Maintaining arc consistency algorithms during the search with an optimal time and space complexity

Jean-Charles Régin

ILOG Sophia Antipolis, Les Taissounières HB2, 1681 route des Dolines, 06560 Valbonne, France, email: regin@ilog.fr

Abstract. In this paper, we present in detail the versions of the arc consistency algorithms for binary constraints based on list of supports and last value when they are maintained during the search for solutions. In other words, we give the explicit codes of MAC-6 and MAC-7 algorithms. Moreover, we present an original way to restore the last values of AC-6 and AC-7 algorithms in order to obtain a MAC version of these algorithms whose space complexity remains in $O(ed)$ while keeping the $O(ed^2)$ time complexity on any branch of the tree search. This result outperforms all previous studies.

1 Introduction

In this paper we focus our attention on binary constraints. For more than twenty years, a lot of algorithms establishing arc consistency (AC algorithms) have been proposed: AC-3 [6], AC-4 [7], AC-5 [12], AC-6 [1], AC-7, AC-Inference, AC-Identical [2], AC-8 [4], AC-2000: [3], AC-2001 (also denoted by AC-3.1 [13]) [3], AC-3_d [11], AC-3.2 and AC-3.3 [5].

The MAC version of an AC algorithm, is the maintain of the algorithm during the search for a solution.

Some MAC versions of AC algorithms are easy, like AC-3 or AC-2000. Some others AC algorithms are much more complex to be maintained during the search. This is mainly the case for algorithms based on the notion of list of support (S-list) and on the notion of last support (last value). These algorithms, like AC-6, AC-7, AC-2001, AC-Inference, involve some data structures that need to be restored after a backtrack. Currently, there is no MAC version of these algorithms capable to keep the optimal time complexity on every branch of the tree search ($O(d^2)$ per constraint, where d is the size of the largest domain), without sacrificing the space complexity. More precisely, the algorithms AC-6, AC-7 and AC-2001 involve data structures that lead to a space complexity of $O(d)$ per constraint, but the MAC versions of these algorithms require to save some modifications of these data structures in order to restart the computations after a backtrack in a way similar as if this backtrack did not happen, and so they keep the same time complexity for any branch of the tree search as for one establishment of arc consistency. These savings have a cost which depends on the depth of the tree search and that are bounded by d. Therefore for these reasons some authors have proposed algorithms having a $O(d \min(n, d))$ space complexity per constraint [8–10], thus the nice space complexity of these AC algorithms is lost for their MAC versions.

In this paper, we propose an original MAC version of the algorithms involving S-list and last data with a space complexity in $O(d)$ per constraint while keeping the optimal time complexity ($O(d^2)$) for any branch of the tree search.

At this moment, our goal is not to propose an algorithm that outperforms MAC-6, MAC-7 or MAC-2001, but to solve an open question. The capability to avoid to need some extra data can also be quite important, for embedded systems for instance, where all the possible memory requirements must be precomputed and reserved.

This paper is organized as follows. First, we recall some definitions of CP and we give a classical backtrack algorithm associated with a propagation mechanism. Then, we give a classical AC algorithm using the S-List and last data structures. Next, we identify the problems of the MAC version of this algorithm, and we propose a new MAC version optimal in time and in space. At last, we conclude.

2 Preliminaries

A finite **constraint network** $\mathcal{N}$ 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 definition domains of $\mathcal{N}$. Indeed, we consider that any constraint network $\mathcal{N}$ can be associated with an initial domain $\mathcal{D}_0$ (containing $\mathcal{D}$), on which constraint definitions were stated.

A **constraint** C on the ordered set of variables $X(C) = (x_{i_1}, \dots, x_{i_r})$ is a subset $T(C)$ 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 $x_{i_1}, \dots, x_{i_r}$. An element of $T(C)$ is called a **tuple on** $X(C)$ and r is the **arity** of C and $\tau[x]$ denotes the value of the variable x in the tuple τ.

