristic.

We have first compared classical schemes to enforce GAC on (positive) table constraints with simple tabular reduction (recall that these schemes are compared here, when used within MAC). More precisely, we have implemented GAC-valid (GACv for short in graphs and tables), GAC-allowed (GACa for short) as well as the refined scheme GAC-valid+allowed (GACva for short). GAC-valid+allowed remains a good representative algorithm for table constraints. Indeed, our own experience confirms the results reported by Gent et al.: GAC-valid+allowed and the trie approach [17] are fairly robust and close in terms of performance. The underlying GAC algorithm embedded in these various schemes is GAC3^{rm} [27].

We have performed a first experimentation with random CSP instances. We have generated different classes of instances from Model RD [45]. Each generated class $\langle r, 60, 2, 20, t\rangle$ contains 20 CSP instances involving 60 Boolean variables and 20 r-ary constraints of tightness t. Provided that the arity r of the constraints is greater than or equal to 8, Theorem 2 [45] holds: an asymptotic phase transition is guaranteed at the threshold point $t_{cr} = 0.875$. It means that the hardest instances are generated when the tightness t is close to t_{cr}. Figure 3 shows the mean cpu time required by MAC to solve 20 instances of each class $\langle 13, 60, 2, 20, t\rangle$ where t ranges from 0.8 to 0.96. On these instances of intermediate difficulty, we can observe that STR is far more efficient than classical schemes (including GACva). When focusing on simple tabular reduction, Figs. 4, 5, 6 and 7 clearly confirm the general observation made

Fig. 3 Mean search cost of solving instances in classes $\langle 13, 60, 2, 20, t\rangle$ with MAC



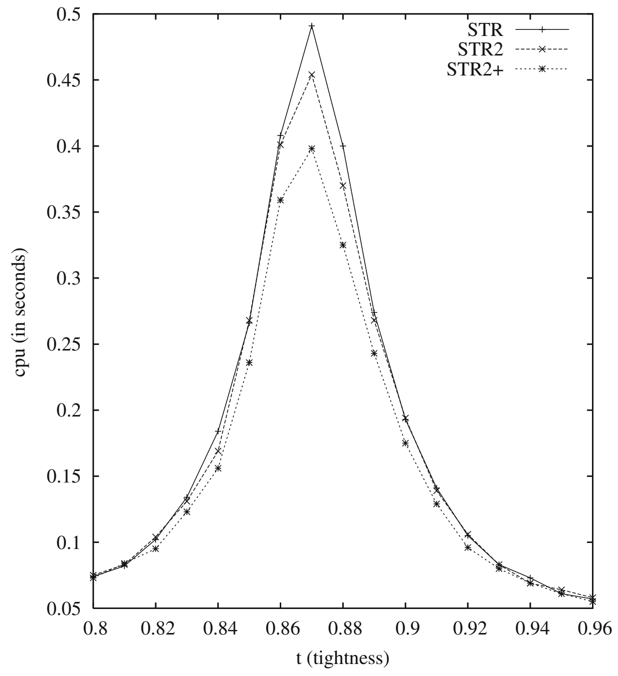


Fig. 4 Mean search cost of solving instances in classes ⟨10, 60, 2, 20, *t*⟩ with MAC

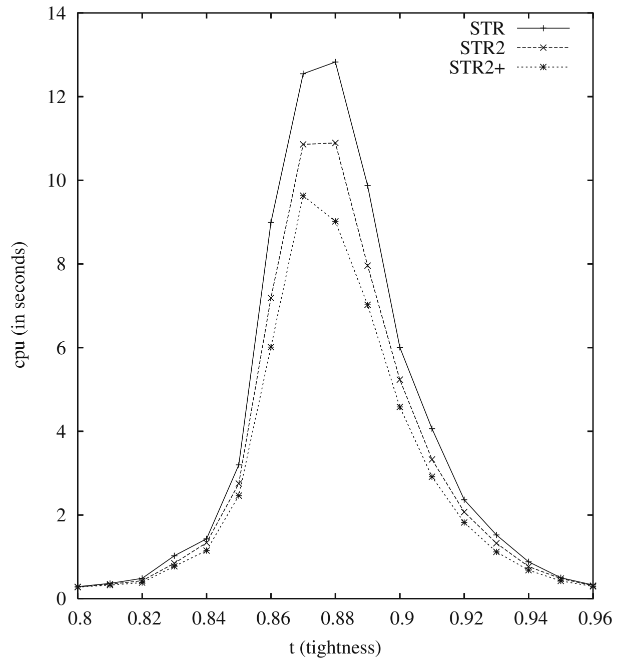


Fig. 5 Mean search cost of solving instances in classes ⟨12, 60, 2, 20, *t*⟩ with MAC

Fig. 6 Mean search cost of solving instances in classes ⟨14, 60, 2, 20, *t*⟩ with MAC

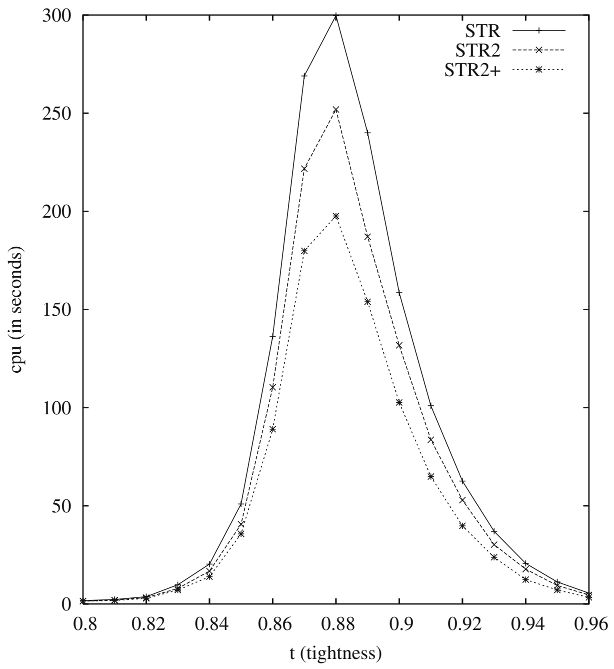
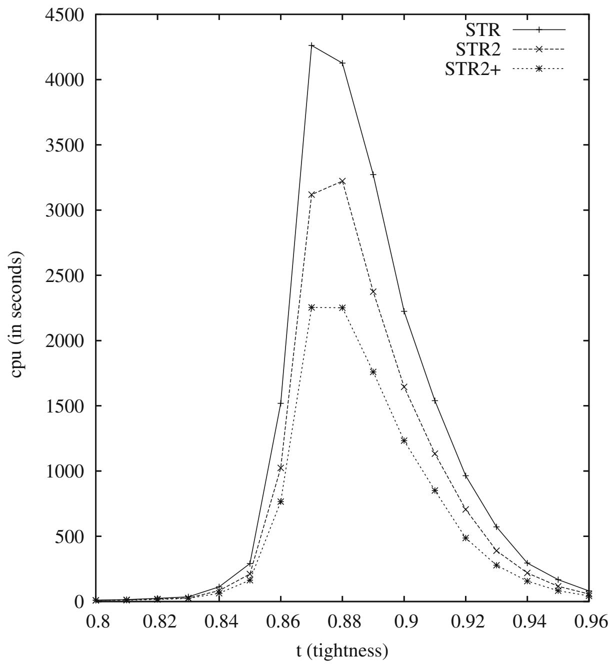


