A Fast Arc Consistency Algorithm for n-ary Constraints

Olivier Lhomme[1] and **Jean-Charles Régin**[2]
[1]ILOG, 1681, route des Dolines, 06560 Valbonne, FRANCE
[2]Computing and Information Science, Cornell University, Ithaca NY 14850 USA
olhomme@ilog.fr, jcregin@cs.cornell.edu

Abstract

The GAC-Scheme has become a popular general purpose algorithm for solving n-ary constraints, although it may scan an exponential number of supporting tuples. In this paper, we develop a major improvement of this scheme. When searching for a support, our new algorithm is able to skip over a number of tuples exponential in the arity of the constraint by exploiting knowledge about the current domains of the variables. We demonstrate the effectiveness of the method for large table constraints.

Introduction

Constraint satisfaction problems (CSPs) form a simple formal frame to represent and solve combinatorial problems in artificial intelligence. They involve finding values for problem variables subject to constraints on which combinations are acceptable. The problem of the existence of solutions to the CSP is NP-complete. Therefore, methods have been developed to simplify the CSP before or during the search for solutions. A filtering algorithm associated with one constraint aims to remove some values that are not consistent with the constraint. When all the values that are inconsistent with the constraint are deleted by the filtering algorithm we say that it achieves arc consistency.

A constraint explicitly defined by the table of the allowed combinations of values is among the most useful kinds of constraints in practice. Such table constraints are typically used to express compatibilities between persons, tasks or machines. Binary table constraints have been extensively studied in the CSP framework, and efficient filtering algorithms have been proposed (Bessière 1994; Bessière, Freuder, & Régin 1999; Bessière & Régin 2001; Zhang & Yap 2001).

Unfortunately, in practice, table constraints often are non-binary. Indeed, in some applications, data comes from databases, and relations from databases are directly translated as table constraints. This is frequently the case for configuration applications, for modeling the different combinations of options. In such cases, table constraints may have quite large arity.

The generalization of the results and algorithms from binary constraints to n-ary ones is not an easy task in general. For example, the best known algorithm for n-ary table constraints, called GAC-Scheme (Bessière & Régin 1997), is such a generalization, and it involves several complex subtleties. Nevertheless, it is a too direct generalization of binary arc-consistency algorithms; we will see that the GAC-Scheme does not take into account the fact that the constraints are n-ary, and may have a behavior which is exponential in the arity of the constraint.

A similar exponential behavior may occur when one wants to design an algorithm for the conjunction of constraints as in (Lhomme 2004), where an efficient algorithm was proposed to avoid this behavior by exploiting the current domains of the variables. Furthermore, it was suggested that the same approach should be advantageous for arc consistency filtering over table constraints.

In this paper, we apply this idea on the GAC-Scheme, which leads us to a new filtering algorithm for n-ary table constraints that really takes into account the fact that their arity is more than two. It combines in a new way the two major concepts of any arc consistency filtering algorithm: the concept of *support* and the concept of *validity*. A support for the value a of the variable x is a tuple which belongs to the table and with value a for x. A support is said to be valid if all the values it contains belong to their respective domains.

The goal of arc consistency filtering algorithms is to search for a valid support for every value of every domain. Consider the value a of the variable x, denoted by (x, a). A valid support for (x, a) can be found by considering successively the supports of (x, a) of the constraint until a valid one is found. When constraints are binary, this is a good strategy. Nevertheless, on n-ary constraints, this approach is not a good one.

Consider the following example: a constraint C is defined on six variables $x_1, x_2, \ldots, x_6$ whose domains are the range of integers from 0 to 4, defined by the list of tuples given in Figure 1.