A value a for a variable x is often denoted by (x, a). Then, (x, a) is **valid** if $a \in D(x)$, and a tuple is **valid** if all the values it contains are valid. Let C be a constraint. C is **consistent** iff there exists a tuple τ of $T(C)$ which is valid. A value $a \in D(x)$ is **consistent with** C iff $x \notin X(C)$ or there exists a valid tuple τ of $T(C)$ with $a = \tau[x]$ (τ is the called a **support** for (x, a) on C.) A constraint 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.

A **filtering algorithm** associated with a constraint C is an algorithm which may remove some values that are inconsistent with C; and that does not remove any consistent values. If the filtering algorithm removes all the values inconsistent with C we say that it establishes the arc consistency of C.

Propagation is the mechanism that consists of calling the filtering algorithm associated with the constraints involving a variable x each time the domain of this variable is modified. Note that if the domains of the variables are finite, then this process terminates because a domain can be modified only a finite number of times. The set of values that have been removed from the domain of a variable x is called **the delta domain** of x. This set is denoted by $\Delta(x)$ [12].

Algorithm 1: function SEARCHFORSOLUTION

```
PROPAGATION()
    while $\exists y$ such that $\Delta(y) \neq \emptyset$ do
        pick $y$ with $\Delta(y) \neq \emptyset$
        for each constraint $C$ involving $y$ do
            if $\neg$ FILTER($C, x, y, \Delta(y)$) then return false
        RESET($\Delta(y)$)
    return true
SEARCHFORSOLUTION($x, a$)
    ADDCONSTRAINT($x = a$)
    if all variables are instantiated then PRINTSOLUTION()
    else
        if PROPAGATION() then
            do
                $y \leftarrow$ SELECTVARIABLE()
                $b \leftarrow$ SELECTVALUE($y$)
                SEARCHFORSOLUTION($y, b$)
                REMOVEFROMDOMAIN($y, b$)
            while $D(y) \neq \emptyset$ and PROPAGATION()
    RESTORECN()
```

Function PROPAGATION of Algorithm 1 is a possible implementation of this mechanism. The filtering algorithm associated with the constraint C defined on x and y corresponds to Function FILTER$(C, x, y, \Delta(y))$. This function removes the values of $D(x)$ that are not consistent with the constraint in regards to $\Delta(y)$. For a constraint C this function will also be called with the parameters $(C, y, x, \Delta(x))$. We also assume that function RESET$(\Delta(y))$ is available. This function sets $\Delta(y)$ to the empty set. The algorithm we propose is given as example, and some other could be designed. For our purpose, we only suppose that the delta domain is available.

Algorithm 1 also contains a classical search procedure (a backtrack algorithm) which selects a variable, then a value for this variable and call the propagation mechanism. Note that at the end of the recursive function SEARCHFORSOLUTION, Function RESTORECN is called. This function restores the data structures used by the constraint when a backtrack occurs. We assume that Function

SEARCHFORSOLUTION is called first with a dummy variable x and a dummy value a such that the constraint $x = a$ has no effect.

3 Arc consistency algorithms

Consider a constraint C defined on x and y for which we study the consequences of the modifications of the domain of y.

Algorithm 2: An AC algorithm

```
FILTER(in $C, x, y, \Delta(y)$): boolean
    $(x, a) \leftarrow$ FIRSTPENDINGVALUE($C, x, y, \Delta(y)$)
    while $(x, a) \neq nil$ do
        if ¬ EXISTVALIDSUPPORT($C, x, a, y, get\Delta Value(C)$) then
1           REMOVEFROMDOMAIN($x, a$)
            if $D(x) = \emptyset$ then return false
        $(x, a) \leftarrow$ NEXTPENDINGVALUE($C, x, y, \Delta(y), a$)
    return true
```

Definition 1 *We call* **pending values** *w.r.t. the variable y the set of* **valid** *values of a variable x for which a support is sought by an AC algorithm when the consequences of the deletion of the values of a variable y are studied.*

Thanks to this definition, the principles of AC algorithms can be easily expressed: **Check whether there exists a support for every pending value and remove those that have none**.
Algorithm 2 is a possible implementation of this principle.