Fig. 7 Mean search cost of solving instances in classes ⟨16, 60, 2, 20, *t*⟩ with MAC



in Section 5 about the increasing interest of using STR2(+) when the arity of the constraints increases. Indeed, while STR2+ is about 20% faster than STR (at the threshold) when the arity of constraints is 10, it becomes two times faster when the arity of constraints is 16. Similar results have been obtained with larger domains.

Next, we have experimented on series of (random and structured) CSP instances involving table constraints, that have been selected as benchmarks for the third[2] and fourth[3] constraint solver competitions, and are available from http://www.cril.fr/~lecoutre/. These series represent a large spectrum of instances, and importantly, allow anyone to reproduce our experimentation.

1. The two series [11] *bdd-21-15* and *bdd-21-18* contain 35 instances each, involving 21 Boolean variables and large and small BDD constraints of arity 15 and 18, respectively. BDD constraints are extensional constraints generated from binary decision diagrams.
2. The two series *mdd-25-7* and *mdd-23-15* contain 25 instances each, involving MDD constraints of arity 7 and 15, respectively. MDD constraints are extensional constraints generated from multi-valued decision diagrams. These series have been submitted by Cheng and Yap to the 2008 constraint solver competition.
3. The series *rand-20-3* contains 50 random ternary instances each involving 20 variables. These instances were introduced in [44].
4. The two series *rand-20-8* and *rand-20-10* contain 20 random instances each involving 20 variables. Each instance of the series *rand-20-8* (resp., *rand-20-10*) involves domains containing 5 (resp., 10) values and 18 (resp., 5) constraints of arity 8 (resp., 10). Tables contain about 78,000 and 10,000 tuples, respectively.
5. Given a grid and a dictionary, the Crossword problem is to fill the grid with words in the dictionary. To generate crossword instances, three series of grids (Herald, Puzzle, Vg) and four dictionaries (Lex, Uk, Words, Ogd) have been used. Herald refers to crossword puzzles taken from the Herald Tribune (Spring, 1999), Puzzle refers to crossword puzzles mentioned in [18, 19] and Vg refers to blank grids. Lex is a dictionary used in [38], Uk is the UK cryptic solvers dictionary, Words is the dictionary in /usr/dict/words under Linux, and Ogd is a french dictionary.[4] Lex and Words are small dictionaries whereas Uk and Ogd are large. The model used to generate the instances is the one identified by $m1$ in [1]: for each grid, there is a variable per white square with 26 possible values (letters of the Latin alphabet), and a constraint for any sequence of white squares that corresponds to a word to be put in the grid. For the Vg grids, all instances only involve extensional constraints because, putting the same word several times on the grid is authorized. The arity of the constraints is given by the size of the grids: for example, *cw-lex-vg5-6* involves table constraints of arity 5 and 6 (the grid being 5 by 6).
6. The series *renault-mod* contains 50 real-world instances involving domains containing up to 42 values and constraints of various arity defined by large tables (the greatest one contains about 50,000 6-tuples).

[2]http://www.univ-artois.fr/CPAI08

[3]http://www.univ-artois.fr/CPAI09

[4]http://pagesperso-orange.fr/ledefi

7. The two series *tsp-20* and *tsp-25* contain 15 instances of the Travelling Salesperson Problem each, involving domains containing up to 1,000 values and ternary constraints defined by large tables (about 20,000 3-tuples).

Table 1 indicates the mean cpu time required to solve the instances of these different series with MAC; best results are printed in bold. Most of the time, STR2+ is the most efficient approach; two exceptions are series *mdd-25-7* and *rand-20-3*. In particular, STR2+ is 3 times faster than STR on the *bdd-21-15* series and 10 times faster than GACva on the *rand-20-10* series. Table 2 presents the results obtained on some representative BDD/MDD and random instances. Here, for each series, we show the results for 3 instances of various difficulty. For example, the instance *bdd-21-15-22* only requires visiting 21 nodes (to be solved) whereas the instance *bdd-21-15-35* requires visiting 1,465 nodes. Typically, when the instance is easy, using simple tabular reduction is rather penalizing. This is not really surprising since managing dynamic tables is then just an overhead. This is particularly visible for easy instances of series *bdd-21-15* and *bdd-21-18*. In terms of memory, the difference of memory consumption between all algorithms is at most by a factor 2. Note that the additional structure in $O(nr)$ required by STR2+ has a very limited practical impact on all these series.

The instances of series *mdd-25-7* and *rand-20-3* are better adapted to GACv. Every constraint c involved in any instance of series *mdd-25-7* is such that $|val(c)| = 5^7$, and every constraint c involved in any instance of series *rand-20-3* is such that $|val(c)| = 20^3$. The relative small size of theses sets (composed of initially valid tuples) is clearly an advantage for the scheme GAC-valid. This explains the good behavior of GACv on these two series.

