# Arc consistency for general constraint networks: preliminary results

**Christian Bessière**
LIRMM–CNRS (UMR 5506)
161 rue Ada
34392 Montpellier cedex 5, France
Email: `bessiere@lirmm.fr`

**Jean-Charles Régin**
ILOG S.A.
9 rue de Verdun
94253 Gentilly Cedex, France
Email: `regin@ilog.fr`

## Abstract

Constraint networks are used more and more to solve combinatorial problems in real-life applications. Much activity is concentrated on improving the efficiency of finding a solution in a constraint network (the constraint satisfaction problem, CSP). Particularly, arc consistency caught many researchers' attention, involving the discovery of a large number of algorithms. And, for the last two years, it has been shown that maintaining arc consistency during search is definitely a worthwhile approach. However, results on CSPs and on arc consistency are almost always limited to binary constraint networks. The CSP is no longer an academic problem, and it is time to deal with non-binary CSPs, as widely required in real world constraint solvers. This paper proposes a general schema to implement arc consistency on constraints of any arity when no specific algorithm is known. A first instantiation of the schema is presented here, which deals with constraints given by a predicate, by a set of forbidden combinations of values, or by a set of allowed ones.

## 1 Introduction

### 1.1 Overview

Constraint satisfaction problems (CSPs) occur widely in artificial intelligence. They are used more and more in real-life applications, such as scene analysis, resource allocation, crew scheduling, time tabling, frequency allocation, car sequencing, etc. The CSP involves finding a solution in a constraint network, i.e. finding values for problem variables subject to constraints.

The general task of solving a CSP being NP-hard, many researchers have concentrated on improving the efficiency of finding a solution in a constraint network. Particularly, arc consistency caught many researchers' attention, involving the discovery of a large number of algorithms [Mackworth, 1977a], [Mohr and Henderson, 1986], [Bessière, 1994], [Bessière *et al.*, 1995]. And, recently, the value of these studies on arc consistency increased since it has been shown that maintaining arc consistency during search is definitely a worthwhile approach when solving large and hard problems [Sabin and Freuder, 1994], [Bessière *et al.*, 1995], [Bessière and Régin, 1996], [Grant and Smith, 1996].

However, results about CSP solving and about arc consistency are almost always limited to *binary* constraint networks, justified by the fact that any non-binary constraint network can be translated into an equivalent binary one with additional variables [Rossi *et al.*, 1990]. But, in practical cases, it is often inconceivable to translate a non-binary constraint into an equivalent set of binary ones with the technique cited above because of the underlying computational and memory cost. Sometimes, when a constraint is *representable* [Montanari, 1974]), it is possible to replace it by a set of binary constraints without introducing new variables, and without generating the set of allowed tuples of each binary constraint. For example, if a constraint states that the three variables $x_1$, $x_2$, and $x_3$ must all take different values, we can equivalently state that there is an inequality constraint on each of the pairs of variables $\{x_1, x_2\}$, $\{x_1, x_3\}$, and $\{x_2, x_3\}$. But, as it has been shown in [Régin, 1994], non-binary constraints lose a part of their semantics when encoded into a set of binary constraints in this way. This leads, for example, to less pruning for arc consistency algorithms handling them.

Hence, if we no longer want to consider the CSP as an academic problem we must be able to deal with any kind of constraints, and so, with non-binary constraints, as widely required in real-world constraint solvers.

### 1.2 Previous work

In the constraint satisfaction community, the number of works involving non-binary constraint networks is rather small. For the particular case of arc consistency, we know two algorithms capable of achieving it. Mackworth [Mackworth, 1977b] proposed the algorithm `CN`, which is a kind of generalization of AC-3 to non-binary constraints. As AC-3, that algorithm has a bad worst-case time complexity ($O(er^2d^{r+1})$, with $e$ the number of constraints in the network, $r$ the maximal arity of the constraints, and $d$ the size of the largest domain). Mohr and Massini [Mohr and Masini, 1988] proposed `GAC4`. It is based on the same idea as AC-4 (computing the number of supports for each value in each domain and removing those with this number equal to zero). Thus, it gets rid of the huge worst-case time complexity of `CN` (`GAC4` is in $O(ed^r)$), but has the same drawbacks as AC-4: space complexity (because of the lists of supported values), and average time complexity. The respective drawbacks of `CN` and `GAC4` are even more important than on their binary versions, so that `CN` can only be applied on ternary constraints and very small domains, and `GAC4` on very tight constraints, where the number of allowed tuples of values is very small.

Finally, we can point out the work of van Beek and Dechter [van Beek and Dechter, 1995], who proposed another definition for arc consistency on non-binary constraint networks, namely *relational arc consistency*. This definition is much stronger than the classical one since for each constraint it requires global consistency on the underlying subnetwork (i.e. the network involving the variables of the given constraint, and all the "smaller" constraints defined on some of these variables).

