CAC: A Configurable, Generic and Adaptive Arc Consistency Algorithm

Jean-Charles Régin
regin@ilog.fr

ILOG, 1681, route des Dolines, 06560 Valbonne, FRANCE

Abstract. In this paper, we present CAC, a new configurable, generic and adaptive algorithm for establishing arc consistency for binary constraints. CAC is configurable, that is by combining some parameters CAC corresponds to any existing AC algorithm: AC-3, AC-4, AC-6, AC-7, AC-2000, AC-2001, AC-8, AC-3_d, AC-3.2 and AC-3.3. CAC is generic, like AC-5, because it may takes into account the structure of the constraints. CAC is adaptive because the underlined algorithm can be changed during the computation in order to use the most efficient one. This new algorithm leads to a new nomenclature of the AC algorithms which is based on the different features used by the algorithm like the values that are reconsidered when a domain is modified, or the fact that bidirectionnality is taken into account, or the way a new support is sought. This new nomenclature shows that several new possible combinations are now possible. In other words, we can easily combine some ideas of AC-3 with some ideas of AC-7 and some ideas of AC-2001 with some ideas of AC-6. Some experimental results highlight the advantages of our approach.

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 [9], AC-6 [1], AC-7, AC-Inference, AC-Identical [2], AC-8 [4], AC-2000: [3], AC-2001 (also denoted by AC-3.1 [10]) [3], AC-3_d [8], AC-3.2 and AC-3.3 [5]. Unfortunately, these algorithms are differently described and their comparison is not easy. Thus, we propose a configurable, generic and adaptive AC algorithm, called CAC.

Configurable means that the previous existing algorithms can be represented by setting some predefined parameters. This has some advantages:

• this unique algorithm can represent all known algorithms.

• it clearly shows the differences between all the algorithms. This extends the discussion started in [3].

• some new arc consistency algorithms can be easily and quickly derived from CAC, because some combinations of parameters have never been tested.

• CAC leads to a new nomenclature which is much more explicit than the current one ("AC-" followed by a number.), because algorithms are now ex-

pressed by combinations of predefined parameters. For instance, AC-3 is renamed: CAC-pvD-sD and AC-6 becomes CAC-pvΔs-last-sD.

Generic means that CAC is also a framework that can be derived to take into account some specificity of some binary constraints. In other word, dedicated algorithms can be written, for functional constraints for instance. This corresponds to a part of the generic aspects of AC-5. In our case, the incremental behavior of the AC-5 is generalized.

Adaptive means that CAC is able to use different algorithms successively as suggested in [3]. For instance, AC-2001 can be used then AC-7 and then AC-2001 depending on which one seems to be the best for the current configuration of domains and delta domains. We think, indeed, that **CP will be strongly improved if a filtering algorithm is in itself capable to select at each time its best version, instead of asking the user to do it a priori**.

AC-algorithms work in two steps. First, an initialization step is called. This step consists of finding a support (i.e. a compatible value) for each value. If a value has no support then it is removed from its domain. Then, in the second step the consequences of the deletion of a value are studied, that is a new support is sought for some values. In this paper, we will consider only the second step which is the most important.

This paper is organized as follows. First, we recall some definitions of CP. Then, we study all the existing algorithms, and we identify different concepts of the AC algorithms and detail CAC algorithm. A new nomenclature is proposed. Then, the adaptive behavior of CAC algorithm is considered. At last, after studying some experiments, 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 $T(C)$ is called a **tuple on** $X(C)$ 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). (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 $a\ \tau[x]$. 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. 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 $\Delta(x)$. More information about delta domains can be found in [9]. 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 others could be designed.