Table 3 gives additional results for series of structured instances. Although instances of series *tsp-20* and *tsp-25* only involve ternary constraints, GACv is not the most efficient approach here. This can be explained by the large domain size $(1,000)$ of certain variables. Finally, Table 4 shows the results that we have obtained for the Crossword problem with respect to four dictionaries (lex, words, uk, ogd). This confirms our previous results. On the most difficult instances, STR2+ is about two times faster than STR and one order of magnitude faster than GACva. One

Table 1 Mean cpu time (in seconds) to solve instances of different series (a time-out of 1,200 s was set per instance) with MAC

Series	#Inst	Classical GAC schemes			Simple tabular reduction		
		GACv	GACa	GACva	STR	STR2	STR2+
bdd-21-15	35	75.7	401.7	65.1	204.0	94.7	**51.1**
bdd-21-18	35	42.3	(22 out)	43.7	77.2	36.0	**23.0**
mdd-25-7	25	**(15 out)**	(24 out)	(17 out)	(21 out)	(21 out)	(21 out)
mdd-23-15	25	(6 out)	(25 out)	(1 out)	157.7	106.5	**81.2**
rand-20-3	48	**88.5**	333.8	194.0	313.3	297.4	252.7
rand-20-8	20	115.8	(17 out)	129.5	104.3	75.4	**59.2**
rand-20-10	20	(20 out)	5.16	6.02	1.24	0.94	**0.71**
crosswords	258	(236 out)	(56 out)	(67 out)	(52 out)	(51 out)	**(48 out)**
renault-mod	45	150.8	50.2	52.2	64.4	55.2	**44.1**
tsp-20	15	29.4	24.9	16.2	9.2	9.1	**8.4**
tsp-25	15	262.8	273.9	194.9	116.2	116.0	**110.6**

Table 2 Representative results obtained on BDD/MDD instances and ternary random instances

Instance		Classical GAC schemes			Simple tabular reduction		
		GACv	GACa	GACva	STR	STR2	STR2+
bdd-21-15-22	cpu	1.32	**1.30**	1.36	14.4	6.30	6.33
#nodes=21	mem	27 M	28 M	28 M	97 M	98 M	102 M
bdd-21-15-32	cpu	65.4	382	62.8	185	80.3	**46.6**
#nodes=1,140	mem	27 M	28 M	28 M	98 M	98 M	102 M
bdd-21-15-35	cpu	88.5	372	83.7	235	115	**61.1**
#nodes=1,465	mem	27 M	28 M	28 M	97 M	98 M	102 M
bdd-21-18-10	cpu	**0.83**	0.99	1.00	7.19	3.55	3.62
#nodes=21	mem	40 M	42 M	44 M	62 M	62 M	65 M
bdd-21-18-2	cpu	44.5	>20 m	45.4	83.4	38.4	**23.5**
#nodes=10,660	mem	61 M		44 M	65 M	65 M	66 M
bdd-21-18-11	cpu	66.5	>20 m	66.7	123	58.5	**34.9**
#nodes=14,716	mem	46 M		44 M	65 M	65 M	101 M
mdd-25-7-23	cpu	**45.2**	783	57.2	123	100	84.7
#nodes=24,926	mem	168 M	231 M	230 M	177 M	177 M	177 M
mdd-25-7-19	cpu	**374**	>20 m	482	742	579	483
#nodes=235K	mem	168 M		231 M	187 M	187 M	187 M
mdd-25-7-18	cpu	**789**	>20 m	997	>20 m	>20 m	>20 m
#nodes=485K	mem	167 M		229 M			
mdd-23-15-25	cpu	583	>20 m	197	162	112	**88.1**
#nodes=33,585	mem	521 M		633 M	530 M	532 M	530 M
mdd-23-15-9	cpu	899	>20 m	288	159	105	**81.6**
#nodes=60,436	mem	523 M		633 M	533 M	533 M	533 M
mdd-23-15-23	cpu	1,125	>20 m	424	158	105	**81.0**
#nodes=92,666	mem	523 M		633 M	535 M	535 M	534 M
rand-20-3-38	cpu	**11.7**	38.1	23.1	39.9	39.4	30.5
#nodes=29,428	mem	21 M	26 M	26 M	26 M	25 M	25 M
rand-20-3-8	cpu	**14.4**	51.4	31.6	46.3	40.5	36.6
#nodes=36,869	mem	21 M	28 M	28 M	26 M	27 M	27 M
rand-20-3-11	cpu	**263**	1,056	582	879	783	725
#nodes=632K	mem	22 M	28 M	28 M	56 M	56 M	56 M

Cpu time is given in seconds and mem(ory) in MiB. The number of nodes (#nodes) explored by MAC is given below the name of each instance

reason explaining the efficiency of simple tabular reduction on crossword puzzles is the fact that the tables become rather small after a few variables have been assigned.

7 Variants of simple tabular reduction

A development of STR involves partially iterating the (current) table. The following observation makes this possible: as soon as all values admit a support, the constraint is necessarily GAC-consistent. It is sufficient to introduce a counter *nbUnsupported*

Table 3 Representative results obtained on random instances with large arity constraints and on structured instances from series *renault-mod*, *tsp-20* and *tsp-25*