### 1.3 Our purpose

Our general aim is to propose a new schema in order to perform arc consistency on any real world constraint network with a reasonably low time and space complexity. In industrial applications, a constraint can be given in many different forms: by the set of allowed tuples in extension (generally when the constraint is very tight), by the set of forbidden tuples in extension (when the constraint is very loose), by a conjunctive constraint, by an arithmetic relation, or by any predicate for which no particular semantics is known (data base query, user's context-dependent constraint, etc.). Thus, a suitable arc consistency algorithm should be able to efficiently handle any of these constraints.

The particular aim of this paper is to present the first steps of our work. First, we will present our general schema for arc consistency on non-binary constraint networks. This schema is based on the AC-7 schema given in [Bessière *et al.*, 1995]. It makes use of the "current support" idea, and of "multidirectionality" (the generalization of bidirectionality to non-binary constraints) in order to save as many constraint checks as possible. Second, we will instantiate this schema with some of the most frequently occurring forms of constraint representations.

Constraints defined by a predicate for which no particular semantics is known are one of the most important forms that have to be handled. Indeed, if we except the predicates for which specific algorithms are known (arithmetic relations [Van Hentenryck *et al.*, 1992], cardinality constraints [Régin, 1996], etc.), the only available algorithms for non-binary constraints are `CN` and `GAC4`. `GAC4` deals only with constraints given in extension, and then is not capable of handling a predicate, even if, theoretically, it is always possible to build the whole set of allowed tuples by checking all the combinations of values for the variables involved in the constraint. It would be impracticable, at least because of space requirements. Moreover, asking the value of a predicate for a particular combination of values (a constraint check) can be very costly in practice, when the answer is contained in a database, or requires heavy computation. This definitively eliminates `CN`, with its huge time complexity, doing and doing again many times the same constraint checks. Therefore, we will propose an instantiation of the schema that will efficiently handle predicates, minimizing the number of constraint checks. However, this schema is certainly not competitive on predicates for which a specific algorithm is available.

Afterwards, we will instantiate the schema for constraints defined by a set of forbidden tuples. This kind of representation is of practical interest when constraints are very loose.

Finally, we will show that the schema can also deal with constraints given in extension by the list of allowed tuples. It leads to an improvement on `GAC4`, which was already written for that kind of constraint.

## 2 Preliminaries

**Constraint network.** A finite *constraint network* $\mathcal{P}$ is defined as a set of $n$ *variables* $X = \{x_1, \dots, x_n\}$, a set of current *domains* $\mathcal{D} = \{D(x_1), \dots, D(x_n)\}$ where $D(x_i)$ is the finite set of possible *values* for variable $x_i$, and a set $\mathcal{C}$ of *constraints* between variables. We introduce the particular notation $\mathcal{D}_0 = \{D_0(x_1), \dots, D_0(x_n)\}$ to represent the set of initial domains of $\mathcal{P}$. Indeed, we consider that any constraint network $\mathcal{P}$ can be associated with an initial domain $\mathcal{D}_0$ (containing $\mathcal{D}$), on which constraint definitions were stated. A total ordering $<_d$ can be defined on $D(x_i), \forall x_i \in X$, without loss of generality.
**Constraints.** A constraint $C$ on the ordered set of variables $X(C) = (x_{i_1}, \dots, x_{i_r})$ is a subset of the Cartesian product $D_0(x_{i_1}) \times \cdots \times D_0(x_{i_r})$ that specifies the *allowed* combinations of values for the variables $x_{i_1} \times \ldots \times x_{i_r}$. An element of $D_0(x_{i_1}) \times \cdots \times D_0(x_{i_r})$ is called a *tuple on* $X(C)$. Two tuples $\tau$ and $\tau'$ 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 $\tau$, and $\tau[k]$ the $k^{th}$ value of $\tau$). The tuples of

$D_0(x_{i_1}) \times \cdots \times D_0(x_{i_r})$ not allowed by $C$ are called the *forbidden* tuples of $C$. Verifying whether a given tuple $\tau$ is allowed by $C$ or not is called a *constraint check.* $|X(C)|$ is the *arity* of $C$.

A constraint $C$ involving the subset of variables $X(C) = (x_{i_1}, \ldots, x_{i_r})$ can be defined by the set of allowed tuples (resp. the set of forbidden tuples) given in extension when the constraint is tight (resp. is loose), or by an arithmetic relation. More generally, it can be represented by any Boolean function $f_C$ defined on $D_0(x_{i_1}) \times \cdots \times D_0(x_{i_r})$.

**Solutions.** A *solution* of a constraint network is an instantiation of the variables such that all the constraints are satisfied.

