Maintaining Arc Consistency Algorithms During the Search Without Additional Space Cost

Conference Paper *in* Lecture Notes in Computer Science · October 2005

DOI: 10.1007/11564751_39 · Source: DBLP

1 author:

Jean-Charles Régin
University of Nice Sophia Antipolis
116 PUBLICATIONS **4,001** CITATIONS

Maintaining arc consistency algorithms during the search without additional space cost

Jean-Charles Régin

Computing and Information Science, Cornell University, Ithaca NY 14850 USA
jcregin@cs.cornell.edu

Abstract. In this paper, we 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, where d is the size of the largest domain and e is the number of constraints. This result outperforms all previous studies.

1 Introduction

In this paper we focus our attention on binary constraints. The MAC version of an algorithm establishing arc consistency (AC algorithm), is the maintaining of this AC algorithm during the search for a solution. For more than twenty years, a lot of 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 [9], AC-3.2 and AC-3.3 [5]. Some AC algorithms, like AC-3 or AC-2000, are easy to maintain during the search whereas some others are much more complex. This is mainly the case for algorithms based on the concept of list of support (S-list) and on the concept of last support (last value). These algorithms, like AC-6, AC-7, or AC-2001, 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 is bounded by d. Therefore some authors have proposed algorithms having a $O(d \min(n, d))$ space complexity per constraint [8,

10, 11], where n is the number of variables. 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-lists and last values 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 main goal is to close this open question and not to propose an algorithm outperforming MAC-6 or MAC-7.

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 value data structures. Next, we identify the problems of the MAC version of this algorithm, and we propose a new MAC version without additional cost. At last, we conclude.

2 Preliminaries

A finite **constraint network** $\mathcal{N}$ is defined as a set of n **variables** $X = \{x_1, \ldots, x_n\}$, a set of current **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 **constraint** C on the ordered set of variables $X(C) = (x_{i_1}, \ldots, x_{i_r})$ is a subset $T(C)$ of the Cartesian product $D(x_{i_1}) \times \cdots \times D(x_{i_r})$ that specifies the **allowed** combinations of values for the variables $x_{i_1}, \ldots, x_{i_r}$. An element of $D(x_{i_1}) \times \cdots \times D(x_{i_r})$ is called a **tuple on** $X(C)$ and $\tau[x]$ is the value of τ assigned to x. A value a for a variable x is often denoted by (x, a). (x, a) is **valid** iff $a \in D(x)$. Let C be a constraint. A tuple τ on $X(C)$ is **valid** iff $\forall x \in X(C), \tau[x] \in D(x)$; and τ is a **support** for (x, a) iff $\tau[x] = a$ and $\tau \in T(C)$. C is **consistent** iff $\exists \tau \in T(C)$ which is valid. A value $a \in D(x)$ is **consistent with** C iff $x \notin X(C)$ or there exists a valid support for (x, a). A constraint is **arc consistent** iff $\forall x \in X(C), D(x) \neq \emptyset$ and $\forall a \in D(x)$, 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 then we say that it establishes arc consistency of C and that it is an AC algorithm.

The delta domain of a variable x with respect to a filtering algorithm F associated with a constraint C is the set of values that have been removed from the domain of x between two successive calls of F. It is denoted by $\Delta(x, F)$ and a value in $\Delta(x, F)$ is called a **delta value**. More information about delta domains can be found in [12] or in the manual of ILOG Solver. Note that the conjunction of the elements in the delta domains form the *waitingList* of AC-6 and AC-7 algorithms. Sometimes, all the filtering algorithms depending on the modifications of $D(x)$ are successively considered after the modifications of x. Thus, if there is no side effect (that is x is not modified by these filtering algorithms), then all these filtering algorithms can share the same delta domain for x, and only one representation is needed. Therefore, there is no need to

specify the filtering algorithm for identifying the delta domain associated with a filtering algorithm and the delta domain of x is denoted by $\Delta(x)$.

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. If every filtering algorithm associated with every constraint establishes arc consistency then we say that the propagation mechanism establishes arc consistency of the network.