Algorithm 1: function PROPAGATION

```
PROPAGATION()
    while ∃y such that Δ(y) ≠ ∅ do
        pick y with Δ(y) ≠ ∅
        for each constraint C involving y do
            if ¬ FILTER(C, x, y, Δ(y)) then return false
        RESET(Δ(y))
    return true
```

3 Arc consistency algorithms

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

Algorithm 2: CAC filtering algorithm

```
FILTER(in C, x, y, Δ(y)): boolean
    get the parameters of C
    pvMode ← SELECTPENDINGVALUEMODE(C, x, y, Δ(y), pvMode)
    sMode ← SELECTEXISTSUPPORTMODE(C, x, y, Δ(y), sMode)
    (x, a) ← pvMode.FIRSTPENDINGVALUE(C, x, y, Δ(y))
    while (x, a) ≠ nil do
        if ¬ EXISTVALIDSUPPORT(C, x, a, y, sMode) then
1           REMOVEFROMDOMAIN(x, a)
            if D(x) = ∅ then return false
        (x, a) ← pvMode.NEXTPENDINGVALUE(C, x, y, Δ(y), a)
    return true
```

Definition 1 *We call* **pending values** *w.r.t. a 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 the 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. This is also the core of the generic CAC algorithm. Functions selectPendingValueMode and selectExistSupportMode can be ignored at this point. They will be detailed later.

We can now give the principles of Functions firstPendingValue, nextPendingValue and of Function existValidSupport for each existing algorithm. We will consider a constraint C involving x and y and that y is modified. Therefore, the pending values belong only to x.
AC-3: The pending values are the values of $D(x)$, and $\Delta(y)$ is not used at all. All the values of $D(x)$ are considered and the search for a valid support is done by checking in $D(y)$ if there is a support for a value of $D(x)$. There is no memorization of the previous computations, so the same computations can be done several times and the time complexity for one constraint is in $O(d^3)$[1]. The advantage of this approach is that the space complexity is null.
AC-4: In AC-4 the tuple set is pre-computed and store in a structure that we call a **table**. This table contains for every value (x, a) a pointer to the next tuple involving (x, a). Therefore, the space complexity of AC-4 is in $O(d^2)$. The pending values are for each $(y, b) \in \Delta(y)$ all the valid values (x, a) such that $((x, a)(y, b)) \in T(C)$. Note that a value (x, a) can be considered several times as a pending value. The search for a support is immediate because Function existValidSupport can be implemented in $O(1)$ by associated with every value (x, a) a counter which counts the number of time (x, a) has a support in $D(y)$. Then, each time this function is called the counter is decremented (because (x, a) lost a valid support) and if the counter is equals to zero then there is no support. AC-4 was the first algorithm reaching an $O(d^2)$ time complexity, because no computation is made twice. However, a lot of computations are systematically done.
AC-5: This algorithm is mainly a generic algorithm. It has been designed in order to be able to take into account the specificity or the structure of the considered constraints. In other words, Function existValidSupport can be specialized by the user in order to benefit from the exploitation of the structure of the constraint. For instance, functional constraints are more much simple and arc consistency for these constraints can be achieved in $O(d)$ per constraint. Function filter and the propagation mechanism we gave are close to AC-5 ideas.
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 uses the idea of AC-4 to determine the pending values, but instead of considering all

[1] In this paper, we will always express the complexities per constraint, because a constraint network can involved several types of constraints. The usual way to express complexities can be obtained by multiplying the complexity we give by the number of binary constraints of the network.

the values supported by the values in $\Delta(y)$, it exploits the fact that the knowledge of one support is enough. AC-6 can be viewed as a lazy computation of supports. AC-6 introduces another data structure which is a variation of the table: the **S-list**: 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). Contrary to AC-4, 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 $\Delta(y)$, and, for a given $\Delta(y)$, a value (x, a) can be considered only once as a pending value. 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 check 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.

AC-Inference: This algorithm uses the S-lists of AC-6 to determine in the same way the values for which a support must be sought, but the search for a new support is different from the AC-6's method. For every value (x, a), two lists of values are used: **P-list**$[(x, a)]$ and **U-list**$[(x, a)]$. P-list$[(x, a)]$ contains some supports of (x, a), where as U-list$[(x, a)]$ contains the values for which their compatibility with (x, a) has never been tested. When a support is sought for (x, a), it checks first if there is a valid value in P-list$[(x, a)]$, and every non valid value that is reached is removed from the P-list. If there is no valid value is found in the P-list, then the values of U-list$[(x, a)]$ are successively considered until a valid support is found. Every value of the U-list$[(x, a)]$ which is checked is removed from the U-list. When a new support is found, then some inference rules can be applied to deduce new supports and the U-list and P-list are accordingly modified. For instance if (x, a) is found to be a support for (y, b) then it is inferred that (y, b) is a support for (x, a). Some other inference rules can be used like