Instance		Classical GAC schemes			Simple tabular reduction		
		GACv	GACa	GACva	STR	STR2	STR2+
rand-20-8-7	cpu	**17.8**	577	19.8	31.0	23.5	18.0
#nodes=14,252	mem	171 M	215 M	215 M	176 M	175 M	175 M
rand-20-8-9	cpu	125	>20 m	152	157	115	**85.0**
#nodes=108K	mem	171 M		216 M	180 M	180 M	180 M
rand-20-8-13	cpu	448	>20 m	459	245	181	**141**
#nodes=569K	mem	171 M		216 M	202 M	202 M	203 M
rand-20-10-12	cpu	>20 m	4.17	5.1	0.53	0.47	**0.36**
#nodes=761	mem		24 M	15 M	11 M	11 M	11 M
rand-20-10-11	cpu	>20 m	5.79	21.3	0.93	**0.48**	0.72
#nodes=790	mem		24 M	27 M	22 M	22 M	22 M
rand-20-10-5	cpu	>20 m	6.29	16.7	1.46	**0.83**	0.92
#nodes=908	mem		24 M	15 M	22 M	11 M	22 M
renault-mod-0	cpu	10.4	1.07	1.05	1.01	1.01	**0.96**
#nodes=287	mem	36 M	42 M	42 M	35 M	35 M	36 M
renault-mod-12	cpu	246	100.0	102	101	89.0	**77.3**
#nodes=415K	mem	36 M	41 M	41 M	55 M	54 M	56 M
renault-mod-14	cpu	968	364	353	395	360	**282**
#nodes=1,135K	mem	37 M	42 M	42 M	43 M	42 M	42 M
tsp-20-190	cpu	5.92	6.47	5.5	3.89	3.86	**3.84**
#nodes=7,738	mem	217 M	10 M	19 M	18 M	19 M	19 M
tsp-20-366	cpu	34.4	42.8	30.1	17.9	17.8	**16.0**
#nodes=31,701	mem	250 M	20 M	20 M	20 M	20 M	20 M
tsp-20-193	cpu	273	205	120	63.2	62.4	**56.0**
#nodes=80,849	mem	295 M	21 M	21 M	23 M	18 M	23 M
tsp-25-13	cpu	3.64	2.81	2.68	1.91	**1.90**	1.95
#nodes=2,421	mem	195 M	19 M	19 M	18 M	18 M	18 M
tsp-25-163	cpu	159	200	126	73.0	71.9	**61.0**
#nodes=89,883	mem	307 M	21 M	21 M	26 M	17 M	26 M
tsp-25-456	cpu	956	1,055	742	436	438	**407**
#nodes=686K	mem	272 M	20 M	20 M	52 M	53 M	53 M

Cpu time is given in seconds and mem(ory) in MiB. The number of nodes (#nodes) explored by MAC is given below the name of each instance

that indicates the current number of values for which no support has been found yet. Algorithm 7 is a variant of Algorithm 1 that exploits this counter. Initially, the counter is set to the total number of values in the domains of uninstantiated variables; see lines 2 and 5. When a support is found for a value, the counter is decremented; see line 15. Finally, when the counter reaches 0, we can stop iterating over the tuples of the table; see lines 18 and 19. Notice that the variable *nbUnsupported* can also be used to break the loop starting at line 24. In particular, when *nbUnsupported*

Table 4 Representative results obtained with MAC on series of Crossword puzzles using dictionaries of different length

Instance		Classical GAC schemes			Simple tabular reduction		
		GACv	GACa	GACva	STR	STR2	STR2+
Crossword puzzles with dictionary lex (24,974 words)							
cw-lex-vg5-6	cpu	>20 m	40.7	61.9	14.7	11.9	**10.8**
#nodes=26,679	mem		2,895K	17 M	16 M	3,917K	16 M
cw-lex-vg5-7	cpu	>20 m	378	950	138	107	**89.7**
#nodes=171K	mem		17 M	17 M	23 M	23 M	24 M
cw-lex-vg6-6	cpu	>20 m	3.38	4.77	1.65	1.33	**1.24**
#nodes=1,602	mem		16 M	16 M	15 M	15 M	15 M
cw-lex-vg6-7	cpu	>20 m	475	>20 m	166	142	**116**
#nodes=152K	mem		17 M		23 M	23 M	23 M
Crossword puzzles with dictionary words (45,371 words)							
cw-words-vg5-5	cpu	14.2	0.38	**0.04**	0.34	0.40	**0.04**
#nodes=38	mem	16 M	16 M	4,972K	15 M	15 M	4,811K
cw-words-vg5-6	cpu	519	1.66	1.95	0.81	0.69	**0.33**
#nodes=718	mem	17 M	17 M	17 M	16 M	16 M	6,302K
cw-words-vg5-7	cpu	>20 m	20.3	36.8	6.78	5.6	**3.98**
#nodes=6,957	mem		18 M	18 M	16 M	16 M	8,185K
cw-words-vg5-8	cpu	>20 m	924	>20 m	271	226	**182**
#nodes=256K	mem		17 M		29 M	29 M	29 M
Crossword puzzles with dictionary uk (225,349 words)							
cw-uk-vg5-5	cpu	5.11	**0.36**	0.43	0.37	**0.36**	**0.36**
#nodes=28	mem	17 M	17 M	17 M	16 M	16 M	16 M
cw-uk-vg5-6	cpu	144	0.92	0.9	0.56	0.52	**0.50**
#nodes=145	mem	19 M	20 M	20 M	19 M	19 M	19 M
cw-uk-vg5-7	cpu	>20 m	3.51	5.44	0.85	0.71	**0.68**
#nodes=408	mem		21 M	21 M	20 M	21 M	19 M
cw-uk-vg5-8	cpu	>20 m	83.9	73.8	7.26	5.65	**4.77**
#nodes=8,148	mem		23 M	22 M	22 M	22 M	22 M
Crossword puzzles with dictionary ogd (435,705 words)							
cw-ogd-vg6-6	cpu	372	0.76	0.67	0.58	0.51	**0.48**
#nodes=98	mem	18 M	19 M	19 M	18 M	18 M	18 M
cw-ogd-vg6-7	cpu	>20 m	101	58.2	11.8	9.00	**7.43**
#nodes=9,522	mem		23 M	23 M	22 M	22 M	22 M
cw-ogd-vg6-8	cpu	>20 m	56.4	7.12	3.39	2.57	**2.26**
#nodes=2,806	mem		28 M	28 M	27 M	27 M	27 M
cw-ogd-vg6-9	cpu	>20 m	744	204	35.6	25.0	**19.5**
#nodes=23,283	mem		31 M	32 M	30 M	30 M	30 M

becomes 0 at line 18, there is no need to enter the loop starting at line 24. This optimization is not shown in the code of Algorithm 7.

Intuitively, to make this variant still more efficient, it seems better to put at the beginning of the table, some tuples that correspond to the first support of at least one value. Here, these tuples are called residues as in [27, 31]. Then, we just need to count the number of residues encountered so far when iterating the table; this is the role of the counter *nbResidues*. When a tuple is found to be the first support of (at least) one value, this tuple is swapped with the first tuple in the table which is not considered as a residue; see lines 16 and 17. The function storeResidue, Algorithm 8, allows us to perform this swapping. However, for STR2, we cannot use this variant as such. The reason is that if invalid tuples are not systematically removed from the (current) table, then the exploitation of S^{val} is not safe anymore.