Algorithm 1: function searchForSolution

```
searchForSolution(x, a)
    addConstraint(x = a)
    if all variables are instantiated then printSolution()
    else
        if propagation() then
            do
                y ← selectVariable()
                b ← selectValue(y)
                searchForSolution(y, b)
                removeFromDomain(y, b)
            while D(y) ≠ ∅ and propagation()
    restoreCN()
propagation()
    while ∃y such that Δ(y) ≠ ∅ do
        pick y with Δ(y) ≠ ∅
        for each constraint C involving y do
            (D(x), Δ(x)) ← filter(C, x, y, Δ(y))
            if D(x) = ∅ then return false
        Δ(y) ← ∅
    return true
```

Function propagation of Algorithm 1 is a possible implementation of this mechanism when delta domains are shared.
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 some values of $D(x)$ that are not consistent with the constraint with respect to $\Delta(y)$. It also updates the delta domain of x. For a constraint C this function will also be called with the parameters $(C, y, x, \Delta(x))$.

Algorithm 1 also contains a classical recursive search procedure which selects a variable, then a value for this variable and call the propagation mechanism. Function restoreCN 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

We will consider a constraint C defined on x and y for which we study the consequences of the modification of $D(y)$. For the sake of clarity, we will avoid adding systematically C as a parameter of every data structure. For instance, we will denote by data$[(x,a)]$ a value linked to (x,a) instead of data$[C,(x,a)]$.

Definition 1 *Let x be a variable and F be a filtering algorithm aiming to remove some values of $D(x)$. We call* **pending values** *of x w.r.t. F, the set of* **valid** *values of x for which a valid support is sought by F when F is called.*

Thanks to this definition, and if every value which is not a pending value is consistent with the constraint then the principles of AC algorithms can be easily expressed:
Check whether there exists a valid support for every pending value and remove those that have none.

Algorithm 2: An AC algorithm

```
FILTER(in C, x, y, Δ(y)): (domain,delta domain)
    (x, a) ← FIRSTPENDINGVALUE(C, x, y, Δ(y))
    while (x, a) ≠ nil do
        if ¬ EXISTVALIDSUPPORT(C, x, a, y, GETΔVALUE(C)) then
            D(x) ← D(x) − {a}
            Δ(x) ← Δ(x) ∪ {a}
            if D(x) = ∅ then return (∅, Δ(x))
        (x, a) ← NEXTPENDINGVALUE(C, x, y, Δ(y), a)
    return (D(x), Δ(x))
```

Algorithm 2 is a possible implementation of this principle. Functions FIRSTPENDINGVALUE and NEXTPENDINGVALUE identify the pending values, and Function EXISTVALIDSUPPORT searches for a valid support for these pending values. We can now detail the principles of these functions for AC-6 and AC-7 algorithms.

AC-6: 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). A value (x,a) is supported by **only one** value of $D(y)$, so there is at most one 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 $\Delta(y)$. If a value of $D(x)$ is not a pending value then it means that its support has not been deleted so the value is consistent with the constraint. The search for a valid support (Function EXISTVALIDSUPPORT) is done by checking in $D(y)$ whether there is a support for (x,a). These checks 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 $O(d)$ because a value belongs to at most one S-list and its time complexity is in $O(d^2)$, because a value can be a pending value only d times and the search for a valid support is done in $O(d)$ globally.

AC-7: AC-7 improves AC-6 by exploiting 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 valid 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 valid support is sought by **inference**. This idea contradicts an invariant of AC-6: a valid support found by inference is no longer necessarily the latest checked value in $D(y)$. Therefore, AC-7 introduces explicitly the concept of **last 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, c) in $D(y)$ with $c < b$. 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 property on which AC-7 is based are often called bidirectionality. Hence, AC-7 is able to save some checks in the domain in regard to AC-6, while keeping the same space and time complexity.

The MAC version of AC-6 needs an explicit representation of the latest 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 equal to a value of y or nil.
S-List: these are classical list data structures.

Algorithm 3: Pending values computation.

```
// b is an internal data
firstPendingValue(C, x, y, Δ(y)): value
    b ←first(Δ(y))
    return traverseS-list(C, x, y, Δ(y))
nextPendingValue(C, x, y, Δ(y), a): value
    return traverseS-list(C, x, y, Δ(y))
traverseS-list(C, x, y, Δ(y)): value
    while (y, b) ≠ nil do
        (x, a) ←seekValidSupportedValue(C, y, b)
        if (x, a) ≠ nil then return (x, a)
        b ←next(Δ(y), b)
    return nil
getΔValue(C, x, y, Δ(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. We assume that FIRST($D(x)$) 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 5.) Algorithm 4 gives a possible implementation of Function EXISTVALIDSUPPORT and Function SEEKVALIDSUPPORT.

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

Algorithm 4: Search for a valid support

