Lazy Arc Consistency

Thomas Schiex  
INRA  
BP 27, Castanet-Tolosan  
31326 Cedex, France  
tschiex@toulouse.inra.fr

Jean-Charles Régis  
ILOG SA.  
BP 85, Gentilly  
F-94253 Cedex, France  
rexis@ilog.fr

Christine Gaspin  
INRA  
BP 27, Castanet-Tolosan  
31326 Cedex, France  
gaspin@toulouse.inra.fr

Gérard Verfaillie  
CERT/ONERA  
BP 4025, Toulouse  
31055 Cedex, France  
verfail@cert.fr

Abstract

Arc consistency filtering is widely used in the framework of binary constraint satisfaction problems: with a low complexity, inconsistency may be detected and domains are filtered. In this paper, we show that when detecting inconsistency is the objective, a systematic domain filtering is useless and a lazy approach is more adequate. Whereas usual arc consistency algorithms produce the maximum arc consistent sub-domain, when it exists, we propose a method, called LAC7, which only looks for any arc consistent sub-domain.

The algorithm is then extended to provide the additional service of locating one variable with a minimum domain cardinality in the maximum arc consistent sub-domain, without necessarily computing all domain sizes.

Finally, we compare traditional AC enforcing and lazy AC enforcing using several benchmark problems, both randomly generated CSP and real life problems.

The Constraint Satisfaction Problem (CSP) framework is increasingly used to represent and solve numerous OR and AI problems. When constraints are binary, arc consistency filtering is one of the most prominent filtering techniques, applied either before any search, or incrementally during backtrack search: (1) it has a limited space and time worst-case complexity, (2) if a domain becomes empty while filtering, the inconsistency of the problem is proven, (3) otherwise, variable domains are filtered and the search for a solution can start on a reduced space.

On some problems, systematic domain filtering may become unproductive and costly. This observation has already been made about forward-checking in (ZE89) and largely clarified in (DM94): the only possible cause for backtrack being a wipe-out, it suffices to prove that at least one value remains in each filtered domain. Obviously the worst-case complexity is the same as for usual forward-checking and the average-case behavior is far better, especially when the domains are large.

This paper is devoted to a similar approach applied to arc consistency filtering. Traditional AC filtering try to produce, when it exists, the maximum arc consistent sub-domain. If this maximum arc consistent sub-domain does not exist (a domain wipe-out occurred), inconsistency is proven. If it exists, it can be used as a basis for a further search, since removed values cannot take part in any solution. When considering wipe-out detection only, the computation of the maximum arc consistent sub-domain is useless and one arc consistent sub-domain is sufficient since it proves the absence of wipe-out.

In some cases, wipe-out detection alone is not enough: backtrack tree-search algorithms such as Really Full Look-Ahead or MAC also use domain sizes as a heuristic to choose the next variable to instantiate. Lazy arc consistency can then be extended to provide the additional service of locating one variable with a minimum domain size in the maximum arc consistent domain, without exhaustive filtering.

After a short introduction to constraint satisfaction problems and arc consistency, lazy arc consistency filtering is introduced and the corresponding algorithm, called LAC7, is described. We prove its correctness and study its space and time complexity. We then extend the algorithm in order to locate a variable with a minimum domain size and we experiment and compare these algorithms with traditional AC enforcing algorithms.

Arc consistency filtering

A binary CSP is defined as follows:

Definition 1 A binary CSP is a triple $(V, D, R)$ where:

- $V$ is a sequence $(1, \ldots, i, \ldots, n)$ of $n$ variables;
- $D$ is a sequence $(D_1, \ldots, D_i, \ldots, D_n)$ of domains, such that $\forall i \in V$, $D_i$ is the finite set of possible values for $i$; $d$ is the size of the largest domain;
- $R$ is a sequence $(R_{ij}, R_{ji})$ of $e$ binary relations (or constraints) such that $\forall R_{ij} \in R$, $R_{ij}$ relates the two variables $i$ and $j$ and is defined by a subset of the Cartesian product $D_i \times D_j$ which specifies the allowed pairs of values for variables $i$ and $j$.