Then, suppose that arc consistency has been established. In particular, $(0, 0, 0, 0, 0, 0)$ is the valid support found for $(x_1, 0)$. Now assume that the value $(x_6, 0)$ is deleted. Thus $(0, 0, 0, 0, 0, 0)$ is no longer valid, and a new valid support has to be found for the value $(x_1, 0)$. The GAC-Scheme

$(0,0,0,0,0,0)$
$(0,0,0,0,1,0)$
$(0,0,0,0,2,0)$
$(0,0,0,0,3,0)$
$(0,0,0,0,4,0)$
$(0,0,0,1,0,0)$
$(0,0,0,1,1,0)$
...
$(0,4,4,4,4,0)$
$(1,1,1,1,1,1)$
$(2,2,2,2,2,2)$
$(3,3,3,3,3,3)$
$(4,4,4,4,4,4)$

Figure 1: Table of the supports of Constraint C

searches for this new valid support by traversing all the supports involving $(x_1, 0)$ until a valid one is found. But all the supports for $(x_1, 0)$ have the value 0 for x_6. Thus, in this case, the GAC-Scheme considers successively all the tuples under the form $(0, *, *, *, *, 0)$, that is 5^4 supports, and checks the validity of all these supports. Since none is valid, the value 0 will be removed from $D(x_1)$. This example perfectly shows that the GAC-Scheme is mainly focused on the concept of support. In this paper, we propose another method whose principles are quite simple. This method assumes that the tuples are ordered and that we can know for any support t and for every value the next support after t containing this value. More precisely, if $t = (0,0,0,0,0,0)$ is the current support for $(x_1, 0)$, then we can know the next support after t which is a support for $(x_6, 1)$ (i.e. $(1,1,1,1,1,1)$.) Then, we can focus our attention on the current domains and deduce some properties based on these current domains.

To be valid, a support for $(x_1, 0)$ must have for x_6 one of the values in the current domain of x_6. The next support after t which is a support for $(x_6, 1)$ is $(1,1,1,1,1,1)$; the next support after t which is a support for $(x_6, 2)$ is $(2,2,2,2,2,2)$; the next support after t which is a support for $(x_6, 3)$ is $(3,3,3,3,3,3)$, The next support after t which is a support for $(x_6, 4)$ is $(4,4,4,4,4,4)$. Thus, a valid support for $(x_1, 0)$ is at least the smallest of the above four supports, i.e., $(1,1,1,1,1,1)$. This means that it is useless to check the validity of a support smaller than $(1,1,1,1,1,1)$. Obviously $(1,1,1,1,1,1)$ is not a support for $(x_1, 0)$, so we must take a support which is greater since $(1,1,1,1,1,1)$ was the smallest possible one. So we look for the next support for $(x_1, 0)$ after $(1,1,1,1,1,1)$. There is none and thus the value $(x_1, 0)$ can be removed.

So, on this example, compared with GAC-Scheme, we have avoided a number of validity checks exponential in the arity of the constraint. The paper develops the idea presented in the above example, and proposes to revise the GAC-Scheme. We will see with the experiments that this method allows us to reduce computation time from one to several orders of magnitude, even on random problems.

Background

Preliminary definitions

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. $\overline{D}(x_i)$ denotes the highest value in $D(x_i)$. 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.

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.

The GAC-Scheme

Algorithm 1: function GENERALFILTER

```
GENERALFILTER(C: constraint; x: variable; a: value, deletionSet: list): Boolean
1  for each τ ∈ S_C(x, a) do
       for each (z, c) ∈ τ do remove τ from S_C(z, c)
2      for each (y, b) ∈ S(τ) do
           remove (y, b) from S(τ)
           if b ∈ D(y) then
3              σ ← SEEKINFERABLEVALIDSUPPORT(y, b)
               if σ ≠ nil then add (y, b) to S(σ)
               else
4                  σ ← SEEKVALIDSUPPORT(C, y, b)
                   if σ ≠ nil then
                       add (y, b) to S(σ)
                       for each x ∈ X(C) do
                           add σ to S_C(x, σ[x])
                   else
                       remove b from D(y)
                       if D(y) = ∅ then return False
                       add (y, b) to deletionSet
   return True
```

Let us review the GAC-Scheme presented in (Bessière & Régin 1997). Function GENERALFILTER of the GAC-Scheme is given in Algorithm 1.