```
existValidSupport(C, x, a, y, δy): boolean
    if last[(x, a)] ∈ D(y) then (y, b) ← last[(x, a)])
    if AC-7 and (y, b) = nil then
        (y, b) ← seekValidSupportedValue(C, x, a)
    if (y, b) = nil then
        (y, b) ← seekValidSupport(C, x, a, y)
    updateS-list(C, x, a, y, δy, b)
    return ((y, b) ≠ nil)
seekValidSupport(C, x, a, y): value
    b ←next(D(y),last[(x, a)])
    while b ≠ nil do
        if last[(y, b)] ≤ (x, a) and ((x, a), (y, b)) ∈ T(C) then
            last[(x, a)] ← (y, b)
            return (y, b)
        b ←next(D(y), b)
    return nil
```

4 Maintenance during the search

In this section, we propose a MAC version of an AC algorithm using S-lists and/or last values having the same space and time complexity as the AC algorithm.

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 combinations allowed 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 usu-

ally based on these data structures. For instance, AC-6 and AC-7 require data structures in $O(d)$.

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

Consider that N is the current node of the search. The internal data structures associated with an AC algorithm contain certain values. These values are called the **state** of the internal 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 internal 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 words, every data contains the same value as it had when N was the current node. This implies that every modification of a data has to be 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 without additional cost.

4.2 Restoration of an equivalent state

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 they contained 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 when backtracking. It is only required to add some values that have been removed.

We propose, in the next sections, to study how the S-Lists and the last values can be managed in order to restore only an equivalent state.

5 S-list management

If an equivalent state is accepted after backtracking, then there is no need to save the modifications of supports, because they do 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 needs to remove from S-lists

the values that are traversed and that are not valid to avoid considering them several times.

Function seekValidSupportedValue manages this deletion. So, this function deserves a particular attention.

This function is called during the computation of the pending values or when a valid 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 non valid values that are reached. In the MAC version, a specific restoration of the S-list is needed. More precisely, if (y, b) is reached when traversing S-list$[(x, a)]$ then (x, a) is the current valid support for (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 valid 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$[(y, b)]$ which is equal 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$[(x, a)]$ which is the next element in the S-list of the support of (x, a). For instance, S-list$[(y, b)] = ((x, a), (x, d), (x, e))$ will be represented by : firstS$[(y, b)] = (x, a)$; nextInS$[(x, a)] = (x, d)$; nextInS$[(x, d)] = (x, e)$; nextInS$[(x, e)] = nil$. 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$[(x, a)]$. 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 a will be restored in $D(x)$. More precisely, assume that (y, b) is the support of (x, a) and that (x, a) has been removed from S-list$[(y, b)]$ when searching for a valid support of (y, b) by inference. Then, the data restoreSupport$[(x, a)]$ 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$[(x, a)]$. Of course, if restoreSupport$[(x, a)]$ is nil then nothing happens.

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 a 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 unidirectional 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. Function updateS-list implements that idea and Algorithm 5 gives a possible implementation of the management of S-lists.

Algorithm 5: Management of Supported Values Lists

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

6 Last management

First, when an equivalent state is restored, it is necessary to slightly modify the AC algorithm. The last value of (x, a) can indeed be valid and not be the current support of (x, a), because the supports are not systematically restored. Thus, it is necessary to check the validity of the last value in the MAC version of an AC algorithm (See first line of Function existValidSupport.)

The concept of last value is necessary for AC-6 and AC-7 algorithms to have an $O(d^2)$ time complexity per constraint. Any last value satisfies the following property:

Property 1 *Let* $(y, b) = last[(x, a)]$ *then*