commutativity or reflexivity (see in [2].) The space and time complexities are in $O(d^2)$ per constraint.
AC-Identical: This is an extension of AC-Inference which exploits the fact that the same constraint can be defined on different variables. In this case, any knowledge obtain from one constraint is inferred for the other similar constraints.
AC-2000: This is a modification of AC-3. The pending values are the values of $D(x)$ that have a support in $\Delta(y)$. No extra data is used, so it is costly to compute the pending values. Thus, AC-2000 proposes to use this set of pending values only if $|\Delta(y)| < 0.2|D(x)|$; otherwise $D(x)$ is considered as in AC-3. Therefore, AC-2000 is the first adaptive AC algorithm.
AC-2001: This algorithm is based on AC-3 and uses the last data of AC-6. That is, the pending values are the same as for AC-3 and function EXISTVALIDSUPPORT is similar as the AC-6'one, except that it is checked if the last value is valid. This algorithm inherits of the space complexity of AC-6, without using the S-lists. Note also that this presentation of AC-2001 is original and simpler than the one given in [3].
AC-3.3: AC-3.3 is an improvement of AC-2001 which associates with every value (x, a) a counter corresponding to a lower bound of the size of S-list$[(x, a)]$. The algorithm does not use any S-list, but counters instead. When a support is sought for, the counter of (x, a) is first tested, if it is strictly greater than 0 then we know that a valid support exists. This support cannot be identified but we know that there is one. If (y, b) is deleted then the counters of all the values supported by (y, b) are decremented.

We will not consider AC-8 [4], AC-3_d [8], and AC-3.2 [5], because they mainly improve AC-3 by proposing to propagate the constraint in regards to specific orderings, and this is not our purpose.

The AC algorithms may use the following data structures:
Support: the current support of (x, a) is denoted by support$[(x, a)]$.
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, P-List, U-list: these are classical list data structures. For any list L we will consider that functions ADD$(L, (x, a))$ and REMOVE$(L, (x, a))$ are available. These functions respectively add (x, a) to L, and remove (x, a) from L. We will also assume that these function and the size of a list can be computed in $O(1)$.
Tuple counters: there are represented by counter$[(x, a)]$ which counts the number of tuples in $T(C)$ containing (x, a) that are valid.
Table: A table is the set of tuples $T(C)$ associated with two functions: FIRSTTUPLE(C, y, b) which returns the first tuple of $T(C)$ containing (y, b) NEXTTUPLE$(C, x, y, b, a))$ which returns the first tuple of $T(C)$ containing (y, b) and following the tuple $((x, a), (y, b)$. These functions return nil when no such specified tuple exists.

Now, we propose to identify the different concepts used by the existing algorithms instead of having one function per algorithm and one parameter corresponding to each specific algorithm.

Algorithm 3: Pending values selection depending on $pvMode$ (b is a local data.)

$pvMode =_{\underline{\mathsf{pvD}}}$

firstPendingValue$(C, x, y, \Delta(y))$: return first$(D(x))$

nextPendingValue$(C, x, y, \Delta(y), a)$: return next$(D(x), a)$

$pvMode =_{\underline{\mathsf{pv}\Delta\mathsf{s}}}$

firstPendingValue$(C, x, y, \Delta(y))$: value
- $b \leftarrow$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**
  - $(x, a) \leftarrow$seekValidSupportedValue(C, y, b)
  - **if** $(x, a) \neq nil$ **then** return (x, a)
  - $b \leftarrow$next$(\Delta(y), b)$
- return nil