We can now give the principles of Functions FIRSTPENDINGVALUE, NEXTPENDINGVALUE and of Function EXISTVALIDSUPPORT for AC-6 and AC-7 algorithm. Since we consider a constraint C involving x and y and that y is modified, then the pending values belong only to x.

AC-6: AC-6 was a major step in the understanding of the AC-algorithm principles. AC-6 mixes some principles of AC-3 with some ideas of AC-4. AC-6 can be viewed as a lazy computation of supports. AC-6 introduces the **S-list** data structure: for every value (y, b), the S-list associated with (y, b), denoted by S-list[(y, b)], is the list of values that are currently supported by (y, b). In AC-6 the knowledge of only one support is enough, then a value (x, a) is supported by **only one** value of $D(y)$, so there is only value of $D(y)$ that contains (x, a) in its S-list. Then, the pending values are the valid values contained in the S-lists of the values in $\Delta(y)$. Function EXISTVALIDSUPPORT is an improvement of the AC-3's one, because the checks in the domains are made w.r.t an ordering and are started from the support that just has been lost, which is the delta value containing the current value in its S-list. The space complexity of AC-6 is in

$O(d)$ and its time complexity is in $O(d^2)$.
AC-7: This is an improvement of AC-6. AC-7 exploits the fact that if (x, a) is supported by (y, b) then (y, b) is also supported by (x, a). Thus, when searching for a support for (x, a), AC-7 proposes, first, to search for a valid value in S-list$[(x, a)]$, and every non valid value which is reached is removed from the S-list. We say that the support is sought by **inference**. This idea contradicts an invariant of AC-6: a support found by inference is no longer necessarily the latest checked value in $D(y)$. Therefore, AC-7 introduces explicitly the notion of **latest checked value** by the data **last** associated with every value. AC-7 ensures the property: If last$[(x, a)] = (y, b)$ then there is no support (y, a) in $D(y)$ with $a < b$. If no support is found by inference, then AC-7 uses an improvement of the AC-6's method to find a support in $D(y)$. When we want to know whether (y, b) is a support of (x, a), we can immediately give a negative answer if last$[(y, b)] > (x, a)$, because in this case we know that (x, a) is a not a support of (y, b) and so that (y, b) is not a support for (x, a). The properties on which AC-7 is based are often called bidirectionnalities. Hence, AC-7 is able to save some checks in the domain in regards to AC-6, while keeping the same space and time complexity.

The MAC version of AC-6 needs an explicit representation of the lastest checked value, thus the AC-6 and AC-7 algorithms use the following data structures:
Last: the last value of (x, a) for a constraint C is represented by last$[(x, a)]$ which is equals to a value of y or nil.
S-List: these are classical list data structures.

Algorithm 3: Pending values computation.

```
firstPendingValue($C, x, y, \Delta(y)$): value
    $b$ ←first($\Delta(y)$)
    return traverseS-list($C, x, y, \Delta(y)$)
nextPendingValue($C, x, y, \Delta(y), a$): value
    return traverseS-list($C, x, y, \Delta(y)$)
traverseS-list($C, x, y, \Delta(y)$): value
    while $(y, b) \neq nil$ do
        $(x, a)$ ←seekValidSupportedValue($C, y, b$)
        if $(x, a) \neq nil$ then return $(x, a)$
        $b$ ←next($\Delta(y), b$)
    return nil
get$\Delta$Value($C, x, y, \Delta(y)$): return $b$
```

We can give a MAC version of AC-6 and AC-7 :

Algorithm 3 is a possible implementation of the computation of pending values. Note that some functions require "internal data" (a data whose value is stored). We assume thatFIRST($D(x)$) returns the first value of $D(x)$ and NEXT($D(x), a$) returns the first value of $D(x)$ strictly greater than a.

Function SEEKVALIDSUPPORTEDVALUE(C, x, a) returns a valid supported value belonging to the S-list[(y, b)] (see Algorithm 6.)

