Enforcing arc consistency on global constraints by solving subproblems on the fly*

Christian Bessière[1] and Jean-Charles Régin[2]

[1] LIRMM-CNRS
161, rue Ada
34292 Montpellier, France
e-mail: bessiere@lirmm.fr
[2] ILOG
1681, route des Dolines
06560 Valbonne, France
e-mail: regin@ilog.fr

Abstract. Constraint networks are used more and more to solve combinatorial problems in real-life applications. As pointed out in [1], this success requires dealing with non-binary constraints, which are widely needed in real world constraint solvers. Since arc consistency is a fundamental piece of reasoning that seems to be of great help during search for solutions, it is important to have efficient arc consistency algorithms for non-binary constraints. In this paper, we propose a new instantiation of GAC-Schema [1] that achieves arc consistency on global constraints (sets of original constraints taken globally). This algorithm, which searches for supports on these global constraints by solving the corresponding subproblem on the fly, outperforms the previous available techniques, such as GAC-Schema and its instantiation to predicates. It can simultaneously take into account the multidirectionality of the global constraint, and take advantage of the knowledge of the constraints of the subproblem. Furthermore, we show on several experiments the interest of our approach for problem modelling.

1 Introduction

In constraint programming, to solve a problem, we begin by designing a model of it using predefined constraints, such as sum, alldiff. Next, we define other constraints specific to the problem. Then we call a procedure to search for a solution.

Often when we are solving a real problem, say problem $\mathcal{P}$, the various simple models that we come up with cannot be solved within a reasonable period of time. In such a case, we may consider a subproblem of the original problem, say subproblem $\mathcal{R}$. We then try to improve the resolution of $\mathcal{R}$ with the hope of thus eventually solving $\mathcal{P}$.

* This work has been partially financed by ILOG S.A. under research contract ILOG/LIRMM no. 980330.

More precisely, for each possible relevant subproblem of $\mathcal{P}$, we construct a *global* constraint that is the conjunction of the constraints involved in the subproblem. We then apply arc consistency to these new constraints. Suppose that this improves the resolution of $\mathcal{P}$ (i.e., the number of backtracks markedly decreases). In this case, we know that it is worthwhile to write another algorithm dedicated to solving the subproblem $\mathcal{R}$ under consideration. In contrast, if the number of backtracks decreases only slightly, then we know that the resolution of $\mathcal{R}$ has only modest effect on the resolution of $\mathcal{P}$. By proceeding in this way, we can improve the resolution of $\mathcal{P}$ much faster.

Let us consider a simple example. This is a problem used in cryptography. It consists of a square with r rows and r columns, each row i (resp. each column j) being associated with an integer $rsum(i)$ (resp. $csum(j)$). The goal is to find a key, i.e., to put the integers 1 to r^2 in the cells of the square in such a way that the sum of the values affected to row i is equal to $rsum(i)$, and the sum of the values affected to column j is equal to $csum(j)$, $\forall i, j \in [1..r]$. (We deal only with satisfiable instances.)

This problem can be modelled as follows. The associated constraint network involves a set $\{V_{ij} | i, j \in [1..r]\}$ of r^2 variables representing the r^2 cells. Each variable takes its values in the domain $\{1..r^2\}$. In order to guarantee that the sums are respected, there is a constraint $C_{Sr} = (\Sigma_{j \in [1..r]} V_{ij} = rsum(i))$ on each row i, and a constraint $C_{Sc} = (\Sigma_{i \in [1..r]} V_{ij} = csum(j))$ on each column j. An *alldiff* constraint is defined on all the variables.

With this modelization it is difficult to solve some instances of the problem when r is greater than 6. Thus we have to improve this model, but how can we do it? At present, we are not able to give a general answer to this question. There are few studies on this topic [5, 4]. In fact, it may not be easy to identify the parts of the problem that lend themselves to relevant subproblems. For instance, we would like to know if it is interesting to be able to achieve arc consistency for the constraint "sum of n all different variables". That is, if it would be of interest to replace each constraint C_S constraining the sum of a row or a column, by a global constraint $C_{S\neq}$, which is the conjunction of C_S and of an *alldiff* constraint $C_{\neq}$ involving the same variables as C_S.

The interest of achieving arc consistency for one constraint is the amount of deletions of values that can follow. Then, the other constraints can use these deletions to perform further domain reductions, and so on.

Unfortunately, there is no specific arc consistency algorithm available for $C_{S\neq}$ constraints, and it is not obvious to find one. Writing such a new algorithm will be time-consuming and subtle in itself. Furthermore, we do not know whether it is interesting to introduce such a constraint inside our model or not. Hence, developing such a specific algorithm can lead to a loss of human time.