We have implemented these variants in AbsCon, and conducted an experimentation in order to compare the behavior of:

- STR alone, Algorithm 1;
- STR^{p+r}, Algorithm 7; p stands for partial iteration and r for residues exploitation;

Algorithm 7: STR^{p+r}(c: constraint): set of variables

```
   Input: $c$ is a constraint (of the constraint network $P$ to be solved)
   Output: the set of variables in $scp(c)$ with reduced domain
 1 $nbResidues \leftarrow 0$
 2 $nbUnsupported \leftarrow 0$
   foreach variable $x \in scp(c) \mid x \notin past(P)$ do
       $gacValues[x] \leftarrow \emptyset$
 5     $nbUnsupported \leftarrow nbUnsupported + |dom(x)|$
   $i \leftarrow 1$
   while $i \leq currentLimit[c]$ do
       $index \leftarrow position[c][i]$
       $\tau \leftarrow table[c][index]$
10     $nbBefore \leftarrow nbUnsupported$
       if isValidTuple($c, \tau$) then
           foreach variable $x \in scp(c) \mid x \notin past(P)$ do
               if $\tau[x] \notin gacValues[x]$ then
                   $gacValues[x] \leftarrow gacValues[x] \cup \{\tau[x]\}$
15                 $nbUnsupported \leftarrow nbUnsupported - 1$
16         if $nbBefore > nbUnsupported$ then
17             storeResidue($c, i$)
18         if $nbUnSupported = 0$ then
19             break;                                   // Stop iterating the table
           $i \leftarrow i + 1$
       else
           removeTuple($c, i, |past(P)|$)               // $currentLimit[c]$ decremented
   // domains are now updated and $X_{evt}$ computed
   $X_{evt} \leftarrow \emptyset$
24 foreach variable $x \in scp(c) \mid x \notin past(P)$ do
       if $gacValues[x] \subset dom(x)$ then
           $dom(x) \leftarrow gacValues[x]$
           if $dom(x) = \emptyset$ then
               throw INCONSISTENCY
           $X_{evt} \leftarrow X_{evt} \cup \{x\}$
   return $X_{evt}$
```

Algorithm 8: storeResidue(c: constraint, i: integer)

```
Input: $c$ is a constraint
Input: $i$ is the position of the tuple to be moved
$tmp \leftarrow position[c][i]$
$position[c][i] \leftarrow position[c][nbResidues]$
$position[c][nbResidues] \leftarrow tmp$
$nbResidues \leftarrow nbResidues + 1$
```

Table 5 Mean cpu time (in seconds) to solve instances of different series (a time-out of 1,200 s was set per instance) with MAC

Series	#Inst	STR	STR^r	STR^p	STR^{p+r}
bdd-21-15	34	204.0	202.5	132.6	**117.6**
bdd-21-18	35	**77.2**	77.7	(3 out)	(3 out)
mdd-25-7	25	**(21 out)**	**(21 out)**	(24 out)	(24 out)
mdd-23-15	25	157.7	**157.4**	(23 out)	(25 out)
rand-20-3	47	**298.1**	298.5	352.5	367.1
rand-20-8	20	104.3	**104.1**	(5 out)	(6 out)
rand-20-10	20	1.24	1.14	0.96	**0.89**
crosswords	205	53.6	54.7	51.0	**43.9**
renault-mod	45	64.4	64.5	49.3	**47.4**
tsp-20	15	9.2	**8.9**	9.2	9.0
tsp-25	15	116.2	115.8	**114.0**	114.2

- STR^p, where only partial iteration is used (lines 16 and 17 of Algorithm 7 discarded);
- STR^r, where only residues are used (lines 18 and 19 of Algorithm 7 discarded);
- STR2 alone, Algorithm 5;
- STR2^r, where residues are used as in Algorithm 7.

Table 5 indicates the mean cpu time required to solve the instances of the different series introduced earlier with variants of STR. A first observation is that managing residues has little impact. If STR^p and STR^{p+r} can be slightly differentiated, STR and STR^r are really quite close. A second observation is that partial iteration may have a significant impact. It may be an advantage (compare STR and STR^p on series *bdd-21-15*) and it may be a disadvantage (compare STR and STR^p on series *bdd-21-18*, *mdd-23-15* and *rand-20-8*). Overall, a general lesson is that the best variant of STR is always outperformed by STR2(+). As Table 6 confirms for STR2(+) that managing residues is not important, we can deduce that STR2+ remains so far the best algorithm based on simple tabular reduction.

8 Comparison with other approaches

We want first to emphasize that it is very difficult to compare the practical behavior of two (or more) sophisticated algorithms developed by independent research teams.

Table 6 Mean cpu time (in seconds) to solve instances of different series (a time-out of 1,200 s was set per instance) with MAC

Series	#Inst	STR2	STR2^r	STR2+	STR2+^{r}
bdd-21-15	34	94.7	93.0	51.1	**50.0**
bdd-21-18	35	36.0	35.9	**23.0**	23.1
mdd-25-7	4	545.7	534.5	**448.3**	449.3
mdd-23-15	25	106.5	105.7	**81.2**	81.3
rand-20-3	48	297.4	300.6	**252.7**	252.8
rand-20-8	20	75.4	74.3	59.2	**59.1**
rand-20-10	20	0.94	0.89	**0.71**	0.76
crosswords	206	51.3	49.0	37.8	**35.9**
renault-mod	45	55.2	54.2	**44.1**	44.7
tsp-20	15	9.2	9.1	**8.4**	8.9
tsp-25	15	116.0	116.4	**110.6**	111.5

Comparison of a new algorithm with previous ones is made possible by means of published results when the environment (operating system, cpu, language, etc.) and the search/inference procedure is well-known (e.g. search heuristics, branching mechanism, etc.). Another option is to reimplement previous algorithms, but with the trap of badly using required data structures. Nevertheless, this was our choice.