Algorithm 4: Function EXISTVALIDSUPPORT

```
EXISTVALIDSUPPORT($C, x, a, y, \delta y$): boolean
if last[$(x, a)$] $\in D(y)$ then $(y, b) \leftarrow$ last[$(x, a)$])
if $AC$-$7$ and $(y, b) = nil$ then
    $(y, b) \leftarrow$ SEEKVALIDSUPPORTEDVALUE($C, x, a$)
if $(y, b) = nil$ then $(y, b) \leftarrow$ SEEKSUPPORT($C, x, a, y$)
UPDATES-LIST($C, x, a, y, \delta y, b$)
return $((y, b) \neq nil)$
```

Algorithm 5: Functions seeking for a valid support

```
SEEKSUPPORT($C, x, a, y$) : value
    $b \leftarrow$NEXT($D(y)$,last[$(x, a)$])
    while $b \neq nil$ do
        if last[$(y, b)$] $\leq (x, a)$ and $((x, a), (y, b)) \in T(C)$ then
            last[$(x, a)$] $\leftarrow (y, b)$
            return $(y, b)$
        $b \leftarrow$NEXT($D(y), b$)
    return nil
```

Algorithm 4 is a possible implementation of Function EXISTVALIDSUPPORT and Algorithm 5 is a possible implementation of Function SEEKSUPPORT.

The S-list representation will be detailed in a specific section, notably because it has been careful designed in order to be maintained during the search.

4 Maintain during the search

The management of the AC algorithms has been studied in detail in [8].

Two types of data structures can be identified for a filtering algorithm (like an AC algorithm for instance) :

• the **external data structures**. These are the data structures from which the constraint of the filtering algorithm is stated, for instance the variables on which the constraint is defined or the list of allowed combinations by the constraint.

• the **internal data structures**. These are the data structures needed by the filtering algorithm. The space complexity of the filtering algorithm is usually based on these data structures. For instance, AC-6 and AC-7 require data structures in $O(d)$.

We will say that the space complexity of a MAC version of an AC algorithm is optimal if it is the same as the space complexity of the AC algorithm.

In this section, we propose a MAC version of an AC algorithm using S-list and/or last value having an optimal space and time complexity.

There is no particular problem when we go down to the search tree, because the instantiation of new variable leads only to the deletion of values. The main difficulty is to manage the data structures when a failure occurs, that is when there is a backtrack.

Consider that n is the current node of the search. The data structures associated with an AC algorithm contain certain values. These values are called the **state** of the data structures. Then, assume that the search is continued from n and then backtracked to n. In this case, two possibilities have been identified [8]:

- the state of the data structure at the node n is exactly restored
- an equivalent state is restored.

4.1 Exact restoration of the state

This method saves the modifications of the state of an AC algorithm in order to restore exactly this state after a backtrack. In other word, every data contains the same value as it had when n was the current node. This implies that every modification of the value of a data has to saved in order to be restored after a backtrack. Every S-list and every last value can be modified d times per constraint during the search. Thus, the space complexity is multiplied by a factor of d. So, this possibility cannot lead to a MAC version with an optimal space complexity.

4.2 Restoration of an equivalent state

This is another notion which is based on the properties that have to be satisfied by the data structures. The algorithms have an optimal time complexity when some properties are satisfied. What is important is not the way they are satisfied, but only the fact that they are satisfied. For some data structures it is not necessary to restore exactly the values it contains before. For instance, if (y, b) was the current support of (x, a) for the node n and if this support changes to become (y, c) then (y, c) can be the current support of (x, a) when all the nodes following n are backtracked. This means that there is no need to change the elements in the S-list, no deletion is needed. It is only required to add some values that have been removed.

This method is much more interesting than the restoration of the exact state. We choose to use it and propose to study how we have to design and how we can manage the S-Lists and the last values in order to restore only an equivalent state, while keeping the optimal time complexity.

5 S-list management

If an equivalent state is accepted after backtracking, then when a support is modified there is no need to save it, because it does not need to be restored. However, in order to keep an optimal time complexity for every branch of the tree search, the MAC version of AC-6 and AC-7 algorithms needs to remove from S-lists the reached values that are not valid. In fact, it is necessary to avoid to consider them several times.

This is Function SEEKVALIDSUPPORTEDVALUE that manages this deletion. So, this function deserves a particular attention.