When dealing with global constraints, the arc consistency mechanism can be seen as the deletion of the values that do not belong to any solution (or allowed tuple) of the subproblem (or constraint). Therefore, arc consistency can be achieved by checking for each value a of each variable x involved in the subproblem whether the subproblem has a solution when x is instantiated with

a. Thus, if we are able to compute a solution for a subproblem, we can achieve arc consistency for its associated global constraint.

Several possibilities can be considered:

(1) All the possible solutions of the subproblem, called allowed tuples, are computed once and for all. They are saved in a database (that can sometimes fit in memory) and the existence of a solution is obtained by searching in the database;
(2) Only the combinations of values that are not solutions are saved. These combinations are called forbidden tuples. Most of the time, this way is chosen when almost all the combinations are solutions;
(3) Finally, solutions can be computed on the fly when the consistency of a value with the global constraint has to be checked.

Bessière and Régin have proposed a general framework, GAC-Schema, in which one can incorporate any specific algorithm to calculate arc consistency on any given kind of constraint [1]. They also provided three instantiations of the schema to efficiently handle constraints defined by a list of allowed tuples (GAC-Schema+allowed), by a list of forbidden tuples (GAC-Schema+forbidden), and by any predicate (GAC-Schema+predicate).

The possibilities (1) and (2) above can be handled by GAC-Schema+allowed and GAC-Schema+forbidden. Their time and space complexity depends on the number of involved tuples. For the example we considered above, these numbers prevent us from using these possibilities. Only the third possibility is acceptable. Therefore, we could use directly GAC-Schema+predicate by representing the subproblem as a predicate. (A tuple would satisfy the predicate iff it is a solution of the subproblem.) However, such an instantiation of GAC-Schema is incapable of using specific knowledge (if available) that underlies the constraint it handles. It will not use the basic constraints composing the subproblem. In other words, the solutions will be computed by a kind of generate-and-test algorithm. This is not acceptable.

In this paper we propose to instantiate GAC-Schema with an algorithm that calls a solver to compute allowed tuples/solutions on the fly. The advantage of this approach is that the knowledge available for the subproblem can be used. Thus, some combinations of values that are forbidden by constraints of the subproblem will be avoided, and then time efficiency improves.

The rest of the paper is organized as follows. In Section 2 we give background on constraint networks and GAC-schema, and recall the property of multidirectionality. In Section 3 we present a simple technique to achieve arc consistency on global constraints by solving the associated subproblem on the fly. This first technique does not take completely multidirectionality into account, whereas it is usually the case in GAC-schema instantiations. In Section 4 we show how to consider multidirectionality as a particular constraint with its own propagation algorithm. Adding it to the subproblem permits to solve it while taking completely multidirectionality into account. Finally, in Section 5, we propose several experiments that show the interest of our approach to efficiently solve some problems, and to identify where are located some difficult parts of a problem. We also study the behavior of the presented algorithm with the classical instantiations of GAC-Schema.

2 Background

2.1 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. A total ordering $<_d$ can be defined on $D(x_i), \forall x_i \in X$, without loss of generality. $highest(D(x_i))$ denotes the highest value in $D(x_i)$ w.r.t. $<_d$.
Constraint. A constraint C on the ordered set of variables $X(C) = (x_{i_1}, \ldots, x_{i_r})$ is a subset 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 instantiation of the variables in $X(C)$ is called a *tuple on* $X(C)$. Two tuples τ and τ' on $X(C)$ can be ordered by the natural lexicographic order $\prec_{lo}$ in which $\tau \prec_{lo} \tau'$ iff $\exists k / \tau[1..k-1] = \tau'[1..k-1]$ and $\tau[k] <_d \tau'[k]$ ($\tau[1..k]$ being the prefix of size k of τ, and $\tau[k]$ the k^{th} value of τ). The tuples on $X(C)$ not allowed by C are called the *forbidden* tuples of C. $|X(C)|$ is the *arity* of C.
Global constraint. A *global constraint* (also called *subproblem* by misuse of language) $C_\Pi = c_1 \wedge \cdots \wedge c_q$ is represented indifferently by its set of (*underlying*) constraints $\{c_1, \ldots, c_q\}$ or by the subnetwork
$\Pi = (X(C_\Pi) = \cup_{c \in C_\Pi} X(c), \mathcal{D}_{X(C_\Pi)}, \{c_1, \ldots, c_q\})$. The set of tuples allowed by C_Π is equal to the set of solutions of Π.
Notation. A value a for a variable x is often denoted by (x, a). $var(C, i)$ represents the ith variable of $X(C)$, while $index(C, x)$ is the position of variable x in $X(C)$.
Arc consistency. Let $\mathcal{N} = (X, \mathcal{D}, \mathcal{C})$ be a constraint network, C a constraint in $\mathcal{C}$. A tuple τ on $X(C)$ is *valid* iff $\forall x \in X(C), \tau[index(C, x)] \in D(x)$; otherwise, it is *rejected*. A value $a \in D(x)$ is *consistent with* C iff $x \notin X(C)$, or $\exists \tau$ allowed by C, such that $a = \tau[index(C, x)]$ and τ is valid. (τ is then called a *support* for (x, a) on C.) C is *arc consistent* iff $\forall x \in X(C), D(x) \neq \varnothing$ and $\forall a \in D(x)$, a is consistent with C. We achieve arc consistency on C by removing every value not consistent with C. Note that a value is consistent with a global constraint C_Π iff it belongs to a solution of the subnetwork Π.