Consider that a value (x, a) has been removed from $D(x)$. We must study the consequences of this deletion. So, for all the values (y, b) that were supported by a tuple containing (x, a) another valid support must be found. In order to perform these operations, the GAC-Scheme uses two lists:

• $S_C(x, a)$ contains all the allowed tuples τ that are the current support for some value, and such that $\tau[x] = a$.

• $S(\tau)$ contains all values for which τ is the current support.

Thus, Line 1 of Algorithm 1 enumerates all the tuples in the S_C list and Line 2 enumerates all the values supported by a tuple. Then, the algorithm tries to find a new valid support for these values either by "inferring" new ones (Line 3) or by explicitly calling function SEEKVALIDSUPPORT (Line 4).

Here is an example of this algorithm:
Consider $X = \{x_1, x_2, x_3\}$ and $\forall x \in X, D(x) = \{a, b\}$; and $T(C) = \{(a, a, a), (a, b, b), (b, b, a), (b, b, b)\}$. First, a valid support for (x_1, a) is sought: (a, a, a) is computed and (a, a, a) is added to $S_C(x_2, a)$ and $S_C(x_3, a)$, (x_1, a) in (a, a, a) is added to $S((a, a, a))$. Second, a valid support for (x_2, a) is sought: (a, a, a) is in $S_C(x_2, a)$ and it is valid, therefore it is a valid support and so there is no need to compute another solution. Then the algorithm searches for a valid support for all the other values.

Now, suppose that value a is removed from x_2. Then all the tuples in $S_C(x_2, a)$ are no longer valid, such as, for example, (a, a, a). The validity of the values supported by this tuple must be reconsidered; that is, the ones belonging to $S((a, a, a))$. Therefore, a new valid support for (x_1, a) must be searched for and so on...

The program which aims to establish arc consistency for C must create and initialize the data structures (S_C, S), and call function GENERALFILTER$(C, x, a, deletionSet)$ (see Algorithm 1) each time a value a is removed from a variable x involved in C, in order to propagate the consequences of this deletion. The set $deletionSet$ is updated to contain the deleted values not yet propagated.

The search for a valid support, which is the main and most difficult task of the GAC-Scheme, is performed by the functions SEEKINFERABLEVALIDSUPPORT and SEEKVALIDSUPPORT of GENERALFILTER. These functions try to take into account the *multidirectionality* inherent in any constraint when establishing arc consistency. Roughly, the advantage of such an approach is to avoid checking several times whether a tuple is a support for the constraint or not.

Definition 1 (multidirectionality) *Let C be a constraint. If σ is a support for a value (x, a) on C, then it is a support on C for all the values $(y, \sigma[y])_{y \in X(C)}$ composing it.*

Function SEEKINFERABLEVALIDSUPPORT "infers" an already checked allowed tuple as support for (y, b) if possible, in order to ensure that it never looks for a support for a value when a tuple supporting this value has already been checked. The idea is to exploit the property: "If (y, b) belongs to a tuple supporting another value, then this tuple also supports (y, b)". Therefore, elements in $S_C(y, b)$ are good candidates to be a new valid support for (y, b). Algorithm 2 is a possible implementation of this function.

Function SEEKVALIDSUPPORT is instantiated differently depending on the type of the constraint involved. (Bessière & Régin 1997) provided three instantiations of the scheme to efficiently handle constraints defined by a list of allowed tuples (GAC-Scheme+allowed), by a list of forbidden tuples (GAC-Scheme+forbidden), and by any predicate (GAC-Scheme+predicate).

Algorithm 2: Function SEEKINFERABLEVALIDSUPPORT

```
SEEKINFERABLEVALIDSUPPORT(y: variable, b: value): tuple
while S_C(y, b) ≠ ∅ do
    σ ← first(S_C(y, b))
    if σ is valid then return σ    /* σ is a valid support */
    else remove σ from S_C(y, b)
return nil
```

However, all these instantiations used the concept of last support to take into account multidirectionality. For every value b of every variable y of $X(C)$, $last(y, b)$ is the last tuple returned by SEEKVALIDSUPPORT as a support for (y, b) if SEEKVALIDSUPPORT(C, y, b) has already been called; nil otherwise. There is an ordering on the tuples, which is local to SEEKVALIDSUPPORT, and $last(y, b)$ gives the point where SEEKVALIDSUPPORT will have to restart the search for a support for (y, b) on C at the next call. Thus, we can avoid considering tuples that have been previously studied for (y, b). In addition, (Bessière & Régin 1999) have proposed to exploit the presence of the last support for every value in order to avoid checking whether some tuples are a support for (y, b) when they have already been unsuccessfully checked for *another value*.