**Notation.** A value $a$ for a variable $x$ is often denoted by $(x, a)$. $\mathsf{var}(C, i)$ represents the i$^{th}$ variable of $X(C)$, while $\mathsf{index}(C, x)$ is the position of variable $x$ in $X(C)$.

**Arc consistency.** Let $\mathcal{P} = (X, \mathcal{D}, \mathcal{C})$ be a constraint network, $C$ a constraint in $\mathcal{C}$.

A tuple $\tau$ of $X(C)$ is *valid* if $\forall (x, a) \in \tau, a \in D(x)$; 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 $a = \tau[\mathsf{index}(C, x)]$ and $\tau$ is valid. ($\tau$ is then called a *support* for $(x, a)$ on $C$.)

$C$ is *arc consistent* iff $\forall x_i \in X(C), D(x_i) \neq \emptyset$ and $\forall a \in D(x_i), a$ is consistent with $C$.

$\mathcal{P}$ is *arc consistent* iff all the constraints of $\mathcal{C}$ are arc consistent.

We achieve arc consistency in $\mathcal{P}$ by removing every value not consistent with at least one constraint in $\mathcal{C}$.

## 3 A general schema for arc consistency

As we pointed out in Section 1, our general aim is, on the one hand, to provide a schema sufficiently general to be instantiated with any kind of constraint. On the other hand, this schema must be powerful enough to avoid as many constraint checks as possible when achieving arc consistency.

We did not choose an AC-3 like or an AC-4 like schema for efficiency reasons. To be reasonably efficient, the schema has to be based on AC-6 or AC-7. They are both based on the search of a single support for each value, the worst-case time complexity being optimal in both algorithms. The difference between them is whether or not they deal with bidirectionality [Bessière *et al.*, 1995].

In a non-binary constraint network, any constraint is multidirectional, as any constraint is bidirectional in a binary constraint network. *Multidirectionality*, indeed, is the fact that for any constraint $C$, a tuple $\tau$ on $X(C)$ is a support for the value $\tau[\mathsf{index}(C, x)]$ ($x$ being a variable involved in $C$) iff $\forall y \in X(C)$, $\tau$ is a support for $\tau[\mathsf{index}(C, y)]$. We say that an algorithm "deals with" multidirectionality iff it never checks whether a tuple is a support for a value when it has already been checked for another value, and never looks for a support for a value on a constraint $C$ when a tuple supporting this value has already been checked.

**Algorithm 1:** function `propagation`

`propagation` (**in** $C$: constraint; **in** $x$: variable;
  **in** $a$: value;
  **in out** deletionStream: list)**: boolean**

**for each** $\tau \in S_C(x, a)$ **do**
  **for each** $(z, c) \in \tau$ **do** remove $\tau$ from $S_C(z, c)$;
  **for each** $(y, b) \in S(\tau)$ **do**
    remove $(y, b)$ from $S(\tau)$;
    **if** $b \in D(y)$ **then**
      $\sigma \leftarrow$ `seekInferableSupport`$(C, y, b)$;
      **if** $\sigma \neq$ ***nil*** **then**
        add $(y, b)$ in $S(\sigma)$ ;
      **else**
        $\sigma \leftarrow$ `seekNextSupport`$(C, y, b, last_C(y, b))$;
        **if** $\sigma \neq$ ***nil*** **then**
          add $(y, b)$ in $S(\sigma)$ ;
          $last_C(y, b) \leftarrow \sigma$ ;
          **for** $k$ **from** *1* **to** $|X(C)|$ **do**
            add $\sigma$ in $S_C(\mathsf{var}(C, k), \sigma[k])$ ;
        **else**
          remove $b$ from $D(y)$ ;
          **if** $D(y) = \emptyset$ **then return false** ;
          add $(y, b)$ in deletionStream ;
**return true** ;

And, dealing with multidirectionality appears to be promising on non-binary constraints. A tuple $\tau$, indeed, allowed by a constraint $C$, can support $|X(C)|$ values. Thus the possible savings are much more important than on binary constraints. AC-7 could save $d^2$ constraint checks thanks to bidirectionality, where $d$ is the domain size of the variables involved (see [Bessière *et al.*, 1995]), while $d^r$ constraint checks can be saved thanks to multidirectionality on an $r$-ary constraint.

The framework we propose is then an AC-7 like schema in which search for support (function `seekNextSupport`) is instantiated differently depending on the type of the constraint involved. This schema (named `GAC-schema`) is able to handle any type of constraint, as soon as the corresponding function `seekNextSupport` is available. Before giving the `seekNextSupport` function associated with the types of constraints we will study in this paper (in the next section), let us describe the function `propagation` (see Alg. 1). The program including it must create and initialize the data structures ($S_C, S$, and $last_C$), and call `propagation`$(C, x, a, \mathsf{deletionStream})$ for each constraint $C$ involving $x$, each time a value $(x, a)$ is removed from $D(x)$, in order to propagate the consequences of this deletion.