This function is called during the computation of the pending values or when a support is sought by inference (AC-7). When it is called for a value (x, a) it traverses the S-list of (x, a) until a valid value is found and removes from this S-list the value that are reached and not valid. In the MAC version, a restoration of the S-list is obviously needed. More precisely, if (y, b) is reached when traversing S-list$[(x, a)]$ this means that (x, a) is the current support of (y, b) for this constraint. Thus, if (y, b) is no longer valid when it is reached, then (y, b) is removed from S-list$[(x, a)]$, but after backtracking the node of the tree search that led to the deletion of (y, b) it is necessary to restore (y, b) in S-list$[(x, a)]$, because at this moment (y, b) will be valid and (y, b) needs to have a support. Therefore, when an element is removed from the S-list when traversing it, it is necessary to save this information in order to restore it later.

In order to avoid unnecessary memory consumption we propose to represent the S-list as follows :

- The first element of an S-list of a value (y, b) is denoted by firstS$[C, (y, b)]$ which is equals to a value of x or *nil*.
- The S-lists exploit the fact that for a constraint, each value (x, a) can be in at most one S-list. So, every value (x, a) is associated with a data nextInS$[C, (x, a)]$ which is the next element in the S-list of the support of (x, a). For instance, S-list$[C, (y, b)] = ((x, a), (x, d), (x, e))$ will be represented by : firstS$[C, (y, b)] = (x, a)$; nextInS$[C, (x, a)] = (x, d)$; nextInS$[C, (x, d)] = (x, e)$; nextInS$[C, (x, e)] = nil$. The nextInS data are systematically associated with every value so they are **preallocated**.

The saving/restoration of a support by MAC, can be easily done by adding a data to every value (x, a): restoreSupport$[C, (x, a)]$. This data contains the support of (x, a) if (x, a) has been removed from the S-list of its support; otherwise it contains *nil*. This data will be used to restore (x, a) in the S-list of its support when (x, a) will be restored in its domain. More precisely, assume that (y, b) is the support of (x, a) and that (x, a) has been removed from S-list$[C, (y, b)]$ when searching for a valid support of (y, b) by inference. In this case, the data restoreSupport$[C, (x, a)]$ will be set to (y, b). And, when (x, a) will be restored in the domain of its variable after backtracking, then (x, a) will be added to the S-list of restoreSupport$[C, (x, a)]$. Of course, if restoreSupport$[C, (x, a)]$ is *nil* then nothing happens.

This mechanism of restoration implies that a solver has the capability to perform some operations when a value is restored in its domain. This is not a strong assumption and can be made with all existing solvers.

Another point must also be considered. Function existValidSupport calls Function updateS-list in order to update the S-list when a new valid support is found. Conceptually there is no problem, if a new valid support (y, b) is found for a value (x, a) then (x, a) is added to S-list[(y, b)]. However, before being added to the S-list (x, a) must be removed from the S-list of its current support. This deletion causes some problems of implementation because the S-lists are simple lists and to perform a deletion it is necessary to know the previous element. In order to avoid this problem, we have decided to systematically remove all the reached elements from the S-list. Thus, every element which is considered is the first element of the list and so there is no longer any problem to remove it. If a new support is found then the element can be safely added to a new S-list. If there is no valid support then it will be necessary to restore (x, a) in the S-list of its support. this result can be easily obtained by using the previous mechanism of saving/restoration. Function updateS-list implements that idea. Algorithm