$pvMode =_{\underline{\mathsf{pv}\Delta\mathsf{t}}}$

firstPendingValue$(C, x, y, \Delta(y))$: value
- $b \leftarrow$first$(\Delta(y))$
- $(x, a) \leftarrow$firstTuple(C, y, b)
- return traverseTuple$(C, x, y, \Delta(y), a)$

nextPendingValue$(C, x, y, \Delta(y), a)$: value
- $a \leftarrow$nextTuple(C, y, b, a)
- return traverseTuple$(C, x, y, \Delta(y), a)$

traverseTuple$(C, x, y, \Delta(y), a)$: C-value
- **while** $(y, b) \neq nil$ **do**
  - **while** $(x, a) \neq nil$ **do**
    - **if** $a \in D(x)$ **then** return (x, a)
    - $(x, a) \leftarrow$nextTuple(C, x, y, b, a)
  - $b \leftarrow$next$(\Delta(y), \delta a)$
  - $(x, a) \leftarrow$firstTuple$(C, x, y, \delta a)$
- return nil

$pvMode =_{\underline{\mathsf{pv}\Delta\mathsf{c}}}$

firstPendingValue$(C, y, \Delta(y))$: value
- $a \leftarrow$first$(D(x))$
- return seekCompatible$(C, x, y, \Delta(y), a)$

nextPendingValue$(C, y, \Delta(y), a)$: value
- $a \leftarrow$next$(D(x), a)$
- return seekCompatible$(C, x, y, \Delta(y), a)$

seekCompatible$(C, x, y, \Delta(y), a)$: value
- **while** $(x, a) \neq nil$ **do**
  - **for** *each* $b \in \Delta(y)$ **do**
    - **if** $((x, a), (y, b)) \in T(C)$ **then** return (x, a)
  - $(x, a) \leftarrow$next$(D(x), a)$
- return nil

$pvMode =_{\underline{pvG}}$: example of generic function: $<$ constraint

firstPendingValue$(C, y, \Delta(y))$: value
- return next$(D(x), \max(D(y)) - 1)$

nextPendingValue$(C, y, \Delta(y), a)$: return next$(D(x), a)$

Thus, we will have a configurable algorithm from which every AC algorithm could be obtain by combining some parameters, each of them corresponding to a concept.

4 Pending values

Finding efficiently a small set of pending values is difficult because pending values sets deal with two different notions at the same time: validity and support. Thus, several set of pending values have been considered. We identify four sets:

1. The values of $D(x)$ (like in AC-3, AC-2001, AC-3.3). This set is denoted by $\underline{\mathsf{pvD}}$.
2. The valid values currently supported by the values of $\Delta(y)$, that is the valid values in the S-lists of $\Delta(y)$. This set is used by AC-6, AC-7, AC-Inference, AC-Identical. It is denoted by $\underline{\mathsf{pv}\Delta\mathsf{s}}$.
3. The values that belong to a tuple containing a value of $\Delta(y)$. A value is pending as many times as it is contained in such a tuple. AC-4 uses this set, and it is denoted by $\underline{\mathsf{pv}\Delta\mathsf{t}}$.
4. The values of $D(x)$ compatible with at least one value of $\Delta(y)$, as in AC-2000. This set is denoted by $\underline{\mathsf{pv}\Delta\mathsf{c}}$.

Since we aim to have a generic algorithm, we propose to define a fifth type: $\underline{\mathsf{pvG}}$ which represents any function given by the user. For instance, for $x < y$ only the modifications of the maximum value of $D(y)$ can lead to new deletions. Thus, the pending values are the values of $D(x)$ that are greater than the maximum value of $D(y)$.

Algorithm 3 is a possible implementation of the computation of pending values. Depending on the set of pending values, the algorithm traverses a particular set. Note that some functions require "internal data" (a data whose value is stored). We assume that$\textsc{first}(D(x))$ returns the first value of $D(x)$ and $\textsc{next}(D(x), a)$ returns the first value of $D(x)$ strictly greater than a. Function $\textsc{seekValidSupportedValue}(C, x, a)$ returns a valid supported value belonging to the S-list$[(y, b)]$.