8.1 Comparison with the MDD approach

The MDD approach [13] to enforce GAC on table constraints can certainly be considered as the most efficient approach that has been developed recently. This is why we have implemented it (using the sparse set data structure as described in the paper) in our platform AbsCon. If as mentioned above, this is not a perfect solution to compare algorithms, we do believe that it can provide interesting general indications. We have implemented the algorithm described in [12, 13] with (MDD^c) and without (MDD) the early cutoff optimization. Table 7 shows how simple tabular reduction (STR2+) and MDD variants behave respectively. On some series (*mdd-23-15*, *rand-20-10* and *crosswords*), STR2+ largely outperforms MDD^c. On some other series (*mdd-25-7* and *rand-20-3*), MDD^c largely outperforms STR2+. This is when the table can be efficiently compressed by means of a MDD: the compression ratio (number of MDD nodes over number of initial tuples) is about 1% for a constraint from series *mdd-25-7* and 14% for a constraint from series *rand-20-3*. Finally, STR2+ and MDD^c have a rather similar behavior on series *renault-mod*, *tsp-20* and *tsp-25*. Results for representative instances are given in Table 8.

8.2 Comparison with binary encoding approaches

There are three well-known binary encodings of non-binary constraint networks. These encodings that transform any non-binary constraint network into an equivalent binary one are:

- the dual encoding [15]
- the hidden variable encoding [36]
- the double encoding [39]

All three binary encodings represent each non-binary constraint c by a *dual* variable whose domain is the set of tuples allowed by c. In the dual encoding, for

Table 7 Mean cpu time (in seconds) to solve instances of different series (a time-out of 1,200 s was set per instance) with MAC

Series	#Inst	STR2+	MDD	MDD^c
bdd-21-15	35	**50.7**	205.4	149.3
bdd-21-18	35	**23.0**	142.5	99.0
mdd-25-7	25	(19 out)	(9 out)	**(6 out)**
mdd-23-15	25	**81.3**	666.1	645.3
rand-20-3	49	244.2	269.4	**129.2**
rand-20-8	20	**59.3**	171.4	109.2
rand-20-10	20	**0.85**	7.3	7.2
crosswords	199	**20.4**	90.9	89.8
renault-mod	46	64.8	74.8	**61.7**
tsp-20	15	**8.8**	10.8	11.0
tsp-25	15	**114.3**	129.5	136.6

Table 8 Representative results obtained with MAC on different series

Instance	#Inst	STR2+	MDD	MDD^c
mdd-25-7-1	cpu	740	166	**124**
#nodes=259K	mem	182 M	79 M	79 M
mdd-25-7-19	cpu	483	156	**121**
#nodes=235K	mem	181 M	78 M	78 M
mdd-25-7-3	cpu	>20 m	388	**280**
#nodes=558K	mem		85 M	85 M
rand-20-3-12	cpu	49.4	48.7	**23.8**
#nodes=38,268	mem	26 M	30 M	30 M
rand-20-3-18	cpu	142	135	**58.0**
#nodes=69,427	mem	26 M	29 M	29 M
rand-20-3-37	cpu	838	931	**447**
#nodes=575K	mem	39 M	42 M	42 M
rand-20-10-12	cpu	**0.72**	4.4	4.13
#nodes=761	mem	22 M	48 M	48 M
rand-20-10-5	cpu	**0.91**	11.4	11.2
#nodes=908	mem	22 M	48 M	48 M
rand-20-10-16	cpu	**1.3**	18.4	17.7
#nodes=867	mem	22 M	48 M	48 M
cw-uk-vg5-7	cpu	**0.65**	1.9	1.87
#nodes=408	mem	20 M	24 M	24 M
cw-uk-vg6-7	cpu	**69.1**	345	344
#nodes=92,493	mem	24 M	29 M	29 M
cw-uk-vg7-7	cpu	**124**	668	659
#nodes=127K	mem	24 M	28 M	27 M
renault-mod-0	cpu	**1.04**	1.19	1.16
#nodes=287	mem	35 M	27 M	27 M
renault-mod-12	cpu	**71.6**	96.7	88.0
#nodes=415K	mem	46 M	37 M	37 M
renault-mod-14	cpu	292	327	**273**
#nodes=1,135K	mem	63 M	55 M	55 M
tsp-25-13	cpu	**1.94**	2.19	2.18
#nodes=2,421	mem	18 M	19 M	19 M
tsp-25-163	cpu	**65.7**	73.1	78.5
#nodes=89,883	mem	24 M	25 M	25 M
tsp-25-456	cpu	**432**	488	515
#nodes=686K	mem	36 M	38 M	38 M

MDD^c outperforms STR2+ on series *mdd-25-7* and *rand-20-3*. STR2+ outperforms MDD^c on series *rand-20-10* and *crosswords*. STR2+ and MDD^c have a rather similar behavior on series *renault-mod* and *tsp-25*

each pair of dual variables that represent non-binary constraints sharing at least one original variable, a binary *dual* constraint is introduced. In the hidden variable encoding, the original variables are kept, and binary *connection* constraints link original and dual variables. The double encoding includes both dual and connection constraints.

Table 9 Greatest domain size of dual variables (d_{dual}), number of dual constraints (e_{dual}), and number of instances that runs out of memory (MO) when considering the dual/double encoding for each series

Series	d_{dual}	e_{dual}	MO
bdd-21-15	≈7,000	3,678,828	35/35
bdd-21-18	≈57,000	8,778	35/35
mdd-25-7	≈45,000	≈1,440	25/25
mdd-23-15	150,000	120	25/25
rand-20-3	2,944	≈680	50/50
rand-20-8	≈78,000	≈152	20/20
rand-20-10	10,000	10	0/20
crosswords-lex	4,042	varied	0/63
crosswords-words	7,360	varied	16/65
crosswords-uk	32,865	varied	53/65
crosswords-ogd	68,064	varied	56/65
renault-mod	48,721	≈4,000	50/50
tsp-20	≈15,000	3,839	15/15
tsp-25	≈15,000	7,549	15/15

Because efficient algorithms [38] have been devised for binary encodings, it is natural to compare them with STR. Let us consider first the dual and the double encodings. In both of these encodings, we have dual constraints. More precisely, if x_c and $x_{c'}$ are two dual variables representing the original constraints c and c', and if $scp(c) \cap scp(c') \neq \emptyset$, then we have a dual constraint involving x_c and $x_{c'}$ and allowing pairs of values that correspond to original tuples with the same projection on $scp(c) \cap scp(c')$. When tables of non-binary constraints are large, we obtain dual variables with large domains. As a result, the tables of dual constraints may require excessive space since, if d_{dual} denotes the greatest domain size for dual variables, a dual constraint may need to store up to $d_{dual} \times d_{dual}$ tuples (pairs of values). Besides, depending on the way original constraints intersect, the number e_{dual} of dual constraints may also be very important. To summarize, the space required for the dual encoding (and a fortiori the double encoding) of a CN embedding non-binary constraints with large tables may easily exceed the available amount of computer memory. This is shown in Table 9 where for each series of instances considered in this paper, we indicate the typical values of d_{dual} and e_{dual} as well as the number