Algorithm 6: Management of Supported Values Lists

```
seekValidSupportedValue($C, x, a$) : value
    while $firstS[C, (x, a)] \neq nil$ do
        $(y, b) \leftarrow$ firstS[$C, (x, a)$]
        if $b \in D(y)$ then return $(y, b)$
        firstS[$C, (x, a)$] $\leftarrow$ nextInS[$C, (y, b)$]
        restoreSupport[$C, (y, b)$] $\leftarrow (x, a)$
    return nil
updateS-list($C, x, a, y, \delta y, b$)
    firstS[$C, (y, \delta y)$] $\leftarrow$ nextInS[$C, (x, a)$]
    if $(y, b) = nil$ then restoreSupport[$C, (x, a)$] $\leftarrow (y, \delta y)$
    else
        nextInS[$C, (x, a)$] $\leftarrow$ firstS[$C, (y, b)$]
        firstS[$C, (y, b)$] $\leftarrow (x, a)$
restoreSupports($C, (x, a)$)
    // the value $a$ is restored in $D(x)$
    $(y, b) \leftarrow$ restoreSupport[$C, (x, a)$]
    if $restoreSupport[C, (x, a)] \neq nil$ then
        // $(x, a)$ is added to the S-list of its support $(y, b)$
        nextInS[$C, (x, a)$] $\leftarrow$ firstS[$C, (y, b)$]
        firstS[$C, (y, b)$] $\leftarrow (x, a)$
        restoreSupport[$C, (x, a)$] $\leftarrow nil$
```

6 gives a possible implementation of the management of S-lists.

6 Last management

The notion of latest checked value (last value) is necessary for AC-6 and AC-7 algorithms to have an $O(d^2)$ time complexity per constraint.

The last value satisfies two properties :

Property 1 *Let* $(y, b) = last[C, (x, a)]$ *then*
$\forall c \in D(y), c < b \Rightarrow ((x, a), (y, c)) \notin T(C)$.

This first property ensures that there is no reason to consider again the values that are less than the last value.

Property 2 *Let* $(y, b) = last[C, (x, a)]$ *then*
Function SEEKSUPPORT *has never checked the compatibility between* (x, a) *and any element* $d \in D(y)$ *with* $d > b$.

This property ensures that no compatibility check has been performed for the values of y greater than the last value.

These two properties ensures that the compatibility between two values will never be checked twice (if the bidirectionnality is not taken into account).

If the last are not restored after backtracking then the time complexity of AC-6 and AC-7 algorithms is in $O(d^3)$. We can prove that claim with the following example. Consider a value (x, a) that has exactly ten supports among the 100 values of y: $(y, 91), (y, 92)..., (y, 100)$; and a node n of the tree search for which the support of (x, a) is $(y, 91)$. If the last value is not restored after a backtrack then there are two possibilities to define a new last value:

1. the new last value is recomputed from the first value of the domain
2. the new last value is defined from the current last value, but the domains are considered as circular domains: the next value of the maximum value is the minimum value.

For the first case, it is clear that all the values strictly less than $(y, 91)$ will have to be reconsidered after every backtrack for computing a valid support.
For the second possibility, we can imagine an example for which before backtracking $(y, 100)$ is the current support; then after the backtrack and since the domains are considered as circular domains it will be necessary to reconsidered again all the values that are strictly less than $(y, 91)$.

Note also, that if the last value is not correctly restored it is no longer possible to totally exploit the bidirectionnality. So, it is necessary to correctly restored the last value.

6.1 Saving-Restoration of Last

The simplest way is to save the current value of last each time it is modified and then to restore these values after a backtrack. This method can be improved by remarking that it is sufficient to save only the first modification of the last value for a given node of the tree search. In this case, the space complexity of AC-6 and AC-7 algorithms is multiplied by $\min(n, d)$ where n is the maximum of the tree search depth [8–10].

6.2 Recomputation of last

First, with the restoration of an equivalent state, it is necessary to slightly modify the algorithm. The last value of (x, a) can indeed be valid and not be the current support of (x, a), because the current support has been found by inference and no support is restored. Thus, it is necessary to check the validity of the last value in the MAC version of an AC algorithm.

Now, we propose an original method to restore the correct last value. Instead of being based on savings this method is based on recomputation.