(i) $\forall c \in D(y), c < b \Rightarrow ((x, a), (y, c)) \notin T(C)$.

(ii) *Function* seekValidSupport *has never checked the compatibility between* (x, a) *and any element* $d \in D(y)$ *with* $d > b$.

This property ensures that the compatibility between two values will never be checked twice.

If the last values 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 valid 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:

- the new last value is recomputed from the first value of the domain
- 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 bidirectionality. So, it is necessary to correctly restore the last values.

6.1 Saving-Restoration of last values

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, 11].

6.2 Recomputation of last values

We propose an original method to restore the correct last value. Instead of being based on savings this method is based on recomputations.

Consider a node $N + 1$ of the tree search obtained from a node N. We will denote by $D_N(y)$ (resp. $D_{N+1}(y)$) the domain of the variable y at node N (resp. $N+1$). Then, we have the following proposition on which our algorithm is based:

Proposition 1 *Let $D_R(y) = D_N(y) - D_{N+1}(y)$ and $K(x, a)$ be the set of values of y that are compatible with (x, a) w.r.t. C. If last$[(x, a)]$ satisfies Property 1 for node $N + 1$ then $\min($last$[(x, a)], \min(K(x, a) \cap D_R(y))$ is a possible value of last$[(x, a)]$ for node N.*

proof: It is sufficient to prove that $\min(\text{last}[(x, a)], \min(K(x, a) \cap D_R(y)))$ satisfies Property 1:

(i) : Let b =last$[(x,a)]$. We have $D_N(y) = D_{N+1}(y) \cup D_R(y)$ and by Property 1.(i) $\forall c \in D_{N+1}(y), c < b \Rightarrow ((x,a),(y,c)) \notin T(C)$. Thus, if $\forall c \in D_R(y), c < b \Rightarrow ((x,a),(y,c)) \notin T(C)$ then last$[(x,a)]$ also satisfies Property 1.(i). On the other hand, if $\exists c \in D_R(y)$ with $c < b$ and $((x,a),(y,c)) \in T(C)$ then $d = min(K(x,a) \cap D_R(y))$ satisfies Property 1.(i).

(ii) If $\min($last$[(x,a)], \min(K(x,a) \cap D_R(y))$ =last$[(x,a)]$ then last$[(x,a)]$ has the same value for node $N+1$ and for node N. Property 1.(ii) is satisfied by last$[(x,a)]$ for node $N+1$ therefore it is also satisfied by last$[(x,a)]$ for node N.
If $\min($last$[(x,a)], \min(K(x,a) \cap D_R(y)) = \min(K(x,a) \cap D_R(y)) = d$ then suppose that Function seekValidSupport has reached a value $c > d$ 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. The value a is still in $D_N(x)$, so it means that Function seekValidSupport found a valid support for (x,a) which is greater than c and so greater than d. Since d is a support and $d \in D_N(y)$ this is not possible. So, Property 1.(ii) holds. $\odot$

This proposition is used to restore the last values for the node N, that is when the node $N+1$ is backtracked.

Example: Consider the following last values for node N: last$[(x,0)] = (y,0)$, last$[(x,1)] = (y,1)$, last$[(x,2)] = (y,2)$; and the following domains $D_N(x) = \{0,1,2\}$ and $D_N(y) = \{0,1,3..,50\}$. Now, suppose that the last values for node $N+1$ are last$[(x,0)] = (y,0)$, last$[(x,2)] = (y,20)$, last$[(x,3)] = (y,20)$; and the domains are $D_{N+1}(x) = \{0,1\}$ and $D_{N+1}(y) = \{0,3..20,30..50\}$, so $D_R(y) = \{1,2,21..29\}$. From Proposition 1, when node $N+1$ is backtracked the last values will be recomputed as follows: last$[(x,0)] = \min((y,0), \min(K(x,0) \cap D_R(y)) = (y,0)$, because 0 is less than any value of $D_R(y)$. Then last$[(x,1)]$ is equal to $\min((y,20), \min(K(x,1) \cap D_R(y))$ and value $(y,1)$ is the minimum so last$[(x,1)] = (y,1)$. For last$[(x,2)]$ an interesting result is obtained: $\min($last$[(x,2)], \min(K(x,2) \cap D_R(y)) = (y,20)$ because the values less than 2 are not possible (in node N we had last$[(x,2)] = (y,2)$) and $2 \notin D_N(y)$ and all the values less than 20 have been negatively checked (because they are still in the domains of $D_{N+1}(y)$).

This example shows that the values that are restored by recomputation can be different from the ones that would had been restored by savings. In fact they are greater than or equal to the ones restored by savings. So, **the backtrack to the node N may benefit from the computations that have been performed after the node N**. Therefore, a lot of computations can be avoided. For our example, for $(x,2)$ the values of $D(y)$ less than 20 will no longer be considered for the next subtrees whose root is N.

Algorithm: Only the values of $D(x) \cup D_R(x)$ needs to have their last value restored (See Algorithm 6.) Function recomputeLast is called for every variable of every constraint after every backtrack.

Algorithm 6: Restoration of last values by recomputation