2.2 The GAC-Schema

Let us review the GAC-Schema presented in [1] since the algorithms we will propose in the following sections are instantiations of that schema. GAC-Schema has been built to take into account the *multidirectionality* inherent in any constraint when achieving arc consistency.

```
PROPAGATION(C: constraint; x: variable; a: value, deletionSet: list): Bool
for each τ ∈ S_C(x, a) do
    for each (z, c) ∈ τ do remove τ from S_C(z, c)
    for each (y, b) ∈ S(τ) do
        remove (y, b) from S(τ)
        if b ∈ D(y) then
1           σ ← seekInferableSupport(C, y, b)
2           if σ ≠ nil then add (y, b) to S(σ)
            else
                σ ← seekNextSupport(C, y, b, last_C(y, b))
                if σ ≠ nil then
                    add (y, b) to S(σ)
                    last_C(y, b) ← σ
                    for k = 1 to |X(C)| do add σ to S_C(var(C, k), σ[k])
                else
                    remove b from D(y)
                    if D(y) = ∅ then return False
                    add (y, b) to deletionSet
return True
```

Algorithm 1: function PROPAGATION

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

We say that GAC-Schema takes multidirectionality into account because:

(1) it never looks for a support for a value on a constraint C when a tuple supporting this value has already been checked, and

(2) it never checks whether a tuple is a support for a value when it has already been unsuccessfully checked for another value.

The function PROPAGATION of GAC-Schema is given in Algorithm 1. The program including it must create and initialize the data structures (S_C, S, and $last_C$), and call $Propagation(C, x, a, deletionSet)$ for each constraint C involving x, each time a value a is removed from $D(x)$, in order to propagate the consequences of this deletion. $deletionSet$ is updated to contain the deleted values not yet propagated. S_C, S, and $last_C$ must be initialized in a way such that:

- $S_C(x, a)$ contains all the allowed tuples τ that are the current support for some value, and such that $\tau[index(C, x)] = a$.
- $S(\tau)$ contains all values for which τ is the current support.
- $last_C(y, b)$ is the last tuple returned by SEEKNEXTSUPPORT as a support for (y, b) if $seekNextSupport(C, y, b, -)$ has already been called; nil otherwise. There is an ordering on the tuples, which is proper to SEEKNEXTSUPPORT, and $last_C(y, b)$ gives the point where SEEKNEXTSUPPORT will have to restart the search for a support for (y, b) on C at the next call.

Line 1 of PROPAGATION "infers" an already checked allowed tuple as support for (y, b) if possible, in order to ensure that the point (1) above is satisfied. The search for support (function SEEKNEXTSUPPORT —see line 2 of PROPAGATION) is instantiated differently depending on the type of the constraint involved. The point (2) above must be taken into account by SEEKNEXTSUPPORT.

```
seekInferableSupport(C: constraint; 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 support */
|_ else remove σ from S_C(y,b)
return nil
```

Algorithm 2: function SEEKINFERABLESUPPORT

3 A first algorithm

In this section, we propose a first instantiation of the GAC-schema that deals with global constraints but that does not take into account the point (2) of multidirectionality (see Section 2).