$S_C$, $S$, and $last_C$ must be initialized in a way such that:

- $S_C(x,a)$ contains all the allowed tuples $\tau$ that are the current support for some value, and such that $\tau[\mathsf{index}(C,x)] = a$.
- $S(\tau)$ contains all values for which $\tau$ 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. $last_C(y,b)$ is perhaps no longer a support in $C$ for $(y,b)$ (if it is no longer valid and `seekInferableSupport` has found a new support for $(y,b)$). It is just here to give the point where `seekNextSupport` will have to restart the search for a support for $(y,b)$ on $C$ at the next call. There is an underlying ordering on the tuples, which is proper to `seekNextSupport`.

**Property 1** $\forall C \in \mathcal{C}, \forall$*allowed tuple* $\tau$ : $S(\tau) \neq \emptyset \Leftrightarrow \forall x \in X(C), \tau \in S_C(x, \tau[\mathit{index}(C,x)])$.

**Property 2** *Given any constraint $C$ of arity $r$, the space complexity of the data structure of* `GAC-schema` *for $C$ is $O(r^2 d)$.*

**Proof.** Each value has at most one support, then $\sum |S(\tau)| \leq rd$ and there are at most $rd$ tuples in memory. Each value is associated with one $last_C$, so the $last_C$ data structure requires $rd$ pointers. A tuple contains $r$ elements, thus the set of all tuples that support at least one value can be represented in $O(r^2 d)$. The number of elements that belong to $S_C$ is bounded above by $r^2 d$ by property 1 and because $\sum |S(\tau)| \leq rd$. Since an element of an $S_C$ list corresponds to a pointer to a tuple, the space complexity of the data structures of `GAC-schema` for $C$ is $O(r^2 d)$. ∎

Each time a value $(x,a)$ is removed from $D(x)$, we must propagate this deletion to each constraint $C$ involving $x$. So, for all the values $(y,b)$ that were supported by a tuple containing $a$ in position $\mathsf{index}(C,x)$, we must find another support (line 2). To take multi-directionality into account, we first check (lines 3 and 4) if there exists a valid tuple containing $(y,b)$ that is already the current support for another value (function `seekInferableSupport`, Alg. 2). If not, the function `seekNextSupport` will look for another support for $(y,b)$, starting the search from $last_C(y,b)$ (line 5). If a new support $\sigma$ is found for $(y,b)$ on $C$, $(y,b)$ is recorded as being currently supported by $\sigma$ (line 6), and for each value contained in $\sigma$, we store the fact that $\sigma$ is the current support for some value (line 7); otherwise, $(y,b)$ has to be removed (lines 8 to 9).

# 4 The main types of constraint representations

In this section, we will present the instantiation of the schema in order to deal with three types of constraints.

**Algorithm 2:** function `seekInferableSupport`

`seekInferableSupport` (**in** $C$: constraint; **in** $y$: variable; **in** $b$: value)**:** tuple

**while** $S_C(y,b) \neq \emptyset$ **do**
  $\sigma \leftarrow first(S_C(y,b))$ ;
  **if** $\exists k/\sigma[k] \notin D(\mathit{var}(C,k))$ **then** remove $\sigma$ from $S_C(y,b)$ ;
  **else return** $\sigma$  /* $\sigma$ is a support */ ;
**return nil** ;

Constraints given by a predicate for which we do not know specific algorithms is the most important type. Indeed, there does not exist any tool to process a predicate in reasonable time and space. Afterwards, we will present a type of constraint which can be viewed as a special case of predicate constraints: constraints given by the set of forbidden tuples. Finally, we will give the `seekNextSupport` function capable of processing constraints given by the set of allowed tuples.

## 4.1 Predicates

When a constraint $C$ is defined by a predicate for which no particular semantics is known, it is necessary to define an ordering on the tuples that will be followed when looking for a support for a value $(y,b)$ on $C$. Thanks to the function `nextTuple` (Alg. 3), the lexicographic order $\prec_{lo}$ defined in Section 2 is used to examine the valid tuples on $X(C)$. Given any tuple $\tau$ on $X(C)$, and any index $k \leq |X(C)|$, `nextTuple`$(C,y,b,\tau,k)$ will return the smallest valid tuple $\sigma$ greater than $\tau$ w.r.t. $\prec_{lo}$, such that the value of $y$ in $\sigma$ is $b$, and such that the prefixes of size $k$ of $\sigma$ and $\tau$ are not equal. The second parameter returned by `nextTuple` is the smallest index $k'$ on which $\sigma$ and $\tau$ have different values.