Algorithm 4: Function seekValidSupportedValue

```
seekValidSupportedValue(C, x, a) : value
    for each value (y, b) ∈ S-list[(x, a)] do
        if b ∈ D(y) then return (y, b)
        remove(S-list[(x, a)], y, b)
    return nil
```

Algorithm 5: Function existValidSupport

```
existValidSupport(C, x, a, y, sMode): boolean
    if slist then remove(S-list[support[(x, a)]], (x, a))
    (y, b) ← nil
    if last then if last[(x, a)] ∈ D(y) then (y, b) ← last[(x, a)]
    if inf and (y, b) = nil then
        (y, b) ← seekValidSupportedValue(C, x, a)
    if (y, b) = nil then (y, b) ← sMode.seekSupport(C, x, a, y)
    if slist then if (y, b) ≠ nil then add(S-list[(y, b)], (x, a))
    if (y, b) ≠ nil then support[(x, a)] ← (y, b)
    return ((y, b) ≠ nil)
```

5 Existence of a valid support

This function also differentiates the existing algorithms. Almost each algorithm uses a different method. We can identify eight ways to determine whether a support exists for (x, a):

1. Check in the domain from scratch (AC-3, AC-2000.)
2. Check if the last value is still valid and if not check in the domain from the last value (AC-2001.)
3. Check in the domain from the last value (AC-6.)
4. Test if a support can be found in S-list$[(x, a)]$, then check in the domain from the last value. Wen searching in the domain uses the fact that last values are available to avoid explicit compatibility checks (AC-7.)
5. Check if there is a valid support in P-list$[(x, a)]$, if there is none check the compatibility with the valid values of U-list$[(x, a)]$. When some compatibility checks are made, then deduce the results of some other compatibility checks and update accordingly some U-lists and P-lists (AC-Inference, AC-identical.)
6. Decrement the counter storing the number of valid supports and test if is strictly greater than 0 (AC-4.)
7. A specific function dedicated to the constraint is used.
8. Use of counters storing a lower bound of the size of the S-list of (x, a), and check in the domain from the last value (AC-3.3.)

The last point deserves a particular attention. The time complexity of using a counter of the number of elements in a list is the same as the management of the list. Moreover, AC-3.3 implies that the counters are immediately updated when a value is removed, which is not the case with AC-7, and the lazy approach used by AC-7 to maintain the consistency of the S-list has been proved more efficient. Therefore, we will prefer the explicit use of S-lists to the use of a lower bound of the size of the S-lists of AC-3.3.

We propose to consider the following parameters:

• last: the search for a valid support is restarting from the last value. The last value is also used to avoid some negative checks (AC-6, AC-7, AC-Inference, AC-Identical, AC-2001, AC-3.3.)

Algorithm 6: Functions seeking for a valid support

$sMode =$ sD

```
SEEKSUPPORT(C, x, a, y) : value
    b ← FIRST(D(y))
    if last then
        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)
    else
        while b ≠ nil do
            if ((x, a), (y, b)) ∈ T(C) then return (y, b)
            b ←NEXT(D(y), b)
    return nil
```

$sMode =$ sC

```
SEEKSUPPORT(C, x, a, y) : value
    counter[(x, a)] ← counter[(x, a)] − 1
    if counter[(x, a)] = 0 then return nil
    else return (y,FIRST(D(y)))
```

$sMode =$ sT

```
SEEKSUPPORT(C, x, a, y) : value
    for each (y, b) ∈ P-list[(x, a)] do
        REMOVE(P-list[(x, a)], (y, b))
        if b ∈ D(y) then return (y, b)
    for each (y, b) ∈ U-list[(x, a)] do
        REMOVE(U-list[(x, a)], (y, b))
        REMOVE(U-list[(y, b)], (x, a))
        if ((x, a), (y, b)) ∈ T(C) then
            ADD(P-list[(y, b)], (x, a))
            if b ∈ D(y) then return (y, b)
    return nil
```

$sMode =$ sGen example of generic function: $<$ constraint

```
SEEKSUPPORT(C, x, a, y) : return false
```

• inf: the search for a valid support is first done by searching for a valid supported value in the S-list (AC-7, AC-3.3.)

• slist: this parameter means that the S-lists are used. It is implies by inf and pvΔs (AC-6, AC-7, AC-Inference, AC-Identical, AC-3.3.)

• sD: the search for a support is made by testing the compatibility between (x, a) and the values in $D(y)$ (AC-3, AC-6, AC-7, AC-2000, AC-2001, AC-3.3.)