As stated in the introduction, a global constraint is a subproblem that is considered globally as a constraint for which the allowed tuples are the solutions of the subproblem. Let us denote by C_Π such a constraint if $\Pi = (X(C_\Pi), \mathcal{D}_{X(C_\Pi)}, \mathcal{C}_\Pi)$ is the subproblem. Achieving arc consistency on C_Π is done by looking for supports for the values of the variables in $X(C_\Pi)$. A support for a value (y, b) on C_Π can be seeked by any search procedure since a support for (y, b) is a solution of the subnetwork $(X(C_\Pi), \mathcal{D}_{X(C_\Pi)}, \mathcal{C}_\Pi \cup \{y = b\})$.

```
seekNextSupport(C_Π: constraint; y: variable, b: value, τ: tuple ): tuple
Π_2 ← (X(C_Π), D_X(C_Π), {y = b} ∪ C_Π)
σ ← searchForSolution(Π_2, τ)
return σ
```

Algorithm 3: function `seekNextSupport`

The new function SEEKNEXTSUPPORT is given by Algorithm 3. First it defines $\Pi_2 = (X(C_\Pi), \mathcal{D}_{X(C_\Pi)}, \{y = b\} \cup \mathcal{C}_\Pi)$ a new CSP and then function SEARCHFORSOLUTION is called. The specification of SEARCHFORSOLUTION(Π_2, τ) is that it must return a solution to the subnetwork Π_2, knowing that τ was the previous solution returned; or *nil* if there is no solution. Starting a search algorithm with an instantiation for which we know there is no solution before it has already been studied in [2]. However, SEARCHFORSOLUTION can also be implemented as a classical search procedure, without using the previous solution found.

4 Dealing with multidirectionality

4.1 Properties

In Section 3, we have proposed a simple instantiation of GAC-schema to global constraints, which does not guarantee the point (2) of multidirectionality. In this section, we give an algorithm that completely takes into account multidirectionality. From now, a global constraint will be denoted C or C_Π indifferently.

Definition 2 *Given a constraint C on $X(C)$, a valid tuple on $X(C)$ that has never been checked against C is called a* candidate tuple*. A partial instantiation of $X(C)$ that can be extended to $X(C)$ as a candidate tuple is called a* candidate*.*

In order to ensure the point (2) of multidirectionality, it is necessary to define an ordering on the tuples that will be followed when looking for a support for a value on a constraint C. The lexicographic ordering $\prec_{lo}$ defined previously is used to visit the tuples on $X(C)$. The function $seekNextSupport(C, y, b, \tau)$ must return the *smallest* support for (y, b) on C greater than τ w.r.t. $\prec_{lo}$. To deal completely with multidirectionality requires checking only candidate tuples. This is done by means of the following property.

Property 1 *Let σ be a valid tuple on $X(C)$. σ is a candidate tuple iff $\nexists x \in X(C)$ such that $\sigma \prec_{lo} last_C(x, \sigma[index(C, x)])$.*

This property can be implemented in different ways. For instance, GAC-Schema+predicate is a possible implementation. Nevertheless, this implementation is a specific algorithm that cannot use the constraints composing a subproblem Π. It, indeed, only checks the satisfiability of the associated global constraint C_Π, i.e., complete instantiations of $X(C_\Pi)$. Thus, GAC-schema+predicate is a generic algorithm for dealing with multidirectionality, but it is not able to efficiently handle global constraints and their underlying constraints.

In fact, satisfying property 1 can be seen as a particular constraint that guarantees that multidirectionality is used completely during the search for support in the constraint C_Π. However, because of its formulation, property 1 cannot be used by SEEKNEXTSUPPORT to perform look ahead on the domains of the variables of $X(C_\Pi)$ during a search for support. Indeed, property 1 states what a non candidate tuple is, but does not state what an "impossible" value (i.e., a value that cannot belong to any candidate tuple) is, because in chronological backtracking, a value has to be added to a partial instantiation to be detected as "impossible" by property 1. In the following section, we give a reformulation of property 1 that gives a condition under which a value cannot belong to any candidate, and thus does not need to be considered by SEEKNEXTSUPPORT. Then, we present an original and efficient way for taking completely into account multidirectionality and for taking advantage of the knowledge of all the underlying constraints in a global constraint.

4.2 Implementation of multidirectionality as a constraint

The idea is thus to define, for each global constraint C, a new constraint, denoted by M_C, whose filtering algorithm removes the values that do not belong to any candidate during a search for support in C. In this way, the PROPAGATION algorithm associated with this constraint will guarantee the point (2) of multidirectionality without any restriction on the search algorithm used in SEEKNEXTSUPPORT, and thus the constraint C will be efficiently handled.

