Optimization of Simple Tabular Reduction for Table Constraints

Christophe Lecoutre

CRIL – CNRS UMR 8188,
Université Lille-Nord de France, Artois
rue de l'université, SP 16, F-62307 Lens
lecoutre@cril.fr

Abstract. Table constraints play an important role within constraint programming. Recently, many schemes or algorithms have been proposed to propagate table constraints or/and to compress their representation. We show that simple tabular reduction (STR), a technique proposed by J. Ullmann to dynamically maintain the tables of supports, is very often the most efficient practical approach to enforce generalized arc consistency within MAC. We also describe an optimization of STR which allows limiting the number of operations related to validity checking or search of supports. Interestingly enough, this optimization makes STR 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 the optimized algorithm we propose is usually around twice as fast as the original STR and can be up to one order of magnitude faster than previous state-of-the-art algorithms on some series of instances.

1 Introduction

Arc Consistency (AC) plays a central role in Constraint Programming (CP). It is a property of constraint networks which can be exploited to identify and remove 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, 11], weak k-singleton arc consistency [19] and conservative dual consistency [12].

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 GAC algorithms for positive table constraints. A positive (resp. negative) table constraint is a constraint that is defined in extension by a set of allowed (resp. disallowed) tuples. Table constraints are commonly used in configuration applications or applications related to databases. Moreover, table constraints play a particular role in constraint programming since they are easily handled by end-users of CP systems. Because any constraint can be theoretically translated into a table one (except that, in practice, this can lead to a time and space explosion), tables can be considered as the universal way of representing constraints.

In the last few years, many works have been devoted to table constraints. Many schemes or algorithms have been proposed to enforce GAC on table constraints or/and to compress their representation. In particular, one line of research (see [15, 13, 8]) aims to combine the two concepts of validity and acceptability of tuples of values. Significant formal and practical results have been obtained with respect to the very classical schemes [5]. Another recent proposal, called simple tabular reduction (STR) [18], 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 present the results of an extensive experimentation including both random and structured CSP instances which show that STR is quite competitive with respect to state-of-the-art algorithms (results in [18] were essentially given for random instances in the context of partition search). We can conclude that when GAC is maintained on table constraints during search, very often, STR is the most efficient approach. Second, we present an optimization of STR which allows limiting the number of validity checking and support search operations. Interestingly, we show that this optimization makes STR 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.

The paper is organized as follows. After introducing some technical background and related work, we present STR. Then, we describe how STR can be optimized. Before concluding, we present the results of an extensive experimentation.

2 Technical Background

A Constraint Network (CN) P is a pair $(\mathscr{X}, \mathscr{C})$ where $\mathscr{X}$ is a finite set of n variables and $\mathscr{C}$ a finite set of e constraints. Each variable $X \in \mathscr{X}$ has an associated domain, denoted $dom(X)$, that contains the set of values allowed for X. Each constraint $C \in \mathscr{C}$ involves an ordered subset of variables of $\mathscr{X}$ and has an associated relation, denoted $rel(C)$, which is the set of tuples allowed for this subset of variables. This subset of variables is the *scope* of C and is denoted $scp(C)$. The *arity* of a constraint is the number of variables in its scope. A *binary* constraint has arity 2.

A solution to a CN is an assignment of a value to each variable such that all the constraints are satisfied. A CN 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 CN is satisfiable or not. A CSP instance is defined by a CN 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.

A central example of inferential simplification is enforcement of GAC, which removes inconsistent values from domains without reducing the set of solutions of the CN. Values are inconsistent if they cannot occur in any solution. In some cases, enforcement of GAC can yield a solution directly, without any search.

Before giving a technical definition of GAC, we introduce the notion of support. Given an ordered set $\{X_1, \dots, X_i, \dots, X_k\}$ of k variables and a k-tuple $\tau = (a_1, \dots, a_i, \dots, a_k)$ of values, the individual value a_i will be denoted by $\tau[i]$ and also $\tau[X_i]$ by abuse of notation. If C is a k-ary constraint, then the k-tuple τ is said to be allowed by C iff $\tau \in rel(C)$; a valid tuple of C iff $\forall X \in scp(C)$, $\tau[X] \in dom(X)$; a support of C iff τ is a valid tuple of C which is allowed by C.

A pair (X, a) with $X \in \mathscr{X}$ and $a \in dom(X)$ will be called a *value* (of P). A tuple τ is a support for a value (X, a) in C iff $X \in scp(C)$ and τ is a support of C such that $\tau[X] = a$. Determining if a tuple is allowed is called a *constraint check* and determining if a tuple is valid is called a *validity check*. A value (X, a) of P is generalized arc-consistent (GAC) iff for every constraint C involving X, there exists a support for (X, a) in C. A variable X of P is GAC iff $dom(X) \neq \emptyset$ and $\forall a \in dom(X)$, (X, a) is GAC. P is GAC iff every variable of P is GAC.

It is easy to see that a value (X, a) of P which is not GAC cannot be included in any solution of P and is therefore inconsistent. Enforcing GAC means making the CN GAC by removing 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 use MAC which is a complete algorithm that is considered to provide the most efficient generic approach to the solution of CSP instances. MAC 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 *depth* of a node ν is the number of variable assignments performed along the path leading from the root of the search tree to ν. A *past* variable is (explicitly) assigned whereas a *future* variable is not (explicitly) assigned. $fut(C)$ is the set of future variables belonging to $scp(C)$. Finally, we emphasise that when GAC is enforced at a given step of the search, values can be removed only from domains of future variables.