When the function `seekNextSupport` (see Alg. 4) is called to search for a new support for $(y,b)$ on $C$, we know from `seekInferableSupport` that there does not exist any tuple already checked that supports $(y,b)$ on $C$. However, we have to take care to avoid checking a tuple $\tau$ which has already been unsuccessfully checked for another value of another variable of $X(C)$. We name *candidate* a valid tuple which has never been checked and thus, which could be a support for $(y,b)$, as op-

**Algorithm 3:** function `nextTuple`

`nextTuple` (**in** $C$: constraint; **in** $y$: variable; **in** $b$: value; **in** $\tau$: tuple; **in** $k$: index)**:** (tuple, index)

/* **returns** $(\sigma, k')$ where:

- $\sigma$ is the smallest valid tuple such that $\sigma[\mathsf{index}(C,y)] = b$, $\tau \prec_{lo} \sigma$ and $\sigma[1..k] \neq \tau[1..k]$
- $k'$ is such that $\sigma[1..k'-1] = \tau[1..k'-1]$ and $\sigma[k'] \neq \tau[k']$

otherwise **returns** (**nil**,–) */

Note that $k'$ will always be less than or equal to $k$.

**Algorithm 4:** function `seekNextSupport`

`seekNextSupport` (**in** $C$: constraint; **in** $y$: variable;
**in** $b$: value; **in** $\tau$: tuple)**:** tuple

1 **if** $\tau \neq nil$ **then**
  $(\sigma, dummy) \leftarrow$ `nextTuple`$(C, y, b, \tau, |X(C)|)$;
**else**
  **for** $i$ **from** 1 **to** $|X(C)|$ **do**
    $\sigma[i] \leftarrow first(\mathsf{var}(C, i))$ ;
2   $\sigma[\mathsf{index}(C, y)] \leftarrow b$ ;
3 $\sigma \leftarrow$ `seekCandidateTuple`$(C, y, b, \sigma, 1)$ ;
$found \leftarrow$ **false**;
4 **while** $(\sigma \neq nil)$ ***and*** $($**not** $found)$ **do**
5   **if** $f_C(\sigma)$ **then** $found \leftarrow$ **true** ;
  **else**
    $(\sigma, k) \leftarrow$ `nextTuple`$(C, y, b, \sigma, |X(C)|)$;
6     $\sigma \leftarrow$ `seekCandidateTuple`$(C, y, b, \sigma, k)$ ;
**return** $\sigma$;

**Algorithm 5:** function `seekCandidateTuple`

`seekCandidateTuple` (**in** $C$: constraint; **in** $y$: variable;
**in** $b$: value;
**in** $\sigma$: tuple; **in** $k$: index)**:** tuple

**while** $(\sigma \neq$ ***nil***$)$ ***and*** $(k \leq |X(C)|)$ **do**
1   /* $\sigma$ is candidate till index $k-1$ */;
  **if** $last_C(\mathsf{var}(C, k), \sigma[k]) \neq$ ***nil*** **then**
2     $\lambda \leftarrow last_C(\mathsf{var}(C, k), \sigma[k])$ ;
    $split \leftarrow 1$ ;
3     **while** $\sigma[split] = \lambda[split]$ **do**
      $split \leftarrow split + 1$ ;
4     **if** $\sigma[split] < \lambda[split]$ **then**
5       **if** $split < k$ **then**
6         $(\sigma, k') \leftarrow$ `nextTuple`$(C, y, b, \sigma, k)$ ;
7         $k \leftarrow k' - 1$ ;
      **else**
8         $(\sigma, k') \leftarrow$ `nextTuple`$(C, y, b, \lambda, |X(C)|)$ ;
9         $k \leftarrow min(k, k' - 1)$ ;
  $k \leftarrow k + 1$ ;
**return** $\sigma$ ;

posed to the valid tuples that have already been checked not to be allowed by $C$ (when we look for support for another value). Dealing with multidirectionality implies only checking the candidates. This can be done thanks to the function `seekCandidateTuple` (see Alg. 5).

`seekCandidateTuple`$(C, y, b, \sigma, k)$ returns the smallest (w.r.t.$\prec_{lo}$) candidate greater than or equal to $\sigma$, where $\sigma$ is valid and $\sigma[1..k-1]$ has been verified to be a possible prefix for a candidate. For each index from $k$ to $|X(C)|$, `seekCandidateTuple` verifies whether $\sigma$ is greater than $last_C(\mathsf{var}(C, k), \sigma[k])$, (denoted by $\lambda$) (lines 2 to 4). If $\sigma$ is smaller than $\lambda$, the search for a candidate has to jump forward: either to the smallest valid tuple following $\sigma$ with a prefix different from $\sigma[1..k]$ (if $\sigma$ and $\lambda$ were diverging before $k$) (line 6), or directly to the valid tuple following $\lambda$ (if $\sigma$ and $\lambda$ were diverging after $k$) (line 8). When we jump forward to the next valid tuple greater than $\sigma$ or $\lambda$, some values before index $k$ may have changed. Then, the value $k$ goes back to the smallest index where the value of $\sigma$ has changed (lines 7 and 9) to keep the property of line 1. When $k$ reaches $|X(C)| + 1$, $\sigma$ is a candidate and is returned.