SEEKNEXTSUPPORT2(C: constraint; y: variable, b: value, τ: tuple):tuple
$\Pi \leftarrow (X(C), \mathcal{D}(X(C)), \{y = b\} \cup C \cup \{M_C\})$
$\sigma \leftarrow$ SEARCHFORSOLUTION(Π, τ)
return σ

Algorithm 4: function SEEKNEXTSUPPORT2

Function SEEKNEXTSUPPORT is modified in order to take into account this new constraint. The specification of SEEKNEXTSUPPORT2(C, y, b, τ), the new version of SEEKNEXTSUPPORT, is the same as for SEEKNEXTSUPPORT, except that the set of constraints of the subnetwork to solve is $\{y = b\} \cup C \cup \{M_C\}$ instead of $\{y = b\} \cup C$.

In order to avoid confusion, let us denote $D^{in}(x)$, the domain of x during the current search for support, as opposed to $D(x)$ the domain of x when that search for support began. In other words, $\mathcal{D}^{in}_{X(C)}$ is the set of domains inside the call of function SEARCHFORSOLUTION.

Definition 3 (breakpoint) *Let $y \in X(C)$, and $b \in D^{in}(y)$. The value k such that $\forall i \in [1..k-1]$, $highest(D^{in}(var(C,i))) = last_C(y,b)[i]$ and $highest(D^{in}(var(C,k))) \neq last_C(y,b)[k]$ is called the* breakpoint *of (y,b) on C w.r.t. $\mathcal{D}^{in}_{X(C)}$, and is denoted by $breakpoint_C(y,b)$.*

The following corollary is a reformulation of property 1.

Corollary 1 *Let $X(C) = (x_{i_1}, \ldots, x_{i_r})$, $y \in X(C)$, $b \in D^{in}(y)$, and $k = breakpoint_C(y,b)$. If $highest(D^{in}(var(C,k))) <_d last_C(y,b)[k]$ then (y,b) cannot belong to any candidate on C in $\mathcal{D}^{in}_{X(C)}$ w.r.t. $\prec_{lo}$.*

Proof. Suppose (y,b) belongs to a candidate partial instantiation σ. By hypothesis (see Definition 3), for any extension τ of σ, $\tau \in D^{in}(x_{i_1}) \times \cdots \times D^{in}(x_{i_r})$ implies that $\tau \prec_{lo} last_C(y,b)$, which is impossible for a candidate tuple.

The following corollary is a particular case of the previous one.

Corollary 2 *Let $x, y \in X(C)$. If $D^{in}(x) = \{a\}$ then,*
$index(C,y) \leq breakpoint_C(x,a)$ and $b <_d last_C(x,a)[index(C,y)]$
$\Rightarrow (y,b)$ cannot belong to any candidate on C in $\mathcal{D}^{in}_{X(C)}$ w.r.t. $\prec_{lo}$.

With these corollaries, we can predict that a value (x_{i_k}, v_{i_k}) cannot belong to any candidate in $D^{in}(x_{i_1}) \times \dots \times D^{in}(x_{i_r})$ without instantiating all the variables from x_{i_1} to x_{i_k}.

We can first define a simple filtering algorithm that only deals with instantiated variables. Due to corollary 2, when a variable x is instantiated with a value a, all the variables that precede $breakpoint_C(x,a)$ in $X(C)$ can be instantiated to the maximal value of their domain. Furthermore, the values less than $last_C(x,a)$ of the variable $var(C, breakpoint_C(x,a))$ can be deleted. Function InstantiatedVar is a possible implementation of this filtering algorithm and is called each time a variable is instantiated. Parameter k of this function represents the breakpoint value of the invoked instantiated variable.

```
InstantiatedVar(C: constraint; y: variable; b: value; k: integer)
for i ∈ [1..k] while i ≤ |X(C)| do
    for a=first(D^in(var(C,k))) to last_C(y,b)[k] − 1 do
        remove a from D^in(var(C,k))
        add (var(C,k), a) to deletionSet
```

Algorithm 5: function InstantiatedVar

A better filtering algorithm can be obtained by the exact application of corollary 1. However, the problem is to determine which values can be deleted by applying this corollary. It is not practicable to recompute the breakpoint value of all the values when a modification arises. In fact, we show that it is not necessary. Consider a value b of a variable y, then $breakpoint_C(y,b)$ can change only in one of the two following cases:

1. $last_C(y,b)$ is changed,
2. the highest value of a domain is removed.