3 Related Work on GAC for Table Constraints

A positive table constraint is a constraint given in extension and defined by a set of allowed tuples. Such constraints arise in practice in configuration problems, and more generally, in problems whose data come from databases. The set of allowed tuples associated with any constraint C is a table denoted by $C.table$. The worst-case space complexity of this table is $O(tr)$ where $t = |C.table|$ denotes the size of the table (i.e. the number of allowed tuples) and r denotes the arity of C.

GAC can be enforced by focusing in turn on each value in each domain; a value (X, a) is removed from its domain unless it is included in a support in every constraint that involves X. The classical generic GAC-valid scheme [5] seeks support by iterating over valid tuples (i.e. tuples that can be built from the current domains of constraint variables) until one is found that is allowed (i.e. accepted by the constraint). When working with table constraints we can instead seek support by iterating over allowed tuples until one is found to be valid [5]. From now on, these two schemes will be denoted by GACv and GACa, respectively. The efficiency of both approaches highly depends on the size of visited lists.

Recently, new schemes have been proposed, combining the exploitation of both valid and allowed tuples. In [15], it is shown that it is possible to skip over an exponential (in the arity) number of allowed tuples by reasoning from the current domains of variables. The key point is to know for each allowed tuple and each value the next tuple of the table that contains this value. To limit the space complexity of this approach which is in $O(trd)$, a sophisticated data structure, called hologram [14], can be used. To further improve the method, the authors propose to exploit lower bounds on supports such as, e.g., the $last$ structure employed in AC2001/3.1.

Another approach, proposed in [13], involves visiting, in turn, lists of valid and allowed tuples. The principle is to avoid considering irrelevant tuples (when a support is looked for) by jumping over sequences of valid tuples containing no allowed tuple and over sequences of allowed tuples containing no valid tuple. This approach admits on some instances a behaviour quadratic in the arity of the constraints whereas classical schemes (GACv and GACa) admit an exponential behaviour. Interestingly, this approach whose worst-case space complexity is $O(tr)$ can be easily grafted to any generic GAC algorithm.

More recently, two additional data structures have been introduced [8] for table constraints. The most promising one corresponds to the tree structure called trie. A trie is a multi-way tree structure useful for storing strings over an alphabet. It can then be used to store large dictionaries of words. The original proposal in [8] is to represent the set of tuples of a constraint by a trie, and to explore this trie when looking for supports. In order to keep cheap the search of supports, the authors suggest to build one trie per variable (of the scope of the constraint), the first level being dedicated to it. The worst-case space complexity of the trie approach is $O(tr^2)$ but as tuples are compressed at the top of each trie, one can expect a better memory usage in practice.

Finally, in a recent work [10], an algorithm has been proposed to compress table constraints. The principle is to represent the initial set of tuples by subsets of the Cartesian product of the domains of the variables involved in the constraint. Interestingly, this approach is also suitable to negative table constraints. Indeed, from an initial set of disallowed tuples, the authors show it is possible to build a set of compressed allowed tuples, whose size is at most nd times the size of the original set. Among related approaches, one can cite the use of directed acyclic graphs (DAGs) [6] and binary decision diagrams (BDDs) [7].

4 Simple Tabular Reduction

Simple tabular reduction (STR) [18] is another original approach to enforce GAC on positive table constraints. The principle of STR is to dynamically maintain the tables of allowed tuples. More precisely, whenever a value is removed from the domain of a variable, the table associated with a constraint involving this variable is updated, removing all tuples that have become invalid. Values which are no more GAC are then (easily) identified and removed. To summarize, GAC is enforced while removing invalid tuples, and consequently, only supports are kept in tables. One related work [16] is the AC algorithm proposed for the hidden variable encoding.

We now formulate in more detail an implementation of STR in which each constraint is an object. Recall that $C.table$ contains the initial set of tuples allowed by the constraint C. Without any loss of generality, we hold this set within an array. The tuple that is the i^{th} element of $C.table$, denoted by $C.table[i]$, can be accessed in constant time. Within $C.table$, every tuple is a member of exactly one of a set of linked lists of tuples. One of these lists links all tuples that are currently valid (and consequently are supports): tuples in this list constitute the contents of the *current table* of C. Any tuple of the current table of C will be called a *current* tuple. To provide access to the disjoint lists of tuples within $C.table$ we introduce the following additional fields for each constraint object C:

- $C.first$ is the position (i.e. the subscript) of the first current tuple in $C.table$. $C.first = -1$ if the current table of C is empty.
- $removedHead$ is an array of size n such that $C.removedHead[d]$ is the position of a first invalid tuple of $C.table$ that was removed when the search was at depth d. $C.removedHead[d] = -1$ if none was removed at depth d.
- $removedTail$ is an array of size n such that $C.removedTail[d]$ gives the position of a last invalid tuple removed at depth d. $C.removedTail[d]$ is relevant only if $C.removedHead[d] \neq -1$.
- $next$ is an array of size $t = |C.table|$ that is used for linking lists of tuples. More precisely, if i is the position of a current tuple of C, then $C.next[i]$ indicates the position of the next tuple in the current table. $C.next[i] = -1$ if i is the position of the last current tuple. Similarly, if i is the position of a tuple removed at depth d, then $C.next[i]$ indicates the position of the next tuple removed at depth d, except that $C.next[i] = -1$ if i is now the position of the last invalid tuple removed at depth d.