The function `seekNextSupport`$(C, y, b, \tau)$ returns the smallest tuple greater than $\tau$ which is checked to be allowed by $C$. From line 1 to line 2, `seekNextSupport` assigns to $\sigma$ the smallest valid tuple following $\tau$ (depending on whether $\tau$ is **nil** or not). Afterwards, $\sigma$ is assigned to the smallest candidate (line 3). The search for a support for $(y, b)$ on $C$ is done in lines 4 to 6: we check $\sigma$ and jump to the next candidate until $f_C(\sigma)$ returns **true** (line 5).

### A sketch of proof.

We will simply show here that `seekCandidateTuple` cannot miss any candidates when jumping forward in lines 6 and 8.

**line 6:** suppose there is a candidate $\sigma'$ between $\sigma$ and the tuple returned by `nextTuple`$(C, y, b, \sigma, k)$. This means that $\sigma'[1..k] = \sigma[1..k]$; otherwise $\sigma'$ would be the tuple returned by `nextTuple`. So, $\sigma'$ is smaller than $\lambda$ because of lines 4 and 5. $\sigma'[k]$ being equal to $\sigma[k]$, $\sigma'$ cannot be a candidate because $\lambda = last_C(\mathsf{var}(C, k), \sigma[k])$ (see the definition of $last_C$).

**line 8:** suppose there is a candidate $\sigma'$ between $\sigma$ and $\lambda$. Then, $\sigma'[1..k]$ is equal to $\sigma[1..k]$ since $\sigma[1..k] = \lambda[1..k]$ (lines 3 to 5). Once again that is impossible because $\lambda = last_C(\mathsf{var}(C, k), \sigma[k])$. Finally, $\sigma'$ cannot be equal to $\lambda$ since $\lambda$ is no longer a valid tuple. (Otherwise `seekInferableSupport` would have found it to support $(y, b)$.) ■

## 4.2 Constraints given in extension

### Negative constraints

In this case, the constraint $C$ is given in extension by the set of forbidden tuples, denoted by $\overline{T}(C)$. Even without loss of generality, we can assume that the constraint is very loose. (Otherwise, the space in memory needed prohibits us from using that representation.) Then $\overline{T}(C)$ contains few elements with regard to the set of tuples on $|X(C)|$. So, if we use the previous method with the predicate $f_C(\sigma)$ such that $f_C(\sigma) \Leftrightarrow \sigma \notin \overline{T}(C)$, then only a few constraint checks will be needed to find a new support or to prove there is none, because almost all valid tuples are candidates.

Such a predicate $f_C(\sigma)$ can be efficiently implemented by using a method like hashing [Sedgewick, 1990]. Hashing, indeed, permits us to find whether an element belongs to a set $S$ with an $O(|S|)$ space complexity and an average time complexity close to a constant.

### Positive constraints

In this case, the set of allowed tuples, denoted by $T(C)$, is explicitly given. Thus, for one constraint, the space

**Algorithm 6:** function `seekNextSupport`

```
seekNextSupport (in C: constraint; in y: variable;
                 in b: value; in dummy): tuple
  while elt(y,b) ≠ nil do
    σ ← Tuple(elt(y,b));
    if isValid(σ) then return σ ;
    elt(y,b) ← next(elt(y,b)) ;
  return nil ;
```

complexity will depend on the size of $T(C)$ similarly to GAC4. So, at first glance, it seems that the space complexity of GAC4 cannot be improved. But, in practice, we may have a problem in which some of the constraints appear many times involving different variables. For instance, in configuration problems, a configurator has to choose the most appropriate components among a catalog of predefined components and arrange them according to constraints that can be assembly rules, performance, etc. Most of the constraints involved with the same kind of components are repeated and are given by their common set of allowed tuples.

For a constraint $C$, the representation of the data structures of GAC4 corresponds to a particular representation of $T(C)$. So, if two constraints are given by $T(C)$, then GAC4 needs two distinct data structures, each with a $O(|T(C)|)$ space complexity. Hence, for $p$ repeated constraints the space complexity of GAC4 will be $O(p \cdot |T(C)|)$. The algorithm we presented only requires an explicit representation of $S$ and $S_C$ lists, and $last_C$ pointers, for each constraint. This leads to a $O(r^2d)$ space complexity per constraint, $r$ being the arity of the constraint involved (see Property 2). If the representation of $T(C)$ can be shared by $p$ repeated constraints, then the global space complexity will be $O(|T(C)| + p.r^2d)$, and a factor $p$ will be gained with regard to GAC4.

