Improving the Expressiveness of Table Constraints

Jean-Charles Régin

Université Nice-Sophia Antipolis, I3S UMR CNRS 6070
2000, route des Lucioles - Les Algorithmes - BP 121
06903 Sophia Antipolis Cedex - France.
jcregin@gmail.com

Abstract. The time complexity of algorithms establishing arc consistency for table constraints depends mainly on the number of involved tuples. Thus, several work have been carried out in order to decrease that number. Notably, Katsirelos and Walsh have proposed to compress the tuple set. They integrate their method into GAC-Schema and they showed how a set of f forbidden tuples may be represented by a set of ndf compressed tuple, where n is the constraint arity and d the size of the largest domain. However, they left open the integration of their method into the best improvements of GAC-Schema like Lhomme and Régin's one. In this paper, we introduce a generalization of compressed tuples: the tuple sequences. They improve the expressiveness of table constraint, because they can represent combinations of allowed and forbidden tuples. We provide an algorithm transforming f forbidden tuples into $f+1$ allowed tuple sequences which may eliminate the need for specific version of GAC-Schema for forbidden tuples. We also give an algorithm showing how allowed tuple sequences and forbidden tuples may be transformed into a set of tuples that are all allowed. At last, we show how to benefit from the best implementation of GAC-Schema.

1 Introduction

Table constraints are ones of the most used constraints in CP. They can be defined from the list of allowed combinations of values of variables (allowed tuples), by the forbidden tuples or from any Boolean function corresponding to the constraint satisfaction.

For establishing arc consistency a generic schema has been developed: GAC-Schema [1]. This algorithm is mainly based on the traversal of allowed tuples until a valid one (i.e. each value of the tuple is in the domain of its variable) is found, and its complexity depends on the number of tuples. Thus, some recent works have been carried out for restructuring the tuples [4–6, 2]. Notably, Katsirelos and Walsh [5] have proposed to compress the set of allowed tuples into Global Cut Seed (GCS tuples [3]). Such a set of tuples contains all the tuples of the Cartesian Product of subsets of the domains of the variables. For instance the compressed tuple $(\{1,2\},\{2\},\{1,2\})$ is equivalent to the set of tuples $\{(1,2,1),(1,2,2),(2,2,1),(2,2,2)\}$. By compressing the list of allowed tuples they showed that the establishing of arc consistency may be sped up. In addition, they explained how their algorithm can be integrated into GAC-Schema.

Katsirelos and Walsh's work is mainly focused on the algorithm computing this tuple compression. They were not really interested in the expressiveness of the compressed tuples. They have just considered this point for representing forbidden tuples by a set of allowed tuples. Unfortunately, their method may require nd times more compressed tuples than the number of forbidden tuples, where n is the constraint arity and d the size of the largest domain. In this paper, we propose to show how a simple generalization of GCS tuples, that we will call tuple sequences, is much more convenient for representing groups of tuples. A tuple sequence is formed by a GCS tuple, a minimum tuple and a maximum tuple w.r.t. an ordering. The advantage of these tuple sets is that they can represent easily forbidden tuples because it becomes easy to represent the complement of a set.

For instance, consider a Table constraint defined on variables having domains equal to $\{a, b, c, d\}$ by the list of forbidden tuples (a, b, c, d), (b, c, d, a) and (d, d, a, a). For convenience, we consider the lexicographic order. We can define a tuple sequence representing the tuple set from the tuple (a, a, a, a) to the tuple (a, b, c, c), which is the tuple immediately preceding the forbidden tuple (a, b, c, d). Such a tuples sequence is simply expressed by $min = (a, a, a, a)$, $max = (a, b, c, c)$ and the GCS tuple $(\{a, b, c, d\}, \{a, b, c, d\}, \{a, b, c, d\}, \{a, b, c, d\})$. We will write it:
$((a, a, a, a), (a, b, c, c), (\{a, b, c, d\}, \{a, b, c, d\}, \{a, b, c, d\}, \{a, b, c, d\}))$. Of course we could avoid having other values than a for the first variable but this is not even necessary. Then, we can repeat this method by defining the tuple sequence $((a, b, d, a), (b, c, c, d), (\{a, b, c, d\}, \{a, b, c, d\}, \{a, b, c, d\}, \{a, b, c, d\}))$ representing the tuples from (a, b, d, a) to (b, c, c, d), and a tuple sequence representing the tuples from (b, c, d, b) to (d, c, d, d), and a tuple sequence representing the tuples from (d, d, a, b) to (d, d, d, d). Thus, we can represent f forbidden tuples with only $f + 1$ tuple sequences. This improves Katsirelos and Walsh's method by a factor of nd.