As it is usual for AC enforcing algorithms, we associate to any binary CSP a symmetric directed graph $G$, with one vertex for each variable and two directed edges $(i, j)$ and $(j, i)$ for each constraint between variables $i$ and $j$. Since relations are bidirectional (this is not a restriction), if the relation $R_{ij}$ is associated to the edge $(i, j)$, a relation $R_{ji}$ can be associated to the inverse edge $(j, i)$, such that $\forall a \in D_i, b \in D_j, R_{ij}(a, b) = R_{ji}(b, a)$. We will use $\text{EDGES}(G)$ to refer to the set of directed edges in $G$ and $\text{NEIGHBORS}(i)$ to refer to the set of variables $j$ such that
(i, j) ∈ EDGES(G). In the remainder of the paper, i, j, ... will be used to refer to variables, and a, b, ... to refer to values.

Definition 2 If D = (D1, ..., Dn) is a CSP domain, a sub-domain D' is a sequence (D'1, ..., D'n), s.t. ∀i, D'i ⊆ Di.

Arc consistency (AC) is a local consistency property, which uses the concept of support and viability.

Definition 3 Let D' = (D'1, ..., D'n) be a sub-domain, i be a variable, a ∈ Di be a value of i and (i, j) ∈ EDGES(G); the value a is supported by D' along (i, j) iff there exists a value b ∈ D'j such that a supports b along (j, i). Obviously, a is also a support for b along the inverse edge (j, i).

Definition 4 Let D' = (D'1, ..., D'n) be a sub-domain, i be a variable and a ∈ D'i be a value of i; the value a is viable with respect to D' iff ∀j ∈ NEIGHBORS(i), a is supported by D'j along (i, j).

Definition 5 A sub-domain D' is arc consistent if it is non empty (∀i ∈ V, D'i ≠ ∅), and all the values in D' are viable with respect to D'.

Property 1 The union of two arc consistent sub-domains is also arc consistent. Thus, if it exists, there is one maximum arc consistent sub-domain (w.r.t. the partial order induced by the inclusion relation). This maximum sub-domain is the union of all arc consistent sub-domains.

Property 2 If a CSP is consistent, there exists a maximum arc consistent sub-domain and any value which takes part in a solution belongs to it.

Arc consistency filtering produces the maximum arc consistent sub-domain (if it exists) by deleting all the values which are not viable with respect to the current domain D. It may either detect inconsistency, using the first part of property 2 or else reduce the search space, using the second part of property 2.

Filtering a CSP by arc consistency can be achieved, either before any search, or incrementally during a back-track search (Nad89; SF94). Many algorithms have been proposed to enforce arc consistency: first AC3 (Mac77), then AC4 (MH86) with optimal worst-case time complexity O(ed^2), AC5 (vHDT92), AC6 (Bes94), which brings a lower worst-case space complexity (O(ed)). More recently, AC6++/AC7 (BFR95) has been introduced: it uses the fact that constraints are bidirectional to improve AC6. Finally, the AC-Inference schema (BFR95) tries to exploit specific constraint properties in order to save constraint checks, but it has a space complexity O(ed^2). We have chosen the algorithm AC7 as the basis of our work.

Lazy Arc Consistency Filtering

Lazy AC filtering relies on the fact that (1) an arc consistent sub-domain is a sub-domain of the maximum arc consistent sub-domain and (2) a consistent CSP has necessarily an arc consistent sub-domain. The occurrence of a wipe-out is therefore equivalent to the inexistence of an arc consistent sub-domain. Consider the CSP whose so-called micro-structure (or consistency graph) is given below. For each of the three constraints, each compatible pair of values is represented by an edge. The three domains are respectively D1 = D2 = D3 = {1, 2, 3, 4}.

The CSP has a single solution: (4, 4, 2). Its maximum arc consistent sub-domain is ({2, 3, 4}, {2, 3, 4}, {1, 2, 3}). It has two arc consistent sub-domains ({2, 3}, {2, 3}, {1, 3}) and ({4}, {4}, {2}). Proving that any of them is arc consistent would also prove that no wipe-out can occur when AC is enforced.

The LAC7 algorithm defined in this paper is derived from the algorithm AC7 proposed in (BFR95). Therefore, it conserves all the desirable properties of AC7 and exploits the general property of bidirectionality verified by any constraint (∀a ∈ D1, b ∈ D2, Rij(a, b) = Rji(b, a)).

Data structures: the data structures of LAC7 contain all the data structures of AC7 plus some new data-structures for laziness (but LAC7 may nevertheless need much less memory than AC7 because of its laziness). Since LAC7 tries to build an arc consistent sub-domain D' ⊆ D, it needs to remember, for each variable i, which values from the initial domain Di are actually in the sub-domain D'i and which values remain available for a possible insertion in D'i. Two arrays of booleans ACTIVE[i, a] and UNCHECKED[i, a] are used with this purpose. For each variable i, an integer CARDACTIVE[i] contains the number of values of its domain which are currently active.