In the next section, we propose simpler and stronger properties by introducing lower and upper bounds of valid tuples.

Lower bounds on the next valid support

Let C be a constraint defined on the set X of variables. For the sake of clarity and without loss of generality, we will consider that the tuples on X are lexicographically ordered and we will denote by $\prec$ this order , and by $\top$ a tuple greater than any other tuple.

The goal of function SEEKVALIDSUPPORT is to find a valid support as quickly as possible. For instance, the instantiation of this function for GAC-Scheme+allowed traverses the elements of $T(C)$ until a valid one is found. We propose to accelerate this traversal by exploiting lower bounds on valid supports.

Definition 2 *Given $x \in X$ and $a \in D(x)$:*

• *$T(C, x, a)$ is the set of tuples of $T(C)$ in which x is assigned to a (i.e. the set of support for (x, a)).*

• *a tuple τ on X is a* **lower bound of the smallest valid tuple (*$lbsvt$*) w.r.t.** *(x, a)* **and** *C iff $\forall \sigma \in T(C, x, a)$: $\sigma \prec \tau \Rightarrow \sigma$ is not valid.*

• *a tuple τ on X is an* **upper bound of the greatest valid tuple (*$ubgvt$*) w.r.t.** *(x, a)* **and** *C iff $\forall \sigma \in T(C, x, a)$: $\sigma \succ \tau \Rightarrow \sigma$ is not valid.*

We will denote by:

• *lb any function which associates with every value a of every variable x a lbsvt w.r.t. (x, a) and C*

• *ub any function which associates with every value a of every variable x an ubgvt w.r.t. (x, a) and C*

We also define $\forall x \in X$: $minlb(x) = \min_{a \in D(x)}(lb(x, a))$

The function $last$ is an example of function lb. The function[1] which associates with every value (x, a) the highest possible tuple of the Cartesian Product of the current domains assigning a to x is an example of function ub.

We immediately have:

Property 1 *Given $x \in X$ and $a \in D(x)$.*
If $lb(x, a) \succ ub(x, a)$ then (x, a) is not consistent with C.

The minimum of the lower bound associated with the values of the domain of a given variable is a general lower bound:

Property 2 *$\forall y \in X, \forall x \in X$ and $\forall a \in D(x)$:*
$minlb(y)$ is an lbsvt w.r.t. (x, a) and C.

proof: The smallest valid tuple of $T(C, x, a)$ contains one value b of $D(y)$. This tuple is valid, so it is greater than or equal to the lbsvt w.r.t. (y, b) and C. Thus, it is also greater than or equal to $minlb(y)$. Thus, $minlb(y)$ is an lbsvt w.r.t. (x, a) and C $\odot$

It is important to note that this property deals only with the values of $D(y)$.

From this property, we can compute a lower bound associated with every variable. Thus, we can take the maximum value of these lower bounds in order to compute a new lower bound:

Corollary 1 *Given $x \in X$, $a \in D(x)$, and*
$\sigma = \max[lb(x, a), \max_{y \in (X - \{x\})}(minlb(y))]$:
σ is an lbsvt w.r.t. (x, a) and C.