The idea of this algorithm is quite simple. Assume that we backtrack from a node n, and consider a variable x. We will denote by $D_R(y)$ the values of the variable y that have to be restored during this backtrack. Then, we have the following proposition on which our algorithm is based:

Proposition 1 *Let (x, a) be a value, and $T(C,(x, a), D_R(y))$ be the set of values of $D_R(y)$ that are compatible with (x, a) w.r.t C. Then* $\min(\mathit{last}[C,(x, a)], \min(T(C,(x, a), D_R(y))))$ *is a possible value of* $\mathit{last}[C,(x, a)]$.

proof: To prove this proposition we need to prove that $\min(\text{last}[C,(x, a)], \min(T(C,(x, a), D_R(y))))$ satisfies both Property 1 and Property 2:

• Property 1:

Let $D_A(y)$ be the domain of the variable y after the backtrack, and $D_B(y)$ be the domain of the variable y before the backtrack. Since the algorithm systematically checked if the last value is valid, then after the backtrack the last value of (x, a) will be the minimum between its current value and the value $b \in D_A(y)$ such that (x, a) and (y, b) are compatible and there is no value $c \in D_A(y)$ with $c < b$ that is compatible with (x, a). We have $D_A(y) = D_B(y) \cup D_R(y)$. Then before the restoration either $\text{last}[C,(x, a)] \in D_B(y)$ or $\text{last}[C,(x, a)] \in D_R(y)$ or $\text{last}[C,(x, a)] \notin D_A(y)$. Moreover, if $\text{last}[C,(x, a)] \in D_B(y)$ before the restoration, then it does not exist another value $c \in D_B(y)$ compatible with (x, a) and such that $(y, c) < \text{last}[C,(x, a)]$ by definition of last. Hence, to compute the minimum, it is enough to compare $\text{last}[C,(x, a)]$ with only the compatible values of $D_R(y)$, and Property 1 is satisfied.

• Property 2:

The last value of (x, a) can only be equal to either nil or a value of y which is compatible with (x, a), by definition. Then, $\min(\text{last}[C,(x, a)], \min(T(C,(x, a), D_R(y))))$ is either equal to $\text{last}[C,(x, a)]$ or equal to $\min(T(C,(x, a), D_R(y)))$. The first case means that the last value has not been modified after node n then Property 2 is satisfied from the branch of the tree search going from the root to the node n after the backtrack to node n. The second case means that the smallest compatible value b of $D(y)$ after the backtrack to node n is the new last value. Suppose that Function seekSupport has reached a value c of y when seeking for a new support for the value (x, a) in the branch of the tree search going from the root to node n. At the node n the value a belongs to the domain of y, so it means that a support d greater than c has been found by Function seekSupport and that a last value greater than c exists. Since $b = \min(T(C,(x, a), D_R(y)))$ is the smallest valid value of y compatible with (x,a), the value d with $d > b$ cannot be a last value, so it is impossible to find such a value and Property 2 holds. $\odot$

Only the values of $D(x) \cup D_R(x)$ needs to have their last value restored. So, we obtain the new algorithm (see Algorithm 7.) Function RECOMPUTELAST is called for every variable of every constraint after every backtrack.

Algorithm 7: Restoration of last by recomputation

```
RECOMPUTELAST$(C, x)$
for each $a \in (D(x) \cup D_R(x))$ do
    for each $b \in D_R(y)$ do
        if $((x,a),(y,b)) \in T(C)$ then
            last$[C,(x,a)] \leftarrow$ min(last$[C,(x,a)],(y,b))$
```

It is important to note that it is possible to recompute a value of last which is greater than the last value that would had been restored by using the saving/restoration mecanism. So, during the backtrack to the node n we can benefit from the computations that have been made after the node n.

Let us study the time complexity for any branch of the tree search, that is when we backtrack from a leaf to the root.
For one restoration and for one variable of a constraint the time complexity of Function RECOMPUTELAST is in $O(|D(x)| \times |D_R(y)|)$.
For one branch of the tree search the time complexity of the calls of Function 7 is in $O(\sum_{D_{Ri}} |D_0(x)| \times |D_{Ri}(y)|) = O(|D_0(x)| \times \sum_{D_{Ri}} |D_{Ri}(y)|)$. Moreover, the set $D_{Ri}(y)$ are pairwise disjoint for one branch of the tree search and their union is included in $D(y)$. Therefore we have $\sum_{D_{Ri}} |D_{Ri}(y)| \leq |D_0(y)|$ and the time complexity is in $O(|D_0(x)| \times |D_0(y)|) = O(d^2)$ per constraint. So, we have the same time complexity as for AC-6 or AC-7 algorithms.