We will also show that tuple sequences may also be quite useful for improving the expressiveness of Table constraints, because they can mix together allowed and forbidden tuples.

`Outline.` The paper is organized as follows. First, we recall some definition. Then, we introduce the notion of tuple cluster and define tuple sequences. Next we will explain how to represent a set of forbidden tuples only by allowed tuples sequences and we will show how tuple sequences may improve the expressiveness of Table constraints. Before concluding, we will detail the integration of tuple sequences into the best implementation of GAC-Schema.

2 Preliminaries

Constraint network. A finite *constraint network* $\mathcal{N} = (X, \mathcal{D}, \mathcal{C})$ is defined as a set of n *variables* $X = \{x_1, \ldots, x_n\}$, a set of *domains* $\mathcal{D} = \{D(x_1), \ldots, D(x_n)\}$

where $D(x_i)$ is the finite set of possible *values* for variable x_i, and a set $\mathcal{C}$ of *constraints* between variables. A value a for a variable x is often denoted by (x, a).
Constraint. A constraint C on the ordered set of variables $X(C) = (x_{i_1}, \ldots, x_{i_r})$ is a subset $T(C)$ of the Cartesian product $D(x_{i_1}) \times \cdots \times D(x_{i_r})$ that specifies the *allowed* combinations of values for the variables $x_{i_1}, \ldots, x_{i_r}$. An element of $D(x_{i_1}) \times \cdots \times D(x_{i_r})$ is called a *tuple on* $X(C)$ and $\tau[x]$ is the value of τ assigned to x. The tuples on $X(C)$ not allowed by C are called the *forbidden* tuples of C. $|X(C)|$ is the *arity* of C. W.l.o.g we will consider that the allowed tuples are ordered by the relation $\prec$. We will denote by d the size of the largest initial domain and by n the arity.
Arc consistency. Let C be a constraint. A tuple τ on $X(C)$ is *valid* iff $\forall x \in X(C), \tau[x] \in D(x)$; and τ is a *support* for (x, a) iff $\tau[x] = a$ and $\tau \in T(C)$. A value $a \in D(x)$ is *consistent with* C iff $x \notin X(C)$ or there exists a valid support for (x, a). C is *arc consistent* iff $\forall x \in X(C), D(x) \neq \emptyset$ and $\forall a \in D(x)$, a is consistent with C.
Lexicographic ordering. A total ordering $<_d$ can be defined on $D(x), \forall x \in X$, without loss of generality. Two tuples τ and τ' on $X(C)$ can be ordered by the natural lexicographic order $\prec_{lex}$ in which $\tau \prec_{lex} \tau'$ iff $\exists k$ such that $\forall j \in [1..k-1], \tau[x_{i_j}] = \tau'[x_{i_j}]$ and $\tau[x_{i_k}] <_d \tau'[x_{i_k}]$.

3 Expressiveness

There are several ways to define Table constraints[1]:

• the tuples satisfying the constraints come from a database.

• all the solutions of a subproblem are computed and they form the allowed tuples of the constraint.

• the user wants to express directly some compatibilities and incompatibilities between combinations of values. In this case, it is useful to be able to express that some combinations of a subset of values are allowed. For instance (x_1, a) can be combined only with (x_2, b), or the values of $\{a, b, c\}$ for the variable x_3 can be combined with the value $\{c, d\}$ of the variables x_5. At the same time, it is convenient to forbid some other combinations.