Besides, corresponding to each variable X, we provide a set $gacValues[X]$ [17] that will contain all values in $dom(X)$ which are proved to have a support when enforcing GAC on a constraint C. With a $O(d)$ space consumption per variable, one can guarantee that all elementary operations (determining if a value is present, adding/removing a value, etc.) are performed in constant time (see [9, 13]).

To enforce GAC at a given depth on a (positive table) constraint C using STR, Algorithm 1 is called. The loops at lines 1, 7 and 15 only iterate over future variables because it is only for these variables that it is possible to remove values from domains. This is an optimization wrt the original algorithm given in [18]. The sets $gacValues$ are emptied at lines 1 and 2 of Algorithm 1 because no value is initially guaranteed to be GAC. Then, all current tuples of the table of C are considered in turn by the loop at lines $4 - 14$. 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 are collected at lines 7 to 9. In constant time (see Algorithm 3), at line 13 an invalid tuple τ is removed from the current table and put (at first position) into the list of invalid tuples that were removed at the current depth d. Note that τ is removed without actually being moved in memory. Once all current tuples have been considered, unsupported values are removed (lines 15 to 20): these are the values in $dom(X) \setminus gacValues[X]$. If a domain becomes empty then $false$ is returned at line 18 because of inconsistency.

Algorithm 1: GACstr(C: Constraint, $depth$: Integer): Boolean

```
1  foreach variable $X \in fut(C)$ do
2      $gacValues[X] \leftarrow \emptyset$
3  $prev \leftarrow -1$ ; $curr \leftarrow C.first$
4  while $curr \neq -1$ do
5      $\tau \leftarrow C.table[curr]$
6      if $isValid(C, \tau)$ then
7          foreach variable $X \in fut(C)$ do
8              if $\tau[X] \notin gacValues[X]$ then
9                  add $\tau[X]$ to $gacValues[X]$
10         $prev \leftarrow curr$ ; $curr \leftarrow C.next[curr]$
11     else
12         $next \leftarrow C.next[curr]$
13         $removeTuple(C, prev, curr, depth)$
14         $curr \leftarrow next$
15 foreach variable $X \in fut(C)$ do
16     if $|gacValues[X]| \neq |dom(X)|$ then
17         if $gacValues[X] = \emptyset$ then
18             return false
19         $dom(X) \leftarrow gacValues[X]$
20         add $X$ to $propagationQueue$
21 return $true$
```

Algorithm 2: isValid(C: Constraint, τ: Tuple): Boolean

```
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, $prev, curr, depth$: Integers)

```
1  if $prev = -1$ then $C.first \leftarrow C.next[curr]$
2  else $C.next[prev] \leftarrow C.next[curr]$
3  $C.next[curr] \leftarrow C.removedHead[depth]$
4  if $C.removedHead[depth] = -1$ then $C.removedTail[depth] \leftarrow curr$
5  $C.removedHead[depth] \leftarrow curr$
```

Algorithm 4: restoreTuples(C: Constraint, $depth$: Integer)