Such an algorithm can be defined by representing the common set $T(C)$ as GAC4 does for one constraint and by using for each repeated constraint and each value $(y, b)$ a pointer that indicates the last element reached in $T(C)$. We will denote it by $elt(y, b)$. The algorithm does no longer compute the tuples and test them; it only looks for a support directly in $T(C)$. The new algorithm is given by the function `seekNextSupport` (Alg. 6) which replaces the previous one. The function $next(elt(y, b))$ gives the next tuple in $T(C)$ containing $b$ for variable $y$.

## 5 Space and time analysis

Let $C$ be a constraint of arity $r$ with a tightness $t$ (proportion of forbidden tuples), and assume that all domains have a size $d$. Then, if a pointer is represented by 4 bytes, a good estimation of the memory required is:

- $16 \cdot r \cdot |\{tuples\ on\ X(C)\ allowed\ by\ C\}| = 16rd^r(1-t)$ Bytes for GAC4
- $24r^2d$ Bytes for the GAC-schema ($C$ being a predicate[1])

The table below gives some results about the space requirements of GAC4 and the GAC-schema. The sizes are given in Megabytes. Only the configurations associated with bold numbers can be used in practice. ("–" means that more than 256 Mb are needed.) Bear in mind that these space requirements are given for only **one** constraint.

| $t$ | 0.001 | 0.02 | 0.2 | 0.5 | 0.8 | 0.98 | 0.999 |
|---|---|---|---|---|---|---|---|
| $r=8$, $d=10$ | | | | | | | |
| GAC-s | **0.015** for any tightness | | | | | | |
| GAC4 | - | - | - | - | - | 256 | 12.8 |
| $r=5$, $d=20$ | | | | | | | |
| GAC-s | **0.012** for any tightness | | | | | | |
| GAC4 | 256 | 251 | 205 | 128 | 51 | 5.1 | **0.26** |
| $r=4$, $d=50$ | | | | | | | |
| GAC-s | **0.019** for any tightness | | | | | | |
| GAC4 | - | - | - | 200 | 80 | 8 | **0.4** |

Finally, to be exhaustive on complexities, we report the worst-case time complexities of the different versions of the GAC-schema presented in this paper. For a predicate of arity $r$, the GAC-schema has a time complexity bounded above by $O(d^r)$. This is a gain of $r^2d$ over the $O(r^2d^{r+1})$ of CN. For a negative constraint $C$, this complexity trivially becomes $O(d^r \cdot |\overline{T}(C)|)$ (see subsection 4.2). For a positive constraint $C$, the worst-case time complexity of the GAC-schema is $O(|T(C)|)$. This is the same as the one of GAC4.

## 6 A cryptogram as an example

In this section we briefly present a very small example, which is easy to understand, and which is sufficient to show some of the advantages of dealing with non-binary predicates. (This example is implemented in ILOG SOLVER 4.0, in which the GAC-schema has been inserted. ) For a complete presentation of the example see [ILOG, 1997].

In the cryptogram of Fig. 1, the problem is to find a one to one mapping from letters $\{A, B, D, E, G, L, N, O, R, T\}$ to $\{0..9\}$ in such a way that the addition obtained by replacing each letter by its associated value is consistent. It was already possible to solve this problem with the previous version of ILOG SOLVER, by encoding it (for instance) as the following constraint network: A variable for each letter; A domain containing the numbers 0..9 for each variable; And two non-binary constraints for which efficient algorithms are

[1]For constraints given in extension we should add the size of the constraint representation (see subsection 4.2).

$$
\begin{array}{r}
D\,O\,N\,A\,L\,D \\
+\ G\,E\,R\,A\,L\,D \\
\hline
R\,O\,B\,E\,R\,T
\end{array}
$$

Figure 1: The cryptogram

known, $100000D + 10000O + 1000N + 100A + 10L + D + 100000G + 10000E + 1000R + 100A + 10L + D = 100000R + 10000O + 1000B + 100E + 10R + T$, and all-different$(A, B, D, E, G, L, N, O, R, T)$. However, with this representation, some of the knowledge we have on the problem was impossible to state, while it could have been helpful in solving the problem efficiently. We know for example, that in the right most column $D + D$ is necessarily equal to $T$ or $10 + T$ since ther is no sum kept back. We know that on the left most column $R$ is equal to $D + G$ or $D + G + 1$ since there is no letter on the left of $R$. More generally, for the third column for example, we can state that $(N + R = B) \vee (N + R = 10 + B) \vee (N + R + 1 = B) \vee (N + R + 1 = 10 + B)$. All these constraints (six in the cryptogram of Fig. 1), are predicates, but no already known specific algorithm can handle them (they are not arithmetic relations, but disjunctions of several arithmetic relations).