Consider the example given in the Introduction section. We define $lb = last$. After establishing arc consistency we have $last(x_6, 1) = (1, 1, 1, 1, 1, 1)$, $last(x_6, 2) = (2, 2, 2, 2, 2, 2)$, $last(x_6, 3) = (3, 3, 3, 3, 3, 3)$ and $last(x_6, 4) = (4, 4, 4, 4, 4, 4)$. After the deletion of $(x_6, 0)$ we have $minlb(x_6) = \min((1, 1, 1, 1, 1, 1), (2, 2, 2, 2, 2, 2), (3, 3, 3, 3, 3, 3), (4, 4, 4, 4, 4, 4)) = (1, 1, 1, 1, 1, 1)$. Corollary 1 applied for x_1 gives $\sigma = \max_{y \in (X - \{x_1\})}(minlb(y)) \succeq minlb(x_6) = (1, 1, 1, 1, 1, 1)$ therefore $ub(x_1, 0) \prec \sigma$ because $0 < \sigma[x_1]$ and so there is no valid support for $(x_1, 0)$.

Improvement versus related work

With our notations, we can simplify the result of (Bessière & Régin 1999) on which their filtering algorithms are based. Property 1 generalizes their first result and their second result can be rewritten (and generalized) as:

Property 3 *Let $x \in X$ with $D(x) = \{a\}$ and $y \neq x$ be any variable of X and b be any value of $D(y)$.*
If $\forall \sigma \in T(C, y, b)$: $\sigma \prec lb(x, a)$ then (y, b) is not consistent with C.

This property leads to fewer deductions than the new corollary we have proposed. Consider the previous example; it is not possible to deduce the same things as Corollary 1 by using Property 3, because after the deletion of $(x_6, 0)$ there is no instantiated variable and if we search for a new valid support for $(x_1, 0)$ then we have $lb(x_1, 0) = last(x_1, 0) = (0, 0, 0, 0, 0, 0)$ which precedes all the other tuples according to the lexicographic order. Thus, an explicit search for a new valid support will be required in order to prove that there is none. Hence, Corollary 1 clearly outperforms the previous results.

[1] More precisely $ub(x, a)$ is defined as follows:
$\forall y \in (X - \{x\})$: $ub(x, a)[y] = \overline{D}(y)$; and $ub(x, a)[x] = a$.

Implementation

There is no particular difficulty to implement Corollary 1. Algorithm 3 gives a synopsis of function SEEKVALIDSUPPORT.

Algorithm 3: Synopsis of Function SEEKVALIDSUPPORT

```
SEEKVALIDSUPPORT(C: constraint, x: variable, a: value): tuple
1 σ ← max[lb(x, a), max_{y∈(X(C)−{x})}(minlb(y))]
  if σ ≻ ub(x, a) then return nil
  else
      search for a new valid support for (x, a) from σ
      return this support if exists and nil otherwise
```

For every variable y, we can maintain $minlb(y)$ by recomputing the new minimum each time $D(y)$ is modified. If the two variables having the largest $minlb(y)$ are maintained then we can know the value of $\max_{y \in (X - \{x\})}(minlb(y))$ for every variable x. There is no need to maintain the values $ub(x, a)$ for every (x, a) because this value can be computed while performing the test $\sigma \prec ub(x, a)$.

It is also possible to implement the multidirectionality by a constraint as proposed by (Bessière & Régin 1999). From Corollary 1 we deduce the following corollary which leads to a filtering algorithm:

Corollary 2 *Given $x \in X$ and $\sigma = \max_{y \in (X - \{x\})}(minlb(y))$.*
$\forall a \in D(x)$: $ub(x, a) \prec \sigma \Rightarrow (x, a)$ is not consistent with C.

Combination of two major concepts: support and validity

The instantiation of the GAC-Scheme (named GAC-Scheme+allowed) dealing with a constraint explicitly defined by the list of its allowed tuples considers that for every value (x, a) the list of tuples involving this value is explicitly given, and that it is possible from a tuple to access the tuple which follows it in that list thanks to function NEXT$((x, a), \sigma)$. This function returns, if it exists, the tuple following σ in the list of tuples of $T(C, x, a)$; otherwise it returns $\top$. Then, function SEEKVALIDSUPPORT traverses the values of $T(C, x, a)$ until a valid tuple is found. Algorithm 4 gives the specific GAC-Scheme+allowed instantiation. The definition of arc consistency involves two main concepts: the concept of support (i.e. a tuple allowed by the constraint) and the concept of validity (i.e. a tuple whose values belong to their respective domains.)