Table 10 Mean cpu time (in seconds) to solve instances of different series (a time-out of 1,200 s was set per instance) with MAC

Series	#Inst	STR2+	AC-H	HAC
Random series				
rand-20-3	14	**236**	493	476
rand-20-8	10	**163**	445	570
rand-20-10	20	**15.2**	46.8	41.1
Crosswords series Herald and puzzle				
crosswords-lex	48	**6.7**	10.1	8.5
crosswords-words	53	**10.1**	26.6	18.0
crosswords-uk	58	**6.8**	15.6	13.6
crosswords-ogd	55	**1.3**	2.7	2.7
Crosswords series Vg				
crosswords-lex	68	**7.9**	25.0	13.0
crosswords-words	64	**21.6**	78.0	43.8
crosswords-uk	39	**38.3**	124	98.4
crosswords-ogd	37	**26.1**	44.2	46.1

Table 11 Representative results obtained on some series

Instance		STR2+	AC-H	HAC
rand-20-3-21	cpu	**152**	365	298
#nodes=171K	mem	42 M	67 M	67 M
rand-20-8-15	cpu	**100**	332	387
#nodes=345K	mem	196 M	607 M	607 M
rand-20-10-15	cpu	**20**	39	35.5
#nodes=21,600	mem	33 M	56 M	56 M
cw-lex-15.04	cpu	**9.2**	24.3	17.8
#nodes=25,934	mem	29 M	79 M	79 M
cw-lex-15.07	cpu	**210**	320	274
#nodes=1,720K	mem	30 M	89 M	89 M
cw-ogd-vg6-9	cpu	**22.7**	78.3	57.2
#nodes=29,986	mem	42 M	257 M	257 M
cw-ogd-vg8-8	cpu	**82.9**	354	257
#nodes=29,362	mem	38 M	341 M	342 M

of instances that runs out of memory (MO) at loading time, given 1GiB of RAM; $\approx$ means that the average value (computed over all instances of the same series) is given. For example, each instance of the series *bdd-21-15* requires a huge number of dual constraints (more than 3,000,000), and each instance of the series *bdd-21-18* involves dual variables with domains containing around 57,000 values. Clearly, many instances of these series cannot be converted using the dual/double encoding: this is only possible here for the series *rand-20-10* and basically for the Crossword instances that are built from a small dictionary (lex and words). As the last three series of Table 9 involve both positive and negative table constraints, we only tried to build dual variables and dual constraints corresponding to positive table constraints while discarding negative ones. Even with this restriction, we didn't succeed in loading any instance of these series.

Now, let us focus on the hidden encoding. We propose to compare STR2+ on the original non-binary instances with our speediest AC algorithm, AC3^{bit+rm} [29], on the hidden variable encoding, denoted by AC-H, and the AC algorithm specialized for the hidden variable encoding proposed in [38], denoted here by HAC. For our comparison, we still use MAC but we now adopt *dom* [21], which is the variable ordering heuristic that selects at each search step the variable with the smallest domain size. Using *dom* and instantiating original variables only (i.e. not dual ones) guarantee that the same search trees are built by the three approaches. Table 10 shows the mean CPU times obtained by MAC on the different series of random instances and crossword puzzles. STR2+ is shown to be two or three times faster than AC3^{bit+rm} and HAC on the hidden variable encoding. Table 11 shows this on some representative instances.

9 Conclusion

Simple tabular reduction (STR) [41] is a simple and effective GAC algorithm for positive table constraints. In this paper, we have proposed an optimization of this algorithm. This new algorithm (STR2+) appears among state-of-the-art GAC algorithms for (non-binary) table constraints. We have shown this experimentally by comparing STR2+ with classical GAC schemes (including the robust GAC-valid+allowed), different binary encodings and also the recent MDD approach. Interestingly, STR2+ and MDD^c seem to be rather complementary: the best choice among STR2+ or MDD^c depends on the compression ratio of built multi-valued decision diagrams.

Dynamically maintaining the list of supports in table constraints, as performed by simple tabular reduction, has some other nice features that might be useful in the near future. Because the current number of supports is permanently known, it is quite easy to compute the current tightness of the constraints. This can be exploited, for example, by search heuristics and/or shaving techniques. Besides, if the current tightness of a constraint is found to be 0, then it means that this constraint is entailed (i.e. similar to a universal constraint). Temporarily discarding such constraints may improve the efficiency of propagation. The identification of entailed constraints can also play an interesting role for counting or enumerating all solutions of loose problems.

Acknowledgements I would like to thank Julian Ullmann for fruitful discussions and helpful comments, as well as the anonymous reviewers for their constructive remarks.