As in AC7, sets of supported values SUPPORTED[(i, j), a] are used to remember the values b for which a is a current support on edge ((i, j))2 (not necessarily the smallest support, unlike AC6). The array INF SUPPORT[(i, j), a] contains, for each ((i, j), a) a value b such that no support for a on edge (i, j) can be found strictly before b. Precisely, the data structures of LAC7 are composed of:

- an array of booleans, ACTIVE[i, a], keeps track of the values that are currently in D'i. In this array, each initial domain Di is considered as the integer range 1 ... |Di|. The following constant time procedures are used to handle Di lists: last(Di) returns the greatest value in Di if Di ≠ ∅ or 0 else. If a ∈ Di, {last(Di)}, next(a, Di) returns the smallest value in Di greater than a. remove(a, Di) removes value a from Di.

- an array of integers, CARDACTIVE[i] holds the number of active values for each variable;

- an array of booleans, UNCHECKED[i, a], keeps track of the values which have not been introduced in D'i. No support is sought for unchecked values and they cannot support active values. ACTIVE[i, a] and UNCHECKED[i, a] can not

---

1 We consider here that arc consistency is strong 2-consistency.

2 Traditionally, these sets are denoted by S_{ij}. 

Data Consistency 217
be simultaneously true. After execution, an active value is provenly viable, an unchecked value has an unknown status and a value which is neither active nor unchecked is deleted.

- an array of lists, SUPPORTED[(i, j), a], contains all the active values b from D_j which are currently considered as supported by (i, a) on edge (j, i), j \in NEIGHBORS(i).

As in AC7, the current support of a value is not necessarily the smallest.

- an array of integers, INF SUPPORT[(i, j), a] contains a value from D_j such that every value in D_j compatible with (i, a) is greater than or equal to INF SUPPORT[(i, j), a].

- a single list, SUPPORTSEEKINGLIST is used to store demands for support. It contains edge-value pairs such as ((i, j), a) (value a seeking support on edge (i, j), j \in NEIGHBORS(i)). It replaces the WaitingList of AC6 and the two lists of AC7.

The SUPPORTED[(i, j), a] and INF SUPPORT[(i, j), a] of AC7 are used by LAC7 to guarantee that AC7 properties are still verified by LAC7 (see (BFR95)).

**Algorithm**: the algorithm is embodied in the function LAC7. All the data-structures are denoted by global variables, with unlimited scope. Initially, all the values are unchecked and inactive. There are two main operations:

1. When an unchecked value (i, a) is activated, a support has to be found for (i, a) on all the edges (i, j), j \in NEIGHBORS(i). Therefore, all corresponding pairs ((i, j), a) are added in the SUPPORTSEEKINGLIST (see function ActivateValue on next page);

2. In order to find a support on edge (i, j) for value (i, a), LAC7 first looks in SUPPORTED[(i, j), a] to check if (i, a) already supports an active value (j, b) (see function SeekTrivialSupport). If so, (j, b) also supports (i, a) and (i, a) is inserted in SUPPORTED[(i, j), b] (Def. 3).

   Else, a support is sought among active or unchecked values in D_j, starting from the current INF SUPPORT[(i, j), a] (see function SeekNextSupport). If a support b is found, (i, a) is inserted in SUPPORTED[(i, j), b] and the integer INF SUPPORT[(i, j), a] is updated. If the value b was unchecked, it is activated.

   If no support is found, the value is deleted and made inactive. If no active value remains in the domain, and if no unchecked value is available, wipe-out occurs. Else, an unchecked value is activated (see function EmptyDomain). Then, the pairs ((j, i), b) such that (j, b) was supported by (i, a) are introduced in the SUPPORTSEEKINGLIST.

The algorithm runs until either a wipe-out occurs (if EmptyDomain returns true on line 4) or the SUPPORTSEEKINGLIST becomes empty: all the active values have an active support, an arc consistent sub-domain has been built.