The first case is easy to deal with. For the second, it is necessary to introduce a new data structure in order to determine when some breakpoint values could have changed. When a highest value (x,a) is removed from $D^{in}(x)$, the values (y,b) such that $last_C(y,b)$ contains (x,a) must be reachable. Therefore, we introduce the following list:

$L_C(x,a) = \{(y,b) \text{ s.t. } last_C(y,b) \text{ contains } (x,a)\}$

The filtering algorithm can be defined by implementing the different operations:

- When (x,a) is added to $last_C(y,b)$ then (y,b) is added to $L_C(x,a)$. And, when (x,a) is no longer in $last_C(y,b)$ then (y,b) is removed from $L_C(x,a)$.
- When a variable is instantiated then InstantiatedVar is called.
- When the $last_C$ of an element is changed, the breakpoint value of this element is recomputed.

```
PROPAGATION(M_C: constraint; x: variable; a: value; deletionSet: list)
3  if a > highest(D^in(x)) then
      for (y, b) ∈ L_C(x, a) do
         if b ∈ D^in(y) and breakpoint_C(y, b) ≥ index(C, x) then
            remove b from D^in(y)
            add (y, b) to deletionSet
      for (y, b) ∈ L_C(x, highest(D^in(x))) do
         if b ∈ D^in(y) and breakpoint_C(y, b) = index(C, x) then
            k ← COMPUTEBREAKPOINT(y, b)
            if k ≤ |X(C)| and highest(D^in(var(C, k))) <_d last_C(y, b)[k] then
               remove b from D^in(y)
               add (y, b) to deletionSet
            else
               breakpoint_C(y, b) ← k
```

Algorithm 6: function PROPAGATION

- When a value a is removed from $D^{in}(x)$, we check whether it was the highest value of $D^{in}(x)$ or not (line 3 of PROPAGATION). Then, we check whether corollary 1 is satisfied for each value of $L_C(x,a)$. Then, the breakpoint of each value (y,b) of $L_C(x, highest(D^{in}(x)))$ satisfying $breakpoint_C(y,b) = index(C,x)$ is recomputed and corollary 1 is checked. [1]

This last point is implemented by function PROPAGATION which is called when the highest value of a variable x has been reduced. (Function COMPUTE-BREAKPOINT(y,b) recomputes the breakpoint value of (y,b).)

4.3 Analysis

First, let us report the worst-case space complexity of the algorithms we proposed to achieve arc consistency on C a global constraint. In the case of SEEKNEXTSUPPORT, the only data structures used are those of GAC-Schema. The worst-case space complexity is then in $O(|X(C)|^2 \cdot d)$, where d is the size of the largest domain (see [1]). When we use SEEKNEXTSUPPORT2, some additional data structures are used to implement the propagation procedure of the multidirectionality constraint M_C. But their size is less than $|X(C)|^2 \cdot d$, so the space complexity remains in $O(|X(C)|^2 \cdot d)$. The worst-case time complexity is bounded above by the complexity of the search procedure, SEARCHFORSOLUTION, used to search for supports.

The ordering of the variables that is used in SEEKNEXTSUPPORT2 must be a static variable ordering (SVO). This SVO is required to deal with the constraint

[1] In some solvers, such as ILOG Solver, the values higher than the highest of a domain that have been deleted since the last propagation are provided by the solver. Thus, they can be dealt with together in one turn, improving our algorithm.

of multidirectionality. With the possibility of using look ahead search algorithms such as FC or MAC to search for supports on $X(C)$, it can seem a strong limitation to use them with a SVO. But we have to keep in mind that this restriction holds only for the subnetwork defined on $X(C)$ by the constraints in C. Nevertheless, in cases where a SVO in the search for support would be a too strong limitation, we can relax the constraint of using complete multidirectionality and use SEEKNEXTSUPPORT with a search algorithm with dynamic variable ordering (DVO). The counterpart is that only one side of multidirectionality is used, the one performed in SEEKINFERABLESUPPORT (i.e., never seeking a support for a value when a tuple supporting it has already been checked). We cannot guarantee that we never check whether a tuple is a support for a value when it has already been unsuccessfully checked for another value.

5 Experimental results

In this section we will present three kinds of experiments. Here is an overall presentation of them:

1. The first one aims at showing that computing arc consistency of a subproblem can greatly improve the resolution of the whole problem. It also shows that computing the solutions of the subproblem on the fly can be sufficient in itself to improve the resolution of the whole problem, even without developing a specific algorithm;
2. The second kind of experiments stresses on the interest of our approach for exhibiting the difficult parts of a problem;
3. The last experiments compare our approach to GAC-Schema+allowed on instances favorable for this algorithm. This proves that even for such problems our algorithm behaves well.

Furthermore, they all show that GAC-Schema+predicate can always be replaced by the algorithm we presented.

5.1 Improvement of the resolution

First, we report a small sample of the results we obtained for the problem defined in the introduction of this paper.