```
1  if $C.removedHead[depth] \neq -1$ then
2      $C.next[C.removedTail[depth]] \leftarrow C.first$
3      $C.first \leftarrow C.removedHead[depth]$
4      $C.removedHead[depth] \leftarrow -1$
```

To enforce GAC on the constraint network, some events must be recorded: here, a variable is put in a queue dedicated to propagation (see line 20 of Algorithm 1) whenever its domain is reduced. Later, this variable will be picked from the queue, and all constraints involving this variable will be enforced to be GAC (a call to GACstr will be performed for a positive table constraint). Also, the code given here can be easily adapted to take into account finer propagation events.

The worst-case time complexity of GACstr (Algorithm 1) is $O(r'd+rt')$ where, for a given constraint C, $r' = |fut(C)|$ denotes the number of future variables in C and t' the size of the current table of C. Indeed, loops at lines 1, 4 and 15 are $O(r')$, $O(rt')$ and $O(r'd)$, respectively. The worst-case space complexity of GACstr is $O(n + rt)$ per constraint since $removedHead$ and $removedTail$ are $O(n)$, $table$ is $O(rt)$ and $next$ is $O(t)$.

Importantly, when backtracking occurs, values must be restored to domains, as is well known, and because of domain restoration, tuples that were invalid may now be valid. If a tuple τ was removed from the current table of C at depth d, then τ must be restored to the current table of C when the search backtracks to depth d or assigns a new value at depth d. In our implementation, tuples are restored by calling Algorithm 4. This algorithm puts the list of invalid tuples removed at the given depth at the head of the current table. Restoration is achieved in constant time (for each constraint) without traversing either list and without moving any tuple in memory [18].

5 Optimizing STR

It is possible to improve STR in two directions. First, as soon as all values in the domain of a variable have been detected GAC, it is futile to continue to seek supports for values of this variable. We therefore introduce a set, S^{sup}, of variables in $fut(C)$ whose domain contains at least one value for which a support has not yet been found. In GACstr2 (Algorithm 5), which is an optimized version of GACstr, lines 1, 6 and 8 initialize S^{sup} to be the same as $fut(C)$. Whenever a support is found for the last unsupported value in the domain of a variable X, line 20 removes X from S^{sup}. If $|gacValues[X]| = |dom(X)|$ at line 19 then all values of $dom(X)$ are supported. Efficiency is gained by iterating only over variables in S^{sup} at lines 16 and 26.

The second direction of improvement avoids unnecessary validity operations. At the end of an invocation of GACstr for constraint C, every tuple τ such that $\tau[X] \notin dom(X)$ (with $X \in scp(C)$) 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 this 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 future variables whose domain has been reduced since the previous invocation of GACstr2. Initially, this set also contains the last assigned variable (denoted by $lastAssignedVariable$ here) if it belongs to the scope of the constraint C. Indeed, after any variable assignment $X = a$, some tuples may become invalid due to the removal of some values in $dom(X)$. This is the only past variable for which validity operations must be performed. Algorithm 6 checks validity only for variables in S^{val}. The set S^{val} is first initialized at lines 2 through 5 of Algorithm 5. At line 9 of this algorithm, $dom(X).getLastRemovedValue()$ is the value that was most recently removed from $dom(X)$ whilst processing this or any other constraint. A special value nul must be used when no value has been removed. $C.lastRemoved[X]$ is the value that was most recently removed from $dom(X)$ whilst processing the specific constraint C (see lines 11 and 30). If these two values differ at line 9 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 10. This is how the membership of S^{val} is determined.

The worst-case time complexity of GACstr2 is $O(r'(d+t'))$. Indeed, performing a validity check is now $O(r')$ instead of $O(r)$, as it can be observed from Algorithm 6. Moreover, the loop starting at line 13 is in $O(r't')$. Like GAcstr, the worst-case space complexity of GACstr2 is $O(n+rt)$ per constraint since data structures inherited from GACstr are $O(n+rt)$ and $lastRemoved$ is $O(r)$; S^{sup} and S^{val} are also $O(r)$ but may be shared by all constraints.

The worst case scenarii used to develop the worst-case time complexities of both GACstr and GACstr2 do not entirely characterize the difference in behaviour 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)
...
```

In this example, the domain of each variable involved in C comprises all digits from 0 to $d-1$. In the table, the first $d-1$ tuples are sequences formed with the same digit (from 0 to $d-2$), while the d^{th} tuple consists of the digit $d-2$ followed by a sequence of $d-1$. Assume that all variables are future, that STR (either of the two algorithms) is applied to this constraint and that no value is removed because all values are present in domains and there also exists a support for $(X_1, d-1)$ in C (although this is not shown above). 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), but some tuples (at least one) will be eliminated. Interestingly, calling GACstr 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 GACstr2 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 is summarized as follows:

Observation 1 *There exist situations where applying GACstr to a r-ary constraint C is $O(rt + rd)$ whereas applying GACstr2 is $O(t + rd)$.*

Most of the time, $d << t$ since $t \in O(d^r)$. In this case, Observation 1 shows that GACstr2 is potentially r times faster than GACstr. The higher the arity, the greater the

Algorithm 5: GACstr2(C: Constraint, $depth$: Integer): Boolean

```
1  $S^{sup} \leftarrow \emptyset$
2  if $lastAssignedVariable \notin scp(C)$ then
3      $S^{val} \leftarrow \emptyset$
4  else
5      $S^{val} \leftarrow \{lastAssignedVariable\}$
6  foreach variable $X \in fut(C)$ do
7      $gacValues[X] \leftarrow \emptyset$
8      $S^{sup} \leftarrow S^{sup} \cup \{X\}$
9      if $dom(X).getLastRemovedValue() \neq C.lastRemoved[X]$ then
10         $S^{val} \leftarrow S^{val} \cup \{X\}$
11         $C.lastRemoved[X] \leftarrow dom(X).getLastRemovedValue()$
12 $prev \leftarrow -1$ ; $curr \leftarrow C.first$
13 while $curr \neq -1$ do
14     $\tau \leftarrow C.table[curr]$
15     if $isValid(C, \tau)$ then
16         foreach variable $X \in S^{sup}$ do
17             if $\tau[X] \notin gacValues[X]$ then
18                 add $\tau[X]$ to $gacValues[X]$
19                 if $|gacValues[X]| = |dom(X)|$ then
20                     $S^{sup} \leftarrow S^{sup} \setminus \{X\}$
21         $prev \leftarrow curr$ ; $curr \leftarrow C.next[curr]$
22     else
23         $next \leftarrow C.next[curr]$
24         $removeTuple(C, prev, curr, depth)$
25         $curr \leftarrow next$
26 foreach variable $X \in S^{sup}$ do
27     if $gacValues[X] = \emptyset$ then
28         return false
29     $dom(X) \leftarrow gacValues[X]$
30     $C.lastRemoved[X] \leftarrow dom(X).getLastRemovedValue()$
31     add $X$ to $propagationQueue$
32 return $true$
```

Algorithm 6: isValid(C: Constraint, τ: Tuple): Boolean

```
1  foreach variable $X \in S^{val}$ do
2      if $\tau[X] \notin dom(X)$ then
3          return false
4  return true
```

benefit of using GACstr2 may be. Finally, one may wonder about backtracking issues. A first solution, when backtracking occurs, is to reinitialize all arrays $lastRemoved$, filling them with the special value nul. Alternatively, one may record the content of such arrays at each depth of search. Upon backtracking, one can then benefit from the original state of the arrays. This approach, which requires an additional structure that is $O(nr)$ per constraint, will be denoted by GACstr2+.