```
RECOMPUTELAST(C, x)
for each a ∈ (D(x) ∪ D_R(x)) do
    for each b ∈ D_R(y) do
        if ((x, a), (y, b)) ∈ T(C) then
            last[(x, a)] ← min(last[(x, a)], (y, b))
```

Time complexity of Function RECOMPUTELAST**:** For one restoration and for one variable of a constraint its time complexity is in $O(|D(x)| \times |D_R(y)|)$. Thus, for one branch of the tree search its time complexity is in $O(\sum_i |D_0(x)| \times |D_{Ri}(y)|) = O(|D_0(x)| \times \sum_i |D_{Ri}(y)|)$. 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 $\sum_i |D_{Ri}(y)| \leq |D_0(y)|$ and the time complexity is in $O(|D_0(x)| \times |D_0(y)|) = O(d^2)$ per constraint, that is the same time complexity as for AC-6 or AC-7 algorithms.

6.3 Improvements:

The previous algorithm needs to be improved to be efficient in practice.

Improvement of Function RECOMPUTELAST:

• the values of $D_R(y)$ can be ordered to reduce the number of tests. 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 equal to $\sum_i |D_{Ri}(y)| \log(|D_{Ri}(y)|) \leq \sum_i |D_{Ri}(y)| \log(d) \leq \log(d) \sum_i |D_{Ri}(y)| \leq d \log(d) \leq d^2$.

• if $(x, a) \in D_R(x)$, then a new data storing the first value of a last for the current node (that is only one data is introduced per value) can be used. This new data saves the last value that has 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.

• if $(x, a) \in D(x)$ and $\text{last}[(x, a)] \notin D(y)$ and $\text{last}[(x, a)] \notin D_R(y)$ then the last is correct and no restoration is needed. In order to avoid considering these values, a global list LM of values of variables is associated with every constraint. If $\text{last}[(x, a)]$ is modified by Function SEEKVALIDSUPPORT or by Function RECOMPUTELAST then (x, a) is added to the head of LM. If (x, a) was already in the list then it is first removed from it before adding it again (that is the ordering of the list is changed). In addition an added value is marked with the current node of the tree search. Then, when node N is backtracked only the values of the list associated with node N have to be considered. These values are at the beginning of the LM list. This method does not change the space complexity, because a value can be in the LM list of a constraint at most once.

Reduction of the number of studied constraints: when a node N is backtracked, it is useless to call Function RECOMPUTELAST for constraints that

have not been propagated at node N. A constraint can be propagated at most d times, because to be propagated at least one value must have been removed. Thus, we can associate with every node of the tree search the list of the constraints that have been propagated for this node without changing the space complexity. Then when node N is backtracked, Function RECOMPUTELAST is called only for the constraints belonging to the list of propagated constraints associated with N.

7 Discussion and Experiments:

We tested our algorithm on different kinds of problems and compared it to a MAC version explicitly saving the last values. For some problems like n-queens there is almost no difference, but for some other problems like the RLFAPs and mainly instance 11, the improvements we give are really worthwhile. Without these improvements a factor of 15 is observed, but with these improvements the new algorithm is slower by only a factor of 2 and this is the worst result we have observed. For a lot of instances only 30% is lost. At last, this new algorithm always performs better than MAC-3, MAC-4 and MAC-2001. In addition, there is still room for improvement.

This method could also be worthwhile for the implementation of non binary constraints. Non binary constraints imply the explicit creation of tuples, whereas with binary constraints we can only work with domains. Moreover, a tuple can support several values. So, the memory management of the creation/deletion of tuples is difficult. Our method should lead to simpler algorithms.

8 Conclusion

In this paper we have presented MAC versions of AC-6 and AC-7 algorithms. We have also given a new method to restore the last 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 and closes an open question. In addition this work can be seen as a step toward a better understanding of AC algorithms and could lead to new improvements to existing algorithms. For instance, we have given an example for which our method saves a lot of checks in regards to the classical MAC algorithm.

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. *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.R. van Dongen. Ac-3d an efficient arc-consistency algorithm with a low space-complexity. In *Proc. CP'02*, pages 755–760, Ithaca, NY, USA, 2002.
10. M.R. van Dongen. Lightweight arc consistency algorithms. Technical Report TR-01-2003, Cork Constraint Computation Center, University College Cork, 2003.
11. M.R. van Dongen. Lightweight mac algorithms. Technical Report TR-02-2003, Cork Constraint Computation Center, University College Cork, 2003.
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.