The results presented in the following are obtained with a MAC search procedure with the "minimum domain" variable ordering heuristic. The procedure used to search for supports inside the arc consistency process of a constraint representing a subproblem is also a MAC procedure. It uses a lexicographic variable ordering. These MAC procedures are those of ILOG Solver [3] without any special features except those presented in this paper.

Let us compare three possible representations of the problem. The first one, denoted by stand, corresponds to the standard representation. It does not involve any constraint "sum of n all different variables" (denoted by $C_{S\neq}$). The second one, denoted by pred, represents each constraint $C_{S\neq}$ as a predicate and uses

GAC-Schema+predicate to handle it. In the third one, denoted by conj, each global constraint $C_{S\neq}$ is handled by the new algorithm we have proposed in this paper.

For 5×5 square:

	# backtracks			cpu time (sec.)		
	stand	pred	conj	stand	pred	conj
easiest	4	0	0	0	8.5	0.5
average	2373	127	127	0.3	18	1.7
hardest	9985	591	591	1.36	51	7.2

The search space is reduced by introducing the $C_{S\neq}$ constraints. The results show that a dedicated constraint should be written, because the overall time performance is not improved, but the number of backtracks is dramatically reduced. However, we have to check if this behavior is true when the size of the square increases.

For 6×6 square:

	# backtracks			cpu time (sec.)		
	stand	pred	conj	stand	pred	conj
easiest	3	0	0	0.03	too long	2.5
average	75,548	281	281	26.2	too long	6.7
hardest	1,623,557	2,598	2,598	520	too long	42.0

In this case it is not possible to use GAC-Schema+predicate because it needs too much time. The new technique becomes really interesting. For the 5×5 square, in average, the number of backtracks is reduced by a factor of 18 compared to the standard modelling. For the 6×6 square the factor becomes 269. Further experiments have shown that this factor increases when the size of the square increases. Thus, there is no doubt about the interest of the $C_{S\neq}$ constraints.

On the other hand, the algorithm we proposed is the best existing way to solve this problem. This means that for the 6×6 square, the best way is to solve subproblems on the fly. This kind of result is not always observed but it is interesting to mention it. In some cases, there is no need to develop a specific algorithm, our approach is in itself sufficient to improve the resolution of a hard problem.

5.2 Identification of the difficulty

Let us consider another problem: the Golomb Ruler Problem. This problem is problem prob006 of the CSPLib benchmark library. Here is its specification taken from this library: " A Golomb ruler may be defined as a set of n integers $0 = x_1 < x_2 < ... < x_n$, such that the $n(n-1)/2$ differences $x_j - x_i$, $1 \leq i < j \leq n$, are distinct. Such a ruler is said to contain n marks and is of length x_n. The objective is to find optimal (minimum length) or near optimal rulers."

This problem is hard to solve with constraint programming. It is difficult to obtain solutions for $n > 13$. We study three possible modelizations of the problem.

The first one is classical: each integer and each difference between integer is represented by a variable, an alldiff constraint involving all the differences is defined.

Since this problem is hard to solve, we need to try to identify some difficult parts, and parts that could be improved.

The second model introduces a $C_{S\neq}$ constraint involving variables $x_i - x_{i-1}$. Arc consistency for this constraint is achieved by using the algorithm we propose in the paper, because even for small n it is impossible to use GAC-Schema+predicate.

The third model proposes to focus on a particular subpart of the Golomb Ruler Problem. In the previous models, the alldiff constraint involves $O(n^2)$ variables. We propose to study a subproblem which involves only $O(n)$ variables. This subproblem is defined by the following constraints:

- a sum constraint: $\sum_{i=2}^{n}(x_i - x_{i-1}) = x_n$
- an alldiff constraint defined on $\{x_i, i = 1..n\} \cup \{(x_i - x_{i-1}), i = 2..n\} \cup \{(x_i - x_{i-2}), i = 3..n\}$

For all the experiment we fix the value of x_n and search for a solution. Here are some results (time is given in seconds):

	$n=8$				$n=9$				$n=10$			
	$x_n = 34$		$x_n = 33$		$x_n = 44$		$x_n = 43$		$x_n = 55$		$x_n = 54$	
	#bk	time	#bk	time	#bk	time	#bk	time	#bk	time	#bk	time
model 1	22	0.3	297	0.4	213	0.4	1298	2.7	844	2.2	5326	19
model 2	3	0.3	122	2.8	48	2.4	343	18.2	183	16.6	1967	161
model 3	0	1.4	5	1.5	4	10.5	25	18.1	16	120	96	226

The comparison of model 1 and model 2 shows that the introduction of the $C_{S\neq}$ constraint reduces the number of backtracks by a factor around 4, which is less than with the previous experiments. The time needed by model 2 is greater than the time needed by model 1. Thus, for this experiment it is necessary to develop a specific algorithm for this constraint if we want to improve in time the resolution of the problem.