6 Experimental Results

In order to show the practical interest of simple tabular reduction, and in particular the optimization we propose, we have experimented using a cluster of Xeon 3.0GHz with 1GiB of RAM under Linux, employing MAC with $dom/ddeg$ and $lexico$ as variable[1] and value ordering heuristics, respectively. We have compared classical schemes to enforce GAC on (positive) table constraints with STR. More precisely, we have implemented GACv and GACa (see Section 3) as well as the scheme described in [13], denoted by GACva here. We do believe that GACva is a representative state-of-the-art algorithm for table constraints. Our own experience confirms the results reported in [8]: GACva and the trie approach are quite robust and close in terms of performance.

We have performed a first experimentation with random CSP instances. We have generated different classes of instances from Model RD [20]. 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. Whatever the arity $r \geq 8$ is, Theorem 2 [20] 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 1 shows the mean cpu time required 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 STR algorithms, Figures 2 and 3 clearly confirm the general observation made in Section 5 about the increasing interest of using GACstr2(+) when the arity of the constraints increases. Indeed, while GACstr2+ is about 20% faster than GACstr (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 are available from `http://www.cril.univ-artois.fr/~lecoutre/`. These series represent a large spectrum of instances, and importantly, allow anyone to easily reproduce our experimentation. The two first series [7] bdd-21-2713-15 and bdd-21-133-18 contain 35 instances each, involving 21 Boolean variables and large and small BDD constraints of arity 15 and 18, respectively. The series renault-mod contains 45 real-world instances (we were unable to solve 5 of them with the selected heuristics within a reasonable amount of time) 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). The two series tsp-20 and tsp-25 contain 15 instances of the Travelling Salesperson Problem each, involving domains containing up to $1,000$

[1] In our implementation, using $dom/wdeg$ does not guarantee exploring the same search tree with classical and STR schemes.

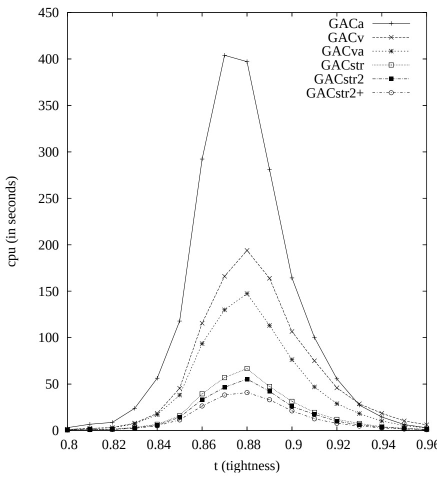

Fig. 1. Mean search cost of solving instances in classes $\langle 13, 60, 2, 20, t\rangle$ with MAC.

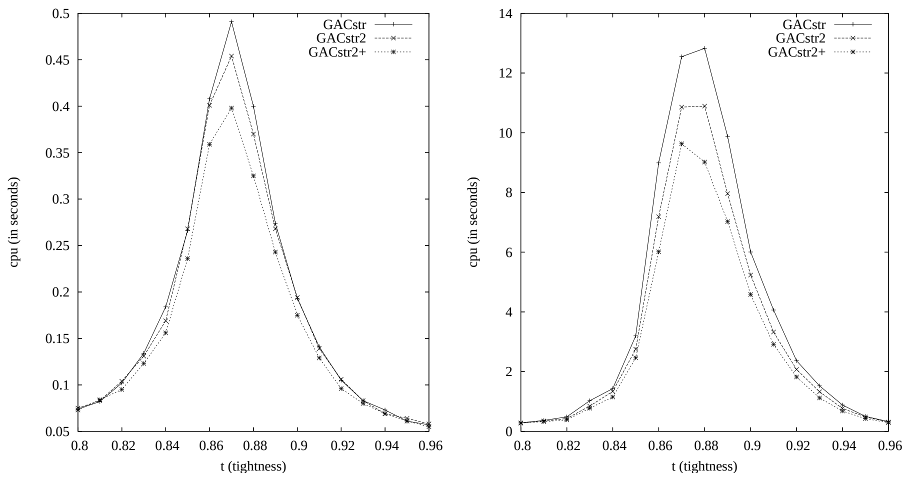

Fig. 2. Mean search cost for classes $\langle 10, 60, 2, 20, t\rangle$ (left) and $\langle 12, 60, 2, 20, t\rangle$ (right).