For the first way, the compression methods are certainly the best methods for representing these tuples. This method could also be used in the second way, but it could be worthwhile to try to compute the solution set in such a way it could be compressed. In other words, the enumeration method should try to compress the tuples while enumerating all solutions.

We propose to focus our attention on the latter point.

The main idea of this paper, is to introduce some structure in the definition of Table constraints. Since we can establish arc consistency for them, even if it

[1] It is also possible to define the Table constraint from a predicate, but this method is not really used in practice. Therefore, we will not consider it.

may be costly, then, with Table constraints we can easily express some complex constraints and benefit from powerful filtering algorithms. The drawback is that the number of tuples may be large and this prevents the definition of Table constraints for some complex constraints.

We propose to introduce a new notion: tuple clusters in order to try to keep the facility to express some constraints with the efficiency of a generic filtering algorithm. In other words, tuple clusters reduce the number of tuples but keep the easy way of expressing some constraints.

3.1 Tuple cluster

A tuple cluster is a general notion which encapsulates the notion of group of tuples provided that we can use them efficiently in conjunction with GAC-Schema.

Definition 1 *A* **tuple cluster** *is a data structure corresponding to a set of tuples and associated with an efficient operation (i.e. linear in the size of the data structure) returning a valid tuple of this set if there exists and nil otherwise.*

Global Cut Seed (GCS tuple) proposed by Focacci and Milano [3] is an example of tuple cluster. A GCS tuple is equivalent to the set of tuples obtained by performing the Cartesian Product of subsets of the domains of the variables. For instance the compressed tuple $(\{a,b\},\{b,c\},\{a,d\})$ is equivalent to the set of tuples $\{(a,b,a),(a,b,d),(a,c,a),(a,c,d),(b,b,a),(b,b,d),$
$(b,c,a),(b,c,d)\}$.

It is easy to check whether a GCS tuple contains a valid tuple or not: if there is no empty intersection between the domains of the GCS tuple and the corresponding current domains, then a valid tuple is contained in the GCS tuple. It is also easy to check whether a GCS tuple contains a support for the value (x,a). We simply set the current domain of x to $\{a\}$ and compute the intersections between domains. If there is no empty intersection then the GCS tuple contains a valid tuple involving (x,a), that is a support for (x,a). This operation may be done in $O(nd)$ in the worst case, because computing if an intersection of two domains is not empty can be done in $O(d)$, for domains of size d. We will see later that we can refine this point.

We propose a generalization of GCS tuples: the tuple sequences. In a GCS tuple there is no ordering and we have some problems to express some subsets of the tuples obtained by the Cartesian product of domains. Tuple sequences remedy to this drawback by introducing boundaries in this enumeration. A tuple sequence is just a GCS associated with a minimum and a maximum tuple w.r.t. the lexicographic ordering. In other words, it contains all the tuples of the GCS that are greater than or equal to the minimum tuple and smaller than or equal to the maximum tuple. Note that a similar introduction of minimum and maximum tuples for guiding the enumeration of set variables has been proposed by Yip and Van Hentenryck [8]. Here is the formal definition of a tuple sequence:

Definition 2 *Let C be a Table constraint defined on $X(C)=\{x_1,...,x_n\}$.*
A **tuple sequence** *s defined by the triplet*

$(\tau^s_{min}, \tau^s_{max}, (D^s(x_1), D^s(x_2), ..., D^s(x_n)))$ *is the set of tuples* t *such that* $t \in D^s(x_1) \times \cdots \times D^s(x_n)$ *and* $\tau^s_{min} \preceq_{lex} t \preceq_{lex} \tau^s_{max}$

Note that if we have $\tau^s_{max} \preceq_{lex} \tau^s_{min}$ then the tuple sequence is empty. For convenience, we will also denote a tuple sequence by $(\tau^s_{min}, \tau^s_{max}, g)$ where g is the GCS involved in the tuple sequence.

The main advantage of tuple sequences is their capability to represent in a positive way a set of forbidden tuples. Even if we could compress some domains, for instance a star ('*') could represent any value of the initial domain, we have to consider that the memory consumption of a tuple sequence is equal to the number of values it contains. The minimum is n and the maximum is in $O(nd)$. In this later case it is important to recall that an exponential number of tuples are expressed by only one tuple sequence.