• sC: the search for a valid support consists of decrementing the counter of valid tuples and checking if it is > 0 (AC-4.)

• sT: the search for a valid support is made by testing the validity of the values of P-list$[(x, a)]$ and by checking if there is a value in U-list$[(x, a)]$ which is valid and compatible with (x, a) (AC-Inference, AC-Identical.)

• sGen: the search for a valid support is defined by a function provided by the user and dedicated to the constraint. We present an example for the $<$ constraint.

From these parameters we can now propose a possible code for Function EXISTVALIDSUPPORT of CAC algorithm (see Algorithm 5.) Possible instantiations of Function SEEKSUPPORT are given by Algorithm 6. An example is also given for constraint $<$.

6 Analysis of different methods

The main issue of AC algorithms is to deal with two different notions: support and validity. It is difficult to handle these two notions at the same time. Thus, the algorithms usually privilegiate one notion:

• When constructing the pending values set, pvD algorithms totally ignores the notion of support. The other algorithms try to combine the two notions: pvDc algorithms consider first the validity, whereas pvΔt algorithms deal first with all supports. And, pvΔs algorithms traverse the current supported values and check the validity.

• When searching for a new support, sD algorithms consider the valid values, and the check if there are support, whereas sT algorithms traverse the supports and check for their validity.

7 Nomenclature

From the different concepts we have identified we can propose a new nomenclature for the AC algorithms. Until now, the naming used the prefix "AC-" followed by a number or date. Excepted AC-Inference or AC-Identical which have tried to express a little bit some ideas of the algorithms, it is clearly impossible to understand the specificity of each algorithm from their name.

The nomenclature we propose uses CAC as prefix which stands for Configurable Arc Consistency algorithm. Then the combinations of parameters corresponding to the AC algorithm are added to the CAC prefix. For instance, CAC-pvD-sD means that the pending values are the values of $D(x)$ and that a

new support is sought in the domain by checking the compatibilities between values. This is exactly the description of AC-3. For adaptive algorithm a "/" is used to differentiate the possibilities: AC-2000 is renamed CAC-pvΔc/pvD-sD. We can describe all the existing algorithms:

name	new name
AC-3	CAC-pvD-sD
AC-4	CAC-pvΔt-sC
AC-6	CAC-pvΔs-last-sD
AC-7	CAC-pvΔs-last-inf-sD
AC-Inference or AC-Identical	CAC-pvΔs-sT
AC-2000	CAC-pvΔc/pvD-sD
AC-2001	CAC-pvD-last-sD
AC-3.3	CAC-pvD-last-inf-sD

Note that CAC-pvΔs-last-sD is a slight improvement of AC-6, because some negative checks are avoided.

8 Adaptive algorithm

The advantage of adaptive algorithm is to avoid some pathological cases of each algorithm. A property which exactly differentiates AC-2001 and AC-6 in regards to the number of operations they need to establish arc consistency has been given in [3]:

Property 1 *The number of values that are considered to find the pending values in:*

- *a* $\underline{\text{pvD}}$ *oriented algorithm is* $\#(pvD) = |D(x)|$.
- *a* $\underline{\text{pv}\Delta\text{s}}$ *oriented algorithm is*
$$\#(pv\Delta s) = |\Delta(y)| + \sum_{b \in \Delta(y)} |S\text{-}list[(y,b)]|$$

These two numbers are sufficient to differentiate AC-2001 and AC-6 because they use both the same algorithm to find a support for a value. This is clearly shown by their new names that are respectively CAC-pvD-last-sD and CAC-pvΔs-last-sD. So, by considering the method to find the pending values that studies the smallest number of values we can define an algorithm which is better than any of two previous ones. We can use first a $\underline{\text{pv}\Delta\text{s}}$ oriented algorithm and then switch to a $\underline{\text{pvD}}$ one and conversely.