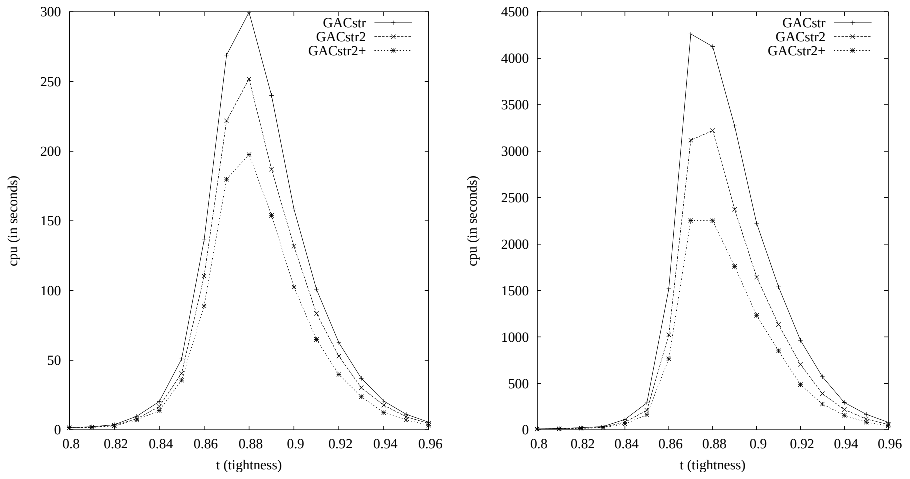

Fig. 3. Mean search cost for classes $\langle 14, 60, 2, 20, t\rangle$ (left) and $\langle 16, 60, 2, 20, t\rangle$ (right).

		Classical GAC schemes			Simple Tabular Reduction		
Series	#Inst	$GACv$	$GACa$	$GACva$	$GACstr$	$GACstr2$	$GACstr2+$
bdd-21-2713-15	35	69.3	386	58.8	164	94.5	52.1
bdd-21-133-18	35	37.3	(23 out)	36.0	66.1	38.3	26.2
renault-mod	45	83.8	45.7	48.0	61.6	54.9	45.4
tsp-20	15	28.4	23.3	14.9	8.80	8.95	8.35
tsp-25	15	254	273	196	119	122	118
rand-8-20-5-18	20	107	(16 out)	119	108	81.2	65.6
rand-10-20-10-5	20	(20 out)	4.49	5.61	1.00	0.77	0.53

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

values and ternary constraints defined by large tables (about $20,000$ 3-tuples). Finally, the two series rand-8-20-5-18 and rand-10-20-10-5 contain 20 random instances each involving 20 variables. Each instance of the series rand-8-20-5-18 (resp., rand-10-20-10-5) involves domains containing 5 (resp., 10) values and 18 (resp., 5) constraints of arity 8 (resp., 10). Tables contain about about $78,000$ and $10,000$ tuples, respectively.

Table 1 indicates the mean cpu time required to solve the instances of these different series. Overall, we can observe that GACstr2+ is always the most efficient approach. In particular, GACstr2+ is 3 times faster than GACstr on the bdd-21-2713-15 series and 10 times faster than GACva on the rand-10-20-10-5 series. Table 2 presents the results obtained on some representative instances. Here, for each series, we show the results for 2 or 3 instances of various difficulty. For example, the instance bdd-21-133-18-10 only requires visiting 21 nodes (to be solved) whereas the instance bdd-21-133-18-11 requires visiting $14,716$ nodes. Typically, when the instance is easy, using STR is rather penalising. This is not really surprising since managing dynamic tables is then just an overhead. This is particularly visible for easy instances of series bdd-21-2713-15 and bdd-21-133-18. 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 $O(nd)$ required by GACstr2+ has a very limited practical impact on all these series.

Finally, we have tested the series of Crossword puzzles that have been recently posted at the web page mentioned earlier. For each grid, there is a variable per white square which can be assigned any of the 26 letters of the Latin alphabet, and a constraint for any sequence of white squares which corresponds to a word that we must put in the grid. Such constraints are defined by a table which contains all words of the right length. The series prefixed by cw-m1c are defined from blank grids and only contain positive table constraints (contrary to model m1 in [1] where no two identical words can be put in the grid, which is then naturally expressed in intension). The arity of the constraints is given by the size of the grids: for example, cw-m1c-lex-vg5-6 involves table constraints of arity 5 and 6 (the grid being 5 by 6). The results that we have obtained (see Table 3) with respect to 4 dictionaries (lex, words, uk, ogd) of different length confirm our previous results. On the most difficult instances, GACstr2+ is about two times faster than GACstr and one order of magnitude faster than GACva. Note that we do not provide mean results for these series because many instances cannot be solved within $1,200$ seconds.