With model 3 the number of backtracks is dramatically reduced, but the time needed to achieve arc consistency for the subproblem we consider is large. Thus, this model can be used only if a specific filtering algorithm is written. From this result we also learn that it seems that if we are able to solve quickly the subproblem we will be able to solve the Golomb Ruler Problem efficiently. This is particularly interesting because this subproblem involves only $O(n)$ variables, whereas the Golomb Ruler Problem involves $O(n^2)$ variables.

In conclusion, for this experiment the algorithm we proposed is not able to improve in time the resolution of the problem. However, with it we have identified a subproblem for which the design of a specific algorithm would improve the resolution of the whole problem. We have understood where a part of the difficulty of the problem is.

5.3 Comparison with GAC-Schema+allowed

Finally, we propose to study a configuration problem. The general formulation is this: given a supply of components and bins of given types, determine all assignments of components to bins satisfying specified assignment constraints subject to an optimization criterion.

In the example we will consider that there are 5 types of components: {glass, plastic, steel, wood, copper}. There are three types of bins: {red, blue, green} whose capacity constraints are: red has capacity 5, blue has capacity 5, green has capacity 6.
The containment constraints are:

- red can contain glass, copper, wood
- blue can contain glass, steel, copper
- green can contain plastic, copper, wood

The requirement constraints are (for all bin types): wood requires plastic.
Certain component types cannot coexist: glass exclusive copper
Certain bin types have capacity constraint for certain components:

- red contains at most 1 of wood
- green contains at most 2 of wood

- for all the bins there is either no plastic or at least 2 plastic.

Given an initial supply of: 12 of glass, 10 of plastic, 8 of steel, 12 of wood, and 8 of copper; what is the minimum total number of bins required to contain the components?

A description of a possible implementation of a similar problem is given in [3]. We will call it "standard model".

Almost all the constraints between types of bins and components are local. The filtering algorithm associated with them leads to few domain reductions. Therefore, we propose to group them inside a single global constraint. Then we can test GAC-Schema+allowed against the algorithm we proposed. It is possible to use GAC-Schema+allowed because the number of allowed tuples is small: 79. They can be computed and stored in a preprocessing phase. However, in real-world applications the number of components can largely grow and this method will no longer be possible.

Here are the results:

	# Backtracks	time (s)
standard model	1,361,709	430
allowed	12,659	9.7
new algorithm	12,659	11

Once again it is quite interesting to represent a part of a problem by a constraint. Moreover, we observe that our algorithm does not loose too much time compared to GAC-Schema+allowed, while this problem was an ideal case for GAC-Schema+allowed.

6 Conclusion

Following the intuition that "the more globally you filter, the more able you are to tackle combinatorial explosion", we proposed a new instantiation of GAC-Schema which achieves arc consistency on global constraints by solving subproblems on the fly. We proved that the knowledge of a subproblem and multidirectionality can be taken into account in a generic way by adding to the considered subproblem a special constraint representing multidirectionality. This instantiation clearly outperforms the technique in which the subproblem is considered as a predicate without the possibility of using underlying knowledge (GAC-Schema+predicate).

The experiments we presented, show that as soon as a problem is difficult enough, the cost of solving subproblems on the fly can be outweighed by the reduction of the search space it implies.

In addition, beyond efficiency concerns, we pointed out that in constraint programming, when a problem is hard to solve, the new instantiation of GAC-Schema we proposed can be very helpful in the modelling phase to isolate parts of the problem on which writing a dedicated filtering algorithm will have chances to pay off.

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. P. Van Hentenryck. Incremental constraint satisfaction in logic programming. In *proceedings ICLP'90*, pages 189–202, Jerusalem, Israel, 1990.
3. ILOG. *ILOG Solver 4.4 User's manual.* ILOG S.A., 1999.
4. J-C. Régin. Minimization of the number of breaks in sports scheduling problems using constraint programming. In *proceedings DIMACS Workshop on Constraint Programming and Large Scale Discrete Optimization*, pages P7:1–23, 1998.
5. B.M. Smith. Succeed-first or fail-first: A case study in variable and value ordering. In *proceedings ILOG Solver and ILOG Scheduler Second International Users' Conference*, Paris, France, 1996.