In this section, we show for the first time how these two concepts can be combined and not only independently considered as in previous studies. In other words, we improve Algorithm 4 by integrating the ideas of Corollary 1, that is

Algorithm 4: Function SEEKVALIDSUPPORT of GAC-Scheme+allowed

SEEKVALIDSUPPORT(C: constraint, x: variable, a: value): tuple
1 $\sigma \leftarrow last(x,a)$
if $\sigma \succ ub(x,a)$ **then** return nil
while $\neg$ ISVALID(σ) **do**
2 $\quad \sigma \leftarrow$ NEXT($(x,a),\sigma$)
$\quad$ **if** $\sigma \succ ub(x,a)$ **then** return nil
$last(x,a) \leftarrow \sigma$
return σ

by using the lower bounds obtained from all the current domains of the variables. That is, we try to extract some information from the values that currently belong to the domains of the variables in order to avoid considering tuples containing values that have been deleted.

First, we show how Line 1 is modified in order to take into account Corollary 1.

Function NEXT of GAC-Scheme+allowed makes a strong assumption on the tuples that it considers. For a value (x,a) this function takes a tuple of $T(C,x,a)$ as parameter and returns a tuple of $T(C,x,a)$. Unfortunately, the lower bound of Corollary 1 does not necessary involve (x,a). Therefore, we need a function which is able to compute a tuple of $T(C,x,a)$ from a tuple which does not assign a to x. We will denote by NEXTIN such a function:

Definition 3 $\forall x \in X, \forall a \in D(x), \forall \sigma \in T(C)$:
NEXTIN$((x,a),\sigma) = \sigma$ *if* $\sigma[x] = a$
NEXTIN$((x,a),\sigma) = \tau$ *if* $\sigma[x] \neq a$ *and* $\exists \tau \in T(C,x,a)$ *with* $\tau \succ \sigma$ *and* $\nexists t \in T(C,x,a)$ *s.t.* $\sigma \prec t \prec \tau$.
NEXTIN$((x,a),\sigma) = \top$ *if* $\sigma[x] \neq a$ *and the previous condition is not satisfied.*

Now, we can replace Line 1 of Algorithm 4 by the code consisting of the call of function NEXTIN with (x,a) and the σ value of Line 1 of Algorithm 3 as parameters, that is:
$\sigma \leftarrow$ NEXTIN$((x,a), \max[lb(x,a), \max_{y \in (X-\{x\})}(minlb(y))])$

We can further reduce the number of computations made by Algorithm 4, by using function NEXTIN in the loop.

First, we have a property similar to Property 2:

Property 4 *Let σ be a lbsvt w.r.t. (x,a) and C, and $y \neq x$ be a variable of X. Then*
$$n(y,\sigma) = \min_{b \in D(y)} [\text{NEXTIN}((y,b), \max(lb(y,b),\sigma))]$$
is an lbsvt w.r.t. (x,a) and C.

proof : The smallest valid tuple svt containing (x,a) will also contain one value b of $D(y)$. By definition svt is greater than any lbsvt w.r.t. (x,a), so we have $svt \succ \sigma$, and svt is greater than any lbsvt w.r.t. (y,b) and C because svt is valid and $svt[y] = b$. Therefore, we have $svt \succ \max(lb(y,b),\sigma)$. By definition of function NEXTIN the property holds. $\odot$

The advantage of this property is that only the values that are currently in the domain of any variable y are considered. So we increase the chance that we will avoid considering tuples that are not valid. This is a way to combine the concept of support and the concept of validity. For the first time, the domains of the variables are explicitly considered before testing the validity of a tuple. These domains are used to compute the next tuple whose validity will be checked.

Then, we obtain a corollary similar to Corollary 1:

Corollary 3 *Let σ be a lbsvt w.r.t. (x,a) and C, and $y \neq x$ be a variable of X. Then*
$$\text{NEXTIN}((x,a), \max[lb(x,a), \max_{y \in (X-\{x\})}(n(y,\sigma))])$$
is an lbvst w.r.t. (x,a) and C.