		Classical GAC schemes			Simple Tabular Reduction		
Instance		$GACv$	$GACa$	$GACva$	$GACstr$	$GACstr2$	$GACstr2+$
bdd-21-133-18-10	cpu	0.82	0.93	0.93	7.18	3.62	3.57
#nodes=21	mem	$39M$	$43M$	$43M$	$63M$	$63M$	$63M$
bdd-21-133-18-2	cpu	38.7	$> 1,200$	38.7	68.1	38.6	25.9
#nodes=$10,660$	mem	$61M$		$72M$	$127M$	$127M$	$126M$
bdd-21-133-18-11	cpu	58.4	$> 1,200$	53.6	104	61.1	43.9
#nodes=$14,716$	mem	$46M$		$59M$	$100M$	$101M$	$101M$
bdd-21-2713-15-22	cpu	0.81	0.74	0.81	13.8	5.85	5.97
#nodes=21	mem	$91M$	$93M$	$93M$	$165M$	$166M$	$175M$
bdd-21-2713-15-32	cpu	61.5	357	55.1	145	82.7	44.5
#nodes=$1,140$	mem	$73M$	$74M$	$74M$	$166M$	$167M$	$176M$
bdd-21-2713-15-35	cpu	78.6	372	71.9	193	121	66.1
#nodes=$1,465$	mem	$73M$	$74M$	$74M$	$167M$	$168M$	$177M$
renault-mod-0	cpu	11.1	1.05	1.04	1.05	0.99	1.04
#nodes=287	mem	$36M$	$41M$	$41M$	$34M$	$34M$	$34M$
renault-mod-12	cpu	149	92.2	88.9	92.4	83.7	77.6
#nodes=$415K$	mem	$39M$	$52M$	$52M$	$49M$	$49M$	$50M$
renault-mod-14	cpu	411	321	318	384	359	302
#nodes=$1,135K$	mem	$40M$	$51M$	$51M$	$66M$	$66M$	$68M$
tsp-20-190	cpu	6.02	6.91	5.56	4.89	4.98	4.59
#nodes=$7,738$	mem	$12M$	$12M$	$12M$	$10M$	$10M$	$10M$
tsp-20-366	cpu	37.0	41.6	32.9	25.2	25.7	23.5
#nodes=$31,701$	mem	$10M$	$10M$	$9,731K$	$9,115K$	$9,124K$	$9,261K$
tsp-20-193	cpu	291	207	146	99.2	101	91.6
#nodes=$80,849$	mem	$16M$	$17M$	$16M$	$17M$	$17M$	$17M$
tsp-25-13	cpu	4.23	3.2	3.03	3.03	3.07	2.86
#nodes=$2,421$	mem	$20M$	$20M$	$20M$	$17M$	$17M$	$17M$
tsp-25-163	cpu	178	205	140	108	105	105
#nodes=$89,883$	mem	$15M$	$15M$	$14M$	$15M$	$15M$	$16M$
tsp-25-456	cpu	$1,060$	$1,140$	813	643	642	683
#nodes=$686K$	mem	$28M$	$28M$	$26M$	$40M$	$40M$	$42M$
rand-10-20-10-5-10	cpu	$> 1,200$	3.86	2.59	0.58	0.51	0.43
#nodes=794	mem		$20M$	$20M$	$16M$	$16M$	$16M$
rand-10-20-10-5-0	cpu	$> 1,200$	4.59	3.22	1.19	1.36	0.83
#nodes=826	mem		$23M$	$23M$	$20M$	$20M$	$20M$
rand-8-20-5-18-10	cpu	42.7	$> 1,200$	51.8	50.9	39.4	31.8
#nodes=$57,579$	mem	$193M$		$283M$	$205M$	$205M$	$205M$
rand-8-20-5-18-13	cpu	420	$> 1,200$	403	241	186	153
#nodes=$569K$	mem	$196M$		$291M$	$221M$	$221M$	$221M$

Table 2. Representative results obtained on various structured and random instances. 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.

		Classical GAC schemes			Simple Tabular Reduction		
		$GACv$	$GACa$	$GACva$	$GACstr$	$GACstr2$	$GACstr2+$
Crossword puzzles with dictionary lex ($24,974$ words)							
cw-m1c-lex-vg5-6	*cpu*	$> 1,200$	38.8	54.2	14.3	12.4	10.7
#nodes=$26,679$	*mem*		$2,889K$	$2,928K$	$2,932K$	$2,935K$	$2,968K$
cw-m1c-lex-vg5-7	*cpu*	$> 1,200$	357	875	134	114	96.3
#nodes=$171K$	*mem*		$4,134K$	$4,173K$	$8,005K$	$8,055K$	$8,059K$
cw-m1c-lex-vg6-6	*cpu*	$> 1,200$	2.98	4.29	1.28	1.05	0.91
#nodes=$1,602$	*mem*		$4,422K$	$4,344K$	$4,226K$	$4,203K$	$4,296K$
cw-m1c-lex-vg6-7	*cpu*	$> 1,200$	436	$1,174$	176	143	118
#nodes=$152K$	*mem*		$5,887K$	$5,692K$	$9,458K$	$9,437K$	$9,555K$
Crossword puzzles with dictionary words ($45,371$ words)							
cw-m1c-words-vg5-5	*cpu*	$> 1,200$	0.04	0.05	0.05	0.05	0.04
#nodes=38	*mem*		$4,969K$	$4,987K$	$4,823K$	$4,791K$	$4,809K$
cw-m1c-words-vg5-6	*cpu*	$> 1,200$	1.19	1.46	0.48	0.37	0.33
#nodes=718	*mem*		$6,508K$	$6,526K$	$6,348K$	$6,273K$	$6,348K$
cw-m1c-words-vg5-7	*cpu*	$> 1,200$	18.6	36.0	6.61	5.21	4.03
#nodes=$6,957$	*mem*		$8,470K$	$8,489K$	$8,276K$	$8,145K$	$8,237K$
cw-m1c-words-vg5-8	*cpu*	$> 1,200$	866	$> 1,200$	273	229	187
#nodes=$256K$	*mem*		$4,604K$		$10M$	$10M$	$10M$
Crossword puzzles with dictionary uk ($225,349$ words)							
cw-m1c-uk-vg5-5	*cpu*	$> 1,200$	0.05	0.05	0.1	0.07	0.07
#nodes=28	*mem*		$12M$	$12M$	$12M$	$12M$	$12M$
cw-m1c-uk-vg5-6	*cpu*	$> 1,200$	0.55	0.5	0.21	0.17	0.17
#nodes=145	*mem*		$17M$	$17M$	$16M$	$16M$	$16M$
cw-m1c-uk-vg5-7	*cpu*	$> 1,200$	2.97	5.18	0.51	0.37	0.34
#nodes=408	*mem*		$22M$	$22M$	$22M$	$22M$	$22M$
cw-m1c-uk-vg5-8	*cpu*	$> 1,200$	82.5	71.9	7.08	5.71	4.78
#nodes=$8,148$	*mem*		$12M$	$12M$	$11M$	$11M$	$11M$
Crossword puzzles with dictionary ogd ($435,705$ words)							
cw-m1c-ogd-vg6-6	*cpu*	$> 1,200$	0.37	0.31	0.23	0.17	0.15
#nodes=98	*mem*		$46M$	$47M$	$46M$	$46M$	$48M$
cw-m1c-ogd-vg6-7	*cpu*	$> 1,200$	95.3	56.1	12.0	8.01	6.81
#nodes=$9,522$	*mem*		$11M$	$11M$	$11M$	$11M$	$11M$
cw-m1c-ogd-vg6-8	*cpu*	$> 1,200$	53.0	6.44	2.91	2.0	1.72
#nodes=$2,806$	*mem*		$24M$	$23M$	$22M$	$22M$	$24M$
cw-m1c-ogd-vg6-9	*cpu*	$> 1,200$	727	214	35.1	25.1	19.1
#nodes=$23,283$	*mem*		$42M$	$41M$	$39M$	$37M$	$40M$