References

1. Beacham, A., Chen, X., Sillito, J., & van Beek, P. (2001). Constraint programming lessons learned from crossword puzzles. In *Proceedings of Canadian conference on AI* (pp. 78–87).
2. Bessiere, C. (2006). Constraint propagation. In *Handbook of constraint programming* (chapter 3). Elsevier.
3. Bessiere, C., & Debruyne, R. (2005). Optimal and suboptimal singleton arc consistency algorithms. In *Proceedings of IJCAI'05* (pp. 54–59).
4. Bessiere, C., Hebrard, E., Hnich, B., & Walsh, T. (2007). The complexity of reasoning with global constraints. *Constraints, 12*(2), 239–259.
5. Bessiere, C., & Régin, J. (1996). MAC and combined heuristics: Two reasons to forsake FC (and CBJ?) on hard problems. In *Proceedings of CP'96* (pp. 61–75).
6. Bessiere, C., & Régin, J. (1997). Arc consistency for general constraint networks: Preliminary results. In *Proceedings of IJCAI'97* (pp. 398–404).
7. Bessiere, C., Régin, J.C., Yap, R., & Zhang, Y. (2005). An optimal coarse-grained arc consistency algorithm. *Artificial Intelligence, 165*(2), 165–185.
8. Boussemart, F., Hemery, F., Lecoutre, C., & Sais, L. (2004). Boosting systematic search by weighting constraints. In *Proceedings of ECAI'04* (pp. 146–150).
9. Briggs, P., & Torczon, L. (1993). An efficient representation for sparse sets. *ACM Letters on Programming Languages and Systems, 2*(1–4), 59–69.
10. Carlsson, M. (2006). *Filtering for the case constraint. Talk given at the advanced school on global constraints*.
11. Cheng, K., & Yap, R. (2006). Maintaining generalized arc consistency on ad-hoc n-ary Boolean constraints. In *Proceedings of ECAI'06* (pp. 78–82).
12. Cheng, K., & Yap, R. (2008). Maintaining generalized arc consistency on ad-hoc r-ary constraints. In *Proceedings of CP'08* (pp. 509–523).
13. Cheng, K., & 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.
14. Dechter, R. (2003). *Constraint processing*. Morgan Kaufmann.
15. Dechter, R., & Pearl, J. (1989). Tree clustering for constraint networks. *Artificial Intelligence, 38*(3), 353–366.
16. Fredkin, E. (1960). Trie memory. *Communications of the ACM, 3*(9), 490–499.

17. Gent, I. P., Jefferson, C., Miguel, I., & Nightingale, P. (2007). Data structures for generalised arc consistency for extensional constraints. In *Proceedings of AAAI'07* (pp. 191–197).
18. Ginsberg, M. L. (1993). Dynamic backtracking. *Journal of Artificial Intelligence Research, 1*, 25–46.
19. Ginsberg, M. L., Frank, M., Halpin, M. P., & Torrance, M. C. (1990). Search lessons learned from crossword puzzles. In *Proceedings of AAAI'90* (pp. 210–215).
20. Gyssens, M., Jeavons, P., & Cohen, D. A. (1994). Decomposing constraint satisfaction problems using database techniques. *Artificial Intelligence, 66*(1), 57–89.
21. Haralick, R. M., & Elliott, G. L. (1980). Increasing tree search efficiency for constraint satisfaction problems. *Artificial Intelligence, 14*, 263–313.
22. Jaffar, J., Michaylov, S., Stuckey, P., & Yap, R. (1992). The CLP(R) language and system. *ACM Transactions on Programming Languages and Systems, 14*(3), 339–395.
23. Katsirelos, G., & Walsh, T. (2007). A compression algorithm for large arity extensional constraints. In *Proceedings of CP'07* (pp. 379–393).
24. Lecoutre, C. (2008). Optimization of simple tabular reduction for table constraints. In *Proceedings of CP'08* (pp. 128–143).
25. Lecoutre, C., & Cardon, S. (2005). A greedy approach to establish singleton arc consistency. In *Proceedings of IJCAI'05* (pp. 199–204).
26. Lecoutre, C., Cardon, S., & Vion, J. (2007). Conservative dual consistency. In *Proceedings of AAAI'07* (pp. 237–242).
27. Lecoutre, C., & Hemery, F. (2007). A study of residual supports in arc consistency. In *Proceedings of IJCAI'07* (pp. 125–130).
28. Lecoutre, C., & Szymanek, R. (2006). Generalized arc consistency for positive table constraints. In *Proceedings of CP'06* (pp. 284–298).
29. Lecoutre, C., & Vion, J. (2008). Enforcing arc consistency using bitwise operations. *Constraint Programming Letters, 2*, 21–35.
30. Lhomme, O., & Régin, J. C. (2005). A fast arc consistency algorithm for n-ary constraints. In *Proceedings of AAAI'05* (pp. 405–410).
31. Likitvivatanavong, C., Zhang, Y., Bowen, J., & Freuder, E. C. (2004). Arc consistency in MAC: A new perspective. In *Proceedings of CPAI'04 workshop held with CP'04* (pp. 93–107).
32. Mackworth, A. K. (1977). Consistency in networks of relations. *Artificial Intelligence, 8*(1), 99–118.
33. Mackworth, A. K. (1977). On reading sketch maps. In *Proceedings of IJCAI'77* (pp. 598–606).
34. Mohr, R., & Masini, G. (1988). Good old discrete relaxation. In *Proceedings of ECAI'88* (pp. 651–656).
35. Pesant, G. (2004). A regular language membership constraint for finite sequences of variables. In *Proceedings of CP'04* (pp. 482–495).
36. Rossi, F., Petrie, C., & Dhar, V. (1990). On the equivalence of constraint satisfaction problems. In *Proceedings of ECAI'90* (pp. 550–556).
37. Sabin, D., & Freuder, E. C. (1994). Contradicting conventional wisdom in constraint satisfaction. In *Proceedings of CP'94* (pp. 10–20).
38. Samaras, N., & Stergiou, K. (2005). Binary encodings of non-binary constraint satisfaction problems: Algorithms and experimental results. *Journal of Artificial Intelligence Research, 24*, 641–684.
39. Stergiou, K., & Walsh, T. (1999). Encodings of non-binary constraint satisfaction problems. In *Proceedings of AAAI'99* (pp. 163–168).
40. Ullmann, J. R. (1977). A binary n-gram technique for automatic correction of substitution, deletion, insertion and reversal errors in words. *Computer Journal, 20*(2), 141–147.
41. Ullmann, J. R. (2007). Partition search for non-binary constraint satisfaction. *Information Science, 177*, 3639–3678.
42. van Dongen, M. R. C. (2006). Beyond singleton arc consistency. In *Proceedings of ECAI'06* (pp. 163–167).
43. van Hentenryck, P., & Ramachandran, V. (1995). Backtracking without trailing in CLP(R-lin). *ACM Transactions on Programming Languages and Systems, 17*(4), 635–671.
44. Xu, K., Boussemart, F., Hemery, F., & Lecoutre, C. (2005). A simple model to generate hard satisfiable instances. In *Proceedings of IJCAI'05* (pp. 337–342).
45. Xu, K., Boussemart, F., Hemery, F., & Lecoutre, C. (2007). Random constraint satisfaction: Easy generation of hard (satisfiable) instances. *Artificial Intelligence, 171*(8–9), 514–534.