3.2 Representation of Forbidden tuples

A set of forbidden tuples can be easily represented by a set of allowed tuple sequences. The idea is to represent it by the complementary of this set. Then, we will see that this complimentary set may be represented by a set of allowed tuple sequences.

We will use the following notation:

- the minimal value in $D(x)$ is denoted by $\underline{D}(x)$, and its maximal value is denoted by $\overline{D}(x)$.
- $\underline{\tau}(C)$ is the minimum combination of values on $X(C)$: $(\underline{D}(x_1), ..., \underline{D}(x_n))$
- $\overline{\tau}(C)$ is the maximum combination of values on $X(C)$: $(\overline{D}(x_1), ..., \overline{D}(x_n))$
- *n-comb*(t, C) denotes the combination of values succeeding immediately t on $X(C)$ w.r.t. the lexicographic ordering. It is equal to *nil* if t is the last combination.
- *p-comb*(t, C) denotes the combination of values preceding immediately t on $X(C)$ w.r.t. the lexicographic ordering. It is equal to *nil* if t is the first combination.

Consider a Table constraint C defined on $X(C) = \{x_1, ..., x_n\}$ whose the domain of x_i is $D(x_i)$, and $F = \{f_1, ...f_q\}$ a set of lexicographically ordered forbidden tuples on $X(C)$. Let P the Cartesian Product of the domains of the variable of $X(C)$, that is $D(x_1) \times \cdots \times D(x_n)$. Then the set of allowed tuples is equal to $A = P - F$.

Let g be the GCS formed by all the domains of the variables of $X(C)$. The set A can be represented by $|F| + 1$ allowed tuple sequences $\{as_1, as_2, ..., as_{|F|+1}\}$ defined as follows:

- the first tuple sequence is as_1 =($\underline{\tau}(C)$,*p-comb*(f_1, C),g).
- for $i = 2, ..., |F|$ the tuple sequence is as_i =(*n-comb*(f_{i-1}, C),*p-comb*(f_i, C),g)
- the last tuple sequence is $as_{|F|+1}$ =(*n-comb*$(f_n$,$\overline{\tau}(C)$,g).

Of course, the GCS part of these tuple sequences may be refined. Note also that some tuple sequences may be empty. This is the case when the minimum or the maximum tuple of the tuple sequence is *nil* or when the maximum tuple is smaller than the minimum tuple. In this case, it is not necessary to represent them.

Clearly, each tuple sequence may be represented in $O(nd)$. Thus, globally the memory consumption is in $O(nd|F|)$. This is more than the representation of $|F|$ forbidden tuples which is in $O(n|F|)$ (because a tuple contains n values), but this is less than Katsirelos and Walsh's method which needs $O(nd|F|)$ GCS tuples and therefore has a memory consumption in $O(n^2d^2|F|)$. Consequently we gain a factor of nd in regards to a pure GCS representation and we lose a factor of d in regards to the explicit representation of forbidden tuples. Therefore we have

Property 1 *A set F of forbidden tuples can be represented by at most $|F|+1$ allowed tuple sequences.*

Another advantage of our method in regards to Katsirelos and Walsh's one is its simplicity.

Combination with allowed tuple sequences In the previous section, we considered that all the tuples that not in the set F of forbidden tuples are allowed. In this section, we study the case where a set A of allowed tuples sequences is given in addition to the set F. Thus, only the tuple of A that are not in F are allowed.

First, we order lexicographically F, the forbidden tuple set. Then, we select each tuple sequence $as \in A$ in turn, and we search for the forbidden tuples that are contained in as. If we have k forbidden tuples that are included then we can split the tuple sequence as into $k+1$ tuple sequences with the method used in the previous section. Thus globally, we may obtained $|A| \times |F|$ tuple sequences. This value is certainly an upper bound of the tuple sequences obtained at the end. For instance, if the tuple sequences of A are disjoint then we will require only $O(|A|+|F|)$ sequences because a forbidden tuple can intersect only one tuple sequence.