Table 3. Representative results obtained on series of Crossword puzzles using dictionaries of different length. 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.

7 Conclusion

Simple tabular reduction (STR) [18] is a simple and effective GAC algorithm for positive table constraints. In this paper, we have proposed an optimization of this algorithm which significantly improves its efficiency. This new algorithm (GACstr2+) appears among state-of-the-art GAC algorithms for non-binary table constraints.

Acknowledgments

We would like to thank Julian Ullmann for fruitful discussions and helpful comments.

References

1. A. Beacham, X. Chen, J. Sillito, and P. van Beek. Constraint programming lessons learned from crossword puzzles. In *Proceedings of Canadian Conference on AI*, pages 78–87, 2001.
2. C. Bessiere. Constraint propagation. In *Handbook of Constraint Programming*, 2006.
3. C. Bessiere and R. Debruyne. Optimal and suboptimal singleton arc consistency algorithms. In *Proceedings of IJCAI'05*, pages 54–59, 2005.
4. C. Bessiere, E. Hebrard, B. Hnich, and T. Walsh. The complexity of reasoning with global constraints. *Constraints*, 12(2):239–259, 2007.
5. C. Bessiere and J. Régin. Arc consistency for general constraint networks: preliminary results. In *Proceedings of IJCAI'97*, pages 398–404, 1997.
6. M. Carlsson. Filtering for the case constraint. Samos, Greece, 2006. Talk given at Advanced School on Global constraints.
7. K. Cheng and R. Yap. Maintaining generalized arc consistency on ad-hoc n-ary Boolean constraints. In *Proceedings of ECAI'06*, pages 78–82, 2006.
8. 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.
9. P. Van Hentenryck, Y. Deville, and C.M. Teng. A generic arc-consistency algorithm and its specializations. *Artificial Intelligence*, 57:291–321, 1992.
10. G. Katsirelos and T. Walsh. A compression algorithm for large arity extensional constraints. In *Proceedings of CP'07*, pages 379–393, 2007.
11. C. Lecoutre and S. Cardon. A greedy approach to establish singleton arc consistency. In *Proceedings of IJCAI'05*, pages 199–204, 2005.
12. C. Lecoutre, S. Cardon, and J. Vion. Conservative dual consistency. In *Proceedings of AAAI'07*, pages 237–242, 2007.
13. C. Lecoutre and R. Szymanek. Generalized arc consistency for positive table constraints. In *Proceedings of CP'06*, pages 284–298, 2006.
14. O. Lhomme. Arc-consistency filtering algorithms for logical combinations of constraints. In *Proceedings of CPAIOR'04*, pages 209–224, 2004.
15. O. Lhomme and J.C. Régin. A fast arc consistency algorithm for n-ary constraints. In *Proceedings of AAAI'05*, pages 405–410, 2005.
16. N. Samaras and K. Stergiou. Binary encodings of non-binary constraint satisfaction problems: algorithms and experimental results. *JAIR*, 24:641–684, 2005.
17. J.R. Ullmann. A binary n-gram technique for automatic correction of substitution, deletion, insertion and reversal errors in words. *Computer Journal*, 20(2):141–147, 1977.
18. J.R. Ullmann. Partition search for non-binary constraint satisfaction. *Information Science*, 177:3639–3678, 2007.
19. M.R.C. van Dongen. Beyond singleton arc consistency. In *ECAI'06*, pages 163–167, 2006.
20. K. Xu, F. Boussemart, F. Hemery, and C. Lecoutre. Random constraint satisfaction: easy generation of hard (satisfiable) instances. *Artificial Intelligence*, 171(8-9):514–534, 2007.