Algorithm 5: New Function SEEKVALIDSUPPORT

SEEKVALIDSUPPORT(C: constraint, x: variable, a: value): tuple
1 $\sigma \leftarrow$ NEXTIN$((x,a), \max[lb(x,a), \max_{y \in (X(C)-\{x\})}(minlb(y))])$
if $\sigma \succ ub(x,a)$ **then** return nil
while $\neg$ ISVALID(σ) **do**
2 $\quad \sigma \leftarrow$ NEXT($(x,a),\sigma$)
3 $\quad$ **for each** $y \in (X(C) - \{x\})$ **do**
$\quad\quad n(y,\sigma) \leftarrow \min_{b \in D(y)}$ [NEXTIN$((y,b), \max(lb(y,b),\sigma))$]
4 $\quad t \leftarrow$ NEXTIN$[(x,a), \max_{y \in (X(C)-\{x\})}(n(y,\sigma))]$
5 $\quad$ **if** $t \succ \sigma$ **then** $\sigma \leftarrow t$
$\quad$ **if** $\sigma \succ ub(x,a)$ **then** return nil
$lb(x,a) \leftarrow \sigma$
return σ

Algorithm 5 gives a possible design of a new function SEEKVALIDSUPPORT based on this corollary. This algorithm uses a function lb which associates with every value (x,a) a lbsvt w.r.t. (x,a) and C. Of course, this function could be replaced by function $last$.

Note that if σ is valid after Line 2 then $\forall y \in (X - \{x\})$: $\sigma[y] \in D(y)$ and $n(y,\sigma) \preceq \sigma$ so $t \preceq \sigma$.

Implementation of Function NEXTIN. This function can be easily implemented if we accept increasing the space complexity of the GAC-Scheme.

First, as with function NEXT, we define for each value (x,a) the ordered list $T(C,x,a)$. That is, for each tuple $\sigma \in T(C,x,a)$, and for each value (x,a) of σ we add a pointer to the next tuple of $T(C,x,a)$ according to the ordering.

Then, for each tuple $\sigma \in T(C)$ and for each value (x,a) of this tuple we add $|D(x)|$ pointers. Each pointer is associated with a value of $D(x)$; thus we will denote it by $p(\sigma,x,b)$. Each pointer $p(\sigma,x,b)$ points to a tuple $\tau \in T(C,x,b)$ such that $\tau \succ \sigma$, and there is no $t \in T(C,x,b)$ with $\sigma \prec t \prec \tau$. With this mechanism of pointer we simply have: if $\sigma[x] = a$ then NEXTIN$((x,a),\sigma) = \sigma$ else NEXTIN$((x,a),\sigma) = p(\sigma,x,a)$. If d is the size of the largest domain then the addition of pointers for every tuple multiplies the space complexity of the GAC-Scheme by a factor of d.

It is possible to implement function NEXTIN without increasing the space complexity of the GAC-Scheme, by using a specific data structure and a more complex procedure. For more information the reader can consult (Lhomme 2004).

In addition, comparisons between tuples can be performed with a time complexity independent of the size of

tuples provided that the list $T(C)$ is given. In this case, we can give a number to each tuple in $T(C)$ which corresponds to its rank according to the ordering on the tuple. Then, the comparison between these two rank numbers is sufficient to compare the two tuples.

Experiments

All the experiments have been made on a Pentium III, 400Mhz, with ILOG Solver 6.1 (ILOG 2005) which implements our method for n-ary constraints explicitly defined by the list of allowed tuples. The gain in efficiency our algorithm may bring is directly related to the number of tuples. If the table contains only a few tuples, there is no efficiency problems: GAC is very fast and there is nothing to gain. Thus, in our experiments, we consider tables with a big number of tuples.

There are two distinct cases to consider: structured sets of tuples and random sets of tuples. As expected, our method leads to a huge improvement in the former case. We also show that, even for problems with no structure at all, i.e. on random problems, our method is worthwhile.