Unfortunately, it is difficult to quickly compute $\#(pv\Delta s)$. The sum, indeed, needs to consider every value of the delta domain independently. However, we immediately have: $\#(pv\Delta s) \geq 2|\Delta(y)|$, and $|\Delta(y)|$ can be incrementally maintained, thus we can consider that we know its value in $O(1)$. Algorithm 7 is a possible implementation of the functions selecting $sMode$ and $pvMode$ that is used by Algorithm 2 (AC-2000 is taken into account in this function.) Switching from a type of algorithm to another one can also cause some other problems, because the different types of algorithms do not use the same data structures. When switching from an algorithm using a data structure to an algorithm that

Algorithm 7: Selection of pvMode and sMode

```
selectPendingValueMode(C,x,y,Δ(y),pvMode):pvMode
    if pvMode=pvΔc/pvD then
        if |Δ(y)| < 0.2|D(x)| then return pvΔc
        return pvD
    if pvMode=pvD/pvΔs then
        if |D(x)| < 2.|Δ(y)| then return pvD
        if |D(x)|<|Δ(y)|+∑_{b∈Δ(y)} |S-list[(y,b)]| then return pvD
        return pvΔs
    return pvMode
selectExistSupportMode(C, x, a, sMode) : sMode
    if sMode=sD/sT then
        if |D(x)| < |P-list[(x,a)]| then return sD
        return sT
    return sMode
```

does not use that data structure we have two possibilities: either the data structure is updated after switching, or it is systematically updated even if it is not used. For the S-lists, the cost to maintain them is $O(1)$ per deletion or addition therefore the second solution is simpler. For the U-lists and the P-lists there is no problem because they do not need to be updated.

We have seen that it is possible to change the way the pending values are computed. Two other possibilities are: use ot not the inf parameter, and switch from sD and sT and conversely. However, it is much more complicated to find a good criteria of selection, because the lists are modified when using inf or sT. Thus, at a given moment the size of the list can be not in favor of one method but becomes strongly in favor of this method after its application. After some experiments it appears that the switch from inf to no inf does not change anything, and it appears that it is interesting to switch from sD to sT and conversely. If the size of the domain is smaller than the size of the P-list then sD is selected, else sT is selected. (see Algorithm 7.)

9 Experiments

We propose a comparisons of the MAC version of the algorithms on the well-known RLFAPs benchmarks. We give the results only for instances SCEN#1, SCEN#11, SCEN#8 because the results are quite representative (and also due to the lack of space). No specific ordering are consider for the constraints (all the algorithms consider the same ordering). For each algorithm we give the ratio between the time needed by the algorithm to solve the problem over the time needed by the best algorithm:

pvΔt	•											
pvΔc			•									
pvΔs						•	•	•	•	•	•	•
pvD		•	•	•	•			•	•		•	•
last				•	•	•	•	•	•		•	•
inf					•		•		•			
sD		•	•	•	•	•	•	•	•			•
sC	•											
sT										•	•	•
#1	19.4	4.2	3.7	2.8	2.4	3.7	3.7	3.2	3.2	1.5	1.1	1
#11	15	7	6.8	4	3.3	2.6	2.5	1.8	1.8	1.7	1.1	1
#8	59.3	68.2	67.3	50	48	21	19	4	3	4	1.1	1

This results show clearly that the adaptive algorithm performs better than non adaptive and that the $\underline{\text{sT}}$ algorithms are better than the others.

10 Conclusion

We have presented CAC a new configurable, generic and adaptive algorithm, which is able to represent all existing algorithms. We have clearly differentiate all the existing algorithms thanks to the identification a the most important concepts. We have proposed new combination of concepts that perform well in practice as shown by the experimental results we gave.

References

1. C. Bessière. Arc-consistency and arc-consistency again. *Artificial Intelligence*, 65(1):179–190, 1994.
2. C. Bessière, E. 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. Mackworth. Consistency in networks of relations. *Artificial Intelligence*, 8:99–118, 1977.
7. R. Mohr and T. Henderson. Arc and path consistency revisited. *Artificial Intelligence*, 28:225–233, 1986.
8. M. 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.
9. P. Van Hentenryck, Y. Deville, and C. Teng. A generic arc-consistency algorithm and its specializations. *Artificial Intelligence*, 57:291–321, 1992.
10. Y. Zhang and R. Yap. Making ac-3 an optimal algorithm. In *Proceedings of IJCAI'01*, pages 316–321, Seattle, WA, USA, 2001.