Property 2 *A table constraint defined by a set A of allowed tuple sequences and a set F of forbidden tuples, can be represented only by a set of allowed tuple sequences having at most $|F||A|$ elements.*

Representation of Forbidden Tuple Sequences Two different cases must be considered depending on whether the tuple sequences are disjoint or not. If the forbidden tuple sequences are disjoint then a method similar to the one presented in the previous section may be used. Therefore, we obtain the same complexity. Unfortunately, this case is very specific and the general case, where we are provided with A a set of allowed tuple sequences and F a set of forbidden tuple sequences, requires an exponential number of allowed tuple sequences for representing both allowed and forbidden tuple sequences.

4 Integration into GAC-Schema

In this section, we show how tuple sequences may be integrated into GAC-Schema. First we show that a tuple sequence is a tuple cluster.

4.1 Minimum valid tuple

Searching for the minimum valid tuple[2] of a tuple sequence s can be efficiently performed by applying the following algorithm:

1. We start with the minimum tuple of s (i.e. τ^s_{min}). If τ^s_{min} is valid then we return it and we stop the algorithm. Otherwise, we search for the first index i such that the value of τ^s_{min} involving x_i is no longer in the domain of $D(x_i)$. That is we have $\forall j = 1..i-1$ $\tau^s_{min}[x_j] \in D(x_j)$ and $\tau^s_{min}[x_i] \notin D(x_i)$.
2. We search, from i to 1, for the first index j such that $D^s(x_j)$ contains a valid value greater than $\tau^s_{min}[x_j]$. If such index j does not exist then the algorithm returns *nil* and stops.
3. We build a new tuple t as follows: for $k = 1$ to $j-1$ we set $t[x_k] = \tau^s_{min}[x_k]$; $t[x_j]$ contains the first value of $D^s(x_j)$ which is valid and greater than $\tau^s_{min}[x_j]$ and for $k = j+1$ to n we set $t[x_k]$ to the minimum valid value of $D^s(x_k)$. If t is less than or equal to τ^s_{max} then the algorithm returns t else it returns *nil*.

For instance, suppose we have $D(x_1) = \{a, b, c\}$,$D(x_2) = \{a\}$,$D(x_3) = \{a, b\}$ and $D(x_4) = \{a, b, c\}$ and $s = (t^s_{min} = (a, b, c, c), t^s_{max} = (c, b, b, b),$ $(\{a, b, c\}, \{a, b, c\}, \{a, b, c\}, \{a, b, c\}))$. Then, the index computed by Step 1 is $i = 2$. From Step 2 we compute $j = 1$ and from Step 3 we obtain (b, a, a, a) which is the first valid tuple of s.

Clearly the algorithm is in $O(nd)$ because each step is in $O(nd)$ therefore a tuple sequence is a tuple cluster.

It is easy to search for the minimum valid tuple in s involving a given value (x, a). We just need to consider that $D(x) = \{a\}$ in the algorithm. Thus, we can easily search for a new support for a given value.

It is also interesting to note that this algorithm can be used in an incremental manner. In this case, we can prove that its time complexity can be amortized. It is straightforward to adapt this algorithm in order to search for the minimum valid tuple in s greater than a given tuple $\sigma \succeq \tau^s_{min}$: we just need to replace τ^s_{min} by σ in the algorithm. Thus, if for a value (x, a) we perform a sequence of searches, each of them starting from the result of the previous one and if this previous one is no longer valid then the cost of all these searches is in $O(nd)$. In addition it is clear that it does not cost more than traversing independently all the tuples involving (x, a) and contained in a tuple sequence s. This means that using tuple sequences cannot cost more than considering independently the tuples of the sequences.

[2] the minimum valid tuple in s is the tuple $t \in s$ such that t is valid and there is no valid tuple $t' \in s$ with $t' \prec_{lex} t$.

We can now detail the integration into GAC-Schema. We will consider only constraints defined by the list of allowed tuples.

4.2 Tuple sequences instead of tuples

Establishing arc consistency means maintaining a support for each value of each variable. When a value has no longer any support it can be safely deleted because it means that the value is not consistent with the constraint.