Structured sets of tuples

In practice, tables represent relations between variables from real data, and, in applications, it is almost always the case that the set of tuples is not random. Some structure is thus hidden in the set of tuples. In such cases, our method may dramatically reduce computation time: the gain can be linear in terms of the number of tuples in the table, i.e., exponential in the arity of the table, as for the example given in the Introduction section. If we consider the very same kind of example but with 8 variables and 10 values per domain, then our method requires less than 10 ms to establish again arc consistency after the deletion of $(x_8, 0)$ whereas the previous GAC-Scheme+allowed requires 5648 ms.

Of course, the structure in the table may be not as obvious as in the above example but our improvement typically leads to a much more robust behavior w.r.t. the tuple ordering and the search strategy.

Random sets of tuples

First we consider a test that is an adaptation of a local search benchmark on Boolean variables. The problems of the first test involve 24 variables. All the constraints have the same arity (14). Each constraint is defined over randomly chosen variables. The Cartesian product size is 2^{14}. We take randomly 2^{13} tuples in each table. Thus, the density of each constraint is around 0.5. The advantage of this model is that, by varying only one parameter, the number of constraints, we vary the difficulty in solving the problem. The cpu time for finding the first solution is reported in the following table.

#ct	#bk	new time	old time	gain
8	184	1.9	26.1	13.7
10	382	5.0	59.8	12.0
12	421	6.3	73.6	11.7
14	305	5.2	59.3	11.4
16	199	3.8	41.5	10.9

- #ct is the number of constraints
- #bk the number of choice points
- "old" corresponds to the GAC-Scheme+allowed algorithm of (Bessière & Régin 1997)
- "new" represents our method
- "gain" is the cpu time ratio

Times are expressed in seconds.

Our method clearly improves the previous one by an order of magnitude.

Then, we test sparse constraints with larger arity (20), on Boolean variables again. The 20 variables of each constraint are randomly chosen among 40, and there are 30,000 tuples per constraint.

#ct	#bk	new time	old time	gain
1	34	0.02	0.56	28
2	30	0.04	1.94	48
3	23	0.09	1.96	22
4	53	0.64	26.28	41
5	1122	21.64	> 300	> 10

The gains are even better than in the previous test.

Now, we test variables with larger domain size (10). The constraints are of arity 6, and share the same set of tuples which contains 100,000 tuples (density 0.1).

#ct	#bk	new time	old time	gain
1	10	0.05	0.4	8
2	8	0.01	0.6	60
3	8	0.04	1.0	25
4	59	0.3	13.5	45
5	14	0.2	7.9	40
6	155	2.9	188	65
7	133	3.3	207	63
8	313	6.3	407	65
9	2439	64	$> 3,600$	> 56

Once again, our method clearly outperforms the previous studies by almost two orders of magnitude.

Conclusion

In this paper, we have proposed a new algorithm for establishing arc-consistency on n-ary constraints given in extension by their allowed combinations of values. The idea is to exploit the current domains of the variables to boost the search for a new valid support. Our method outperforms the best known arc-consistency algorithm.

Acknowledgments

We thank Ulrich Junker for his constructive comments.

References

Bessière, C., and Régin, J.-C. 1997. Arc consistency for general constraint networks: preliminary results. In *Proceedings of IJCAI'97*, 398–404.

Bessière, C., and Régin, J.-C. 1999. Enforcing arc consistency on global constraints by solving subproblems on the fly. In *Proceedings of CP'99*, 103–117.

Bessière, C., and Régin, J.-C. 2001. Refining the basic constraint propagation algorithm. In *Proceedings of IJCAI'01*, 309–315.

Bessière, C.; Freuder, E.; and Régin, J.-C. 1999. Using constraint metaknowledge to reduce arc consistency computation. *Artificial Intelligence* 107(1):125–148.

Bessière, C. 1994. Arc-consistency and arc-consistency again. *Artificial Intelligence* 65(1):179–190.

ILOG. 2005. *ILOG Solver 6.1 User's manual*. ILOG S.A.

Lhomme, O. 2004. Arc-consistency filtering algorithms for logical combinations of constraints. In *Proc. CP-AI-OR'04*.

Zhang, Y., and Yap, R. 2001. Making ac-3 an optimal algorithm. In *Proceedings of IJCAI'01*, 316–321.