| | without DVO | | with DVO | |
|---|---|---|---|---|
| | #bt | seconds | #bt | seconds |
| without `GAC-s` | 4612 | 0.72 | 138 | 0.05 |
| with `GAC-s` | 61 | 0.1 | 1 | 0.00 |

The table above presents the number of backtracks (#bt) and cpu-time performances of solving the cryptogram with ILOG SOLVER 4.0 (maintaining arc consistency) on a Pentium Pro 200, with or without the `GAC-schema` and the 6 additional constraints, and with and without the use of the minimal domain dynamic variable ordering (DVO).

Of course it is a very small illustration that is given here with that sample problem. In this problem, indeed, the additional constraints are only added to improve the search. They are not necessary to encode the problem: the first representation was already ensuring that the solutions of the constraint network were the solutions of the cryptogram.

## 7 Conclusion

While arc consistency had been widely studied by the CSP community, there did not exist any algorithm that efficiently achieves arc consistency on non-binary constraints. Thus, we presented `GAC-schema`, which is built to take into account the last improvements available on binary constraints (AC-7 like schema). We saw that it is even more important than on binary constraints to use such improvements (e.g. multidirectionality). With specialized instantiations, this schema is capable of efficiently dealing with predicates, positive constraints, or negative constraints. The perspectives of this work are to propose other instantiations of the schema in order to improve arc consistency processing on some other types of constraint representations frequently occurring in industrial problems.

## References


[Bessière and Régin, 1996] C. Bessière and J.C. Régin. MAC and combined heuristics: two reasons to forsake FC (and CBJ?) on hard problems. In *Proceedings CP'96*, pages 61–75, Cambridge MA, 1996.

[Bessière *et al.*, 1995] C. Bessière, E.C. Freuder, and J.C. Régin. Using inference to reduce arc consistency computation. In *Proceedings IJCAI'95*, pages 592–598, Montréal, Canada, 1995.

[Bessière, 1994] C. Bessière. Arc-consistency and arc-consistency again. *Artificial Intelligence*, 65:179–190, 1994.

[Grant and Smith, 1996] S.A. Grant and B.M. Smith. The phase transition behavior of maintaining arc consistency. In *Proceedings ECAI'96*, pages 175–179, Budapest, Hungary, 1996.

[ILOG, 1997] ILOG. *User's manual.* ILOG SOLVER, 4.0 edition, 1997.

[Mackworth, 1977a] A.K. Mackworth. Consistency in networks of relations. *Artificial Intelligence*, 8:99–118, 1977.

[Mackworth, 1977b] A.K. Mackworth. On reading sketch maps. In *Proceedings IJCAI'77*, pages 598–606, Cambridge MA, 1977.

[Mohr and Henderson, 1986] R. Mohr and T.C. Henderson. Arc and path consistency revisited. *Artificial Intelligence*, 28:225–233, 1986.

[Mohr and Masini, 1988] R. Mohr and G. Masini. Good old discrete relaxation. In *Proceedings ECAI'88*, pages 651–656, Munchen, FRG, 1988.

[Montanari, 1974] U. Montanari. Networks of constraints: Fundamental properties and applications to picture processing. *Information Science*, 7:95–132, 1974.

[Régin, 1994] J.C. Régin. A filtering algorithm for constraints of difference in CSPs. In *Proceedings AAAI'94*, pages 362–367, Seattle WA, 1994.

[Régin, 1996] J.C. Régin. Generalized arc consistency for global cardinality constraint. In *Proceedings AAAI'96*, pages 209–215, Portland OR, 1996.

[Rossi *et al.*, 1990] F. Rossi, C. Petrie, and V. Dhar. On the equivalence of constraint satisfaction problems. In *Proceedings ECAI'90*, pages 550–556, Stockholm, Sweden, 1990.

[Sabin and Freuder, 1994] D. Sabin and E.C. Freuder. Contradicting conventional wisdom in constraint satisfaction. In *Proceedings PPCP'94*, Seattle WA, 1994.

[Sedgewick, 1990] R. Sedgewick. *Algorithms in C.* Addison-Wesley Publishing Company, 1990.

[van Beek and Dechter, 1995] P. van Beek and R. Dechter. On the minimality and global consistency of row-convex constraint networks. *Journal of the ACM*, 42(3):543–561, 1995.

[Van Hentenryck *et al.*, 1992] P. Van Hentenryck, Y. Deville, and C.M. Teng. A generic arc-consistency algorithm and its specializations. *Artificial Intelligence*, 57:291–321, 1992.