There are two processes in general purpose arc consistency algorithm: one for determining what are the values for which a support must be sought when a value has been deleted; and another one for searching for a new support for a value. The former process is usually done by associating with each value (x, a) a list of tuples containing it and that are currently a support of another value. Then, when the value (x, a) is deleted we know the values that have lost their support (because the tuple is no longer valid) by traversing this list of supported tuple. The latter process is the key of the algorithms and all the recent improvements are focused on the search for a new support for a given value.

Let us detail these two steps. Consider that a value (x, a) has been removed from $D(x)$: for all the values (y, b) that were supported by a tuple containing (x, a) another support must be found because the current one is no longer valid. These values are involved in a tuple of the list denoted by $S_C(x, a)$. Thus, GAC-Schema enumerates all the tuples in the S_C lists and considers the valid values supported by the tuple. Then, the algorithm tries to find a new valid support for these values.

The search for a valid support, is the main and the most difficult task of the GAC-Schema. For the sake of clarity we will not consider here the "multidirectional" aspect of GAC-Schema and assume that the search for a new support is mainly based on the traversal of the allowed tuples of the constraints. It considers successively the allowed tuples involving the value (y, b) for which a support is sought until a valid one is found. This can be done efficiently by linking together the allowed tuples involving the same value. That is, for each value (y, b) we define the list of allowed tuples involving it. It costs only one pointer (to the next tuple) for each value of each tuple. This is equivalent to the memory cost for representing all the tuples.

There is almost no change if we consider tuple clusters instead of tuples: we just need to associate the two notions. That is, when we consider a tuple we also need to know the tuple cluster it comes from. The set of tuples is now a set of tuple clusters. Each value in the tuple cluster is linked to the next tuple cluster containing it. This does not increase the memory consumption because it adds only one pointer for each value in the tuple cluster. Thus, while searching for a support for (y, b), instead of traversing the tuples containing (y, b) we traverse tuple clusters involving (y, b) by following the pointers associated with (y, b), until a valid tuple t belonging to the current tuple cluster is found. Such a valid tuple t is a support for (y, b). Since several tuples involving (y, b) may be valid in the tuple cluster we compute the minimum one by using the algorithm given in section 4.1. We give this information to GAC-Schema and also the tuple cluster k

from where it comes. That is, we return a pair (t,k). Then, GAC-Schema places t in S_C lists and uses t and k for computing another support.

A pair (t,k) contains two important information that will be useful to control the time complexity: k is the first tuple cluster containing a valid tuple involving (y,b) and t is the minimum valid tuple containing (y,b) in k. Thus, when t will no longer be valid we will be able to continue the search for a support from the point we stopped it. Therefore, within a tuple cluster we can avoid repeating the same computation. In fact, as we mentioned it in the previous section, if we start the search for a valid tuple in the tuple cluster from t the latest valid support, then we can amortize the time complexity. In addition, we can go from a tuple cluster to another tuple cluster as we did with tuple in the classical implementation of GAC-Schema. Therefore, considering tuple clusters instead of tuples may save a lot of memory and will not increase the worst case time complexity.

Traversing the list of allowed tuples or of tuple clusters containing (y,b) while searching for a support for (y,b) is a simple method but it has a major drawback: it does not consider the current domains of the variables for improving the search. The domain are only used to check the validity of tuples. In 2005, Lhomme and Régin [7] dramatically improved GAC-Schema by using the domain information while searching for a new support.

4.3 Integration into Lhomme and Régin's algorithm

In GAC-Schema the tuples are linked together: each value is associated with a pointer to the next tuple containing it. Lhomme and Régin shown that the algorithm can be speed up if for each tuple t and for each value (x,a) we know the first tuple following t which contains (x,a). It is important to note that we need this information for all values and not only for the values belonging to t. With this information huge parts of the tuple set can be avoided because they cannot contain any valid tuple. In fact, a support for a value (y,b) must contain (y,b) but it must also be valid, that is containing valid values for the other variables. So, the validity of values is used while searching for a support for (y,b), thanks to this chaining.