It is possible to give some improvements of the previous algorithm :

First, if the values of $D_R(y)$ are ordered then the number of tests can be limited, because the first compatible value which is less that the last value will become the new last value. If the complexity of one sort is in $d\log(d)$ then the time complexity of all the sorts for one branch of the tree search will be depends on $\sum_{i=k..1} |D_{Ri}(y)| \log(|D_{Ri}(y)|) \leq \sum_{i=k..1} |D_{Ri}(y)| \log(d) \leq \log(d) \sum_{i=k..1} |D_{Ri}(y)| \leq d\log(d)$. This number is less than d^2.

Then, we can separate the study of the values of x. There are several possibilities (note that $D(x)$ and $D_R(x)$ are disjoint) :

- $(x,a) \in D_R(x)$. It is possible to use a new data that stores the first value of a last for the current node (that is only one data is introduced per value). This new data saves the last value that have to be restored for the values that are removed by the current node. So, the last of these values can be restored in $O(1)$ per value.

– $(x, a) \in D(x)$. In this case there are two possible cases :
  - last$[C,(x,a)] \notin D(y)$ and last$[C,(x,a)] \notin D_R(y)$. In this case the last is correct and no restoration is needed.
  - last$[C,(x,a)] \in D(y)$ or last$[(x,a)] \in D_R(y)$. There is no specific improvement : the systematic checks seem to be needed.

7 Conclusion

In this paper we have presented MAC versions of AC-6 and AC-7 algorithms. We have also given a way to restore the latest checked value that lead to MAC-6 and MAC-7 algorithms having the same space complexity as AC-6 and AC-7. This result improves all the previous studies.

References

1. C. Bessière. Arc-consistency and arc-consistency again. *Artificial Intelligence*, 65(1):179–190, 1994.
2. C. Bessière, E.C. Freuder, and J-C. Régin. Using constraint metaknowledge to reduce arc consistency computation. *Artificial Intelligence*, 107(1):125–148, 1999.
3. C. Bessière and J-C. Régin. Refining the basic constraint propagation algorithm. In *Proceedings of IJCAI'01*, pages 309–315, Seattle, WA, USA, 2001.
4. A. Chmeiss and P. Jégou. Efficient path-consistency propagation. *International Journal on Artificial Intelligence Tools*, 7(2):79–89, 1998.
5. C. Lecoutre, F. Boussemart, and F. Hemery. Exploiting multidirectionnality in coarse-grained arc consistency algorithm. In *Proceedings CP'03*, pages 480–494, Cork, Ireland, 2003.
6. A.K. Mackworth. Consistency in networks of relations. *Artificial Intelligence*, 8:99–118, 1977.
7. R. Mohr and T.C. Henderson. Arc and path consistency revisited. *Artificial Intelligence*, 28:225–233, 1986.
8. J-C. Régin. *Développement d'outils algorithmiques pour l'Intelligence Artificielle. Application à la chimie organique.* PhD thesis, Université de Montpellier II, 1995.
9. M. van Dongen. Lightweight arc consistency algorithms. Technical Report TR-01-2003, Cork Constraint Computation Center, CS Department, University College Cork, Western Road, Cork, 2003.
10. M. van Dongen. Lightweight mac algorithms. Technical Report TR-02-2003, Cork Constraint Computation Center, CS Department, University College Cork, Western Road, Cork, 2003.
11. M.R. van Dongen. Ac-3d an efficient arc-consistency algorithm with a low space-complexity. In *Proceedings CP'02*, pages 755–760, Ithaca, NY, USA, 2002.
12. P. Van Hentenryck, Y. Deville, and C.M. Teng. A generic arc-consistency algorithm and its specializations. *Artificial Intelligence*, 57:291–321, 1992.
13. Y. Zhang and R. Yap. Making ac-3 an optimal algorithm. In *Proceedings of IJCAI'01*, pages 316–321, Seattle, WA, USA, 2001.