With tuples, there are two methods for implementing this chaining:

• for each tuple, we associate each value of each variable with a pointer representing the link to the next tuple containing the value. This methods is particularly efficient in practice. However it multiplies the memory consumption by a factor of d, which is a lot!

• a complex data structure is used: the hologram tuples. This data structure sacrifices a part of the time complexity in order to amortize the space complexity. We do not consume more memory but we need $O(d)$ to access to the next pointer for any value

We propose to apply the same ideas for tuple clusters. If we consider tuple clusters instead of tuples we do not really change the algorithm. We associate with each value of a tuple cluster a pointer to the next tuple cluster containing it. Thus, if the tuple cluster involves all values then we can reach the next tuple cluster for each value! If this is not the case, then we can add the missing information, either explicitly and we increase the memory consumption or by using the hologram data structure in which tuples are replaced by tuple clusters. Note that the memory increasing in the former case is less problematic with tuple clusters than with simple tuples, because we have already more information in tuple clusters since they can involve more than n values. Thus, we can consider that there is no change with tuple clusters in regards to the use of simple tuples.

5 Discussion

Tuple sequences could be generalized by considering for each tuple sequence a specific ordering for the variables and a specific ordering for the domain of each variable. This would not change the algorithm that would work in that case. We did not impose such orderings because it would have complicated the definitions and we did not find any real world example requiring such a modification.

The compression techniques like the one proposed by Katsirelos and Walsh could also be used in our case, because tuple sequences integrate GCS tuples. The drawback of this method is that it can be time consuming (See [5]). Nevertheless, it could be worthwhile to investigate the possibility to compress tuples into tuple sequences instead of GCS tuples.

We have also mentioned two points that deserve more attention. First, when the constraint is defined from the set of solutions of a problem (usually defined by a subset of constraints of the whole problem), we should try to compress the set of solutions while computing them. In that way, we could improve the time for computing all the solutions and obtain a more pertinent set of tuples. Secondly, a better integration of a set of forbidden combinations within a set of allowed combinations should be more considered. We have proposed a first method, but it is insufficient for representing some real world constraints.

6 Conclusion

In this paper we have introduced a new kind of tuple cluster: the tuple sequences. They generalize the GCS tuples used in Table constraints by Katsirelos and Walsh. The representation of forbidden tuples by allowed tuple sequences is simple and requires less memory than their representation by GCS tuples. Therefore, there is currently almost no more any reason to have a dedicated version of GAC-Schema dealing only with forbidden tuples. In addition, we have shown that tuple sequences may be used for mixing allowed and forbidden tuples in a simple way. At last, we have explained how tuple clusters may be easily integrated into the best implementations of GAC-Schema like the one of Lhomme and Régin.

References

1. C. Bessière and J-C. Régin. Arc consistency for general constraint networks: preliminary results. In *Proceedings of IJCAI'97*, pages 398–404, Nagoya, 1997.
2. K. Cheng and R. Yap. An mdd-based generalized arc consistency algorithm for positive and negative table constraints and some global constraints. *Constraints*, 15, 2010.
3. F. Focacci and M. Milano. Global cut framework for removing symmetries. In *Proc. CP'01*, pages 77–92, Paphos, Cyprus, 2001.
4. I. Gent, C. Jefferson, I. Miguel, and P. Nightingale. Data structures for generalised arc consistency for extensional constraints. In *Proc. AAAI'07*, pages 191–197, Vancouver, Canada, 2007.
5. G. Katsirelos and T. Walsh. A compression algorithm for large arity extensional constraints. In *Proc. CP'07*, pages 379–393, Providence, USA, 2007.
6. C. Lecoutre. Optimization of simple tabular reduction for table constraints. In *Proc. CP'08*, pages 128–143, Sydney, Australia, 2008.
7. O. Lhomme and J-C. Régin. A fast arc consistency algorithm for n-ary constraints. In *Proc. AAAI'05*, pages 405–410, Pittsburgh, USA, 2005.
8. Justin Yip and Pascal Van Hentenryck. Exponential propagation for set variables. In *Proc. CP-10*, pages 499–513, 2010.