Note that LAC7 offers the usual incrementality of AC algorithms and more: if the status of an unchecked value is desired, it suffices to activate the value and to start again with LAC7 from line 2; if a value a is deleted from D_i, either it is unchecked and nothing has to be done or it is active and it suffices to propagate the deletion as in the algorithm (after line 4) and to start again with LAC7 from line 2. Generally, if a constraint is added, all the active values of the variables linked by this constraint have to seek a support along it and it is then sufficient to start again with LAC7 from line 2.

With LAC7, the usual order on integers, we obtain the arc consistent sub-domain (\{2,3\}, \{2,3\}, \{1,3\}). The sub-domain (\{4\}, \{4\}, \{2\}), a solution, would have been produced if the inverse order had been used.

**Correctness**: We denote D^0 = (D_1^0, ..., D_n^0) the initial domain of the CSP, D = (D_1, ..., D_n) the domain defined by unchecked or active values, D^a = (D_1^a, ..., D_n^a) for

---

**Function LAC7()**: boolean

```
Function LAC7(): boolean
SUPPORTSEEKINGLIST \leftarrow \emptyset
for all i \in V do
    CARDACTIVE[i] = 0
    for all a \in D_i do
        ACTIVE[i, a] \leftarrow false
        UNCHECKED[i, a] \leftarrow true
for all (i, j) \in EDGES(G) do
    for all a \in D_i do
        SUPPORTED[i, j, a] \leftarrow \emptyset
        INF SUPPORT[(i, j), a] \leftarrow 1
1 for all i \in V do
    if EmptyDomain(i) then return false
2 repeat
    Pick ((i, j), a) from SUPPORTSEEKINGLIST
    if ACTIVE[i, a] then
        if SeekTrivialSupport((i, j), a, b) then
            Put a in SUPPORTED[(j, i), b]
        else
            b \leftarrow INF SUPPORT[(i, j), a]
            if SeekNextSupport((i, j), a, b) then
                Put a in SUPPORTED[(j, i), b]  
                INFSUPPORT[(i, j), a] \leftarrow b
            else
                remove(a, D_i)
                ACTIVE[i, a] \leftarrow false
                CARDACTIVE[i] \leftarrow CARDACTIVE[i] - 1
        if EmptyDomain(i) then return false
    for b \in SUPPORTED[(i, j), a] do
        if b \in SUPPORTED[(j, i), b] then
            remove(b, D_j)
            CARDACTIVE[j] \leftarrow CARDACTIVE[j] - 1
    for all i \in V do
        CARDACTIVE[i] \leftarrow CARDACTIVE[i] + 1
7 until SUPPORTSEEKINGLIST = \emptyset
return true
```
active values only and $D^T$ for the maximum arc consistent sub-domain of the CSP (if any). The proof relies on three lemmas.

**Lemma 1** When $\text{LAC}_7$ returns true, $D^a \neq \emptyset \Rightarrow D^a$ is arc-consistent.

When a value is activated (see function $\text{ActivateValue}$), a demand for support on all incident edges is posted in $\text{SUPPORTSEEKLINGLIST}$ and when an active value is found without support, it is removed from $D_i$ and made inactive. So, every active value has either an active support or a demand for support pending. When $\text{LAC}_7$ returns true, $\text{SUPPORTSEEKLINGLIST}$ is empty hence every active value is supported. Now, we have to show that all supports are active. This is obviously true after initialization and it remains true afterwards, since (1) when a support is found by $\text{SeekNextSupport}$, it is immediately activated on line 3 of function $\text{LAC}_7$, (2) $\text{SeekTrivialSupport}$ seeks only active values (test on line 1 of the function), (3) when a value $a$ is activated, the set $\text{SUPPORTED}[i, j, a] \neq \emptyset$ is emptied (line 5 of function $\text{LAC}_7$).

**Lemma 2** If $D^T$ exists then $D^T \subseteq D$.

As in $\text{AC7}$, the array INFSUPPORT is updated in such a way that if $R_{ij}(a, b)$ holds for $(i, a), (j, b) \in D$ then INFSUPPORT$[(i, j), a] \leq b$. Hence, when a value is seeking a support and no trivial active support is found, we can start the search after INFSUPPORT$[(i, j), a]$ without losing any support and we do not have to check $R_{ij}(a, b)$ for values $b$ such that INFSUPPORT$[(i, j), b] > a$. Therefore, a value $(i, a)$ is removed from $D_i$ when it has no support in $D_j$ on edge $(i, j)$. So, if all previously removed values are out of $D^T$, then this value $(i, a)$ is out of $D^T$. Since, initially, $D^T \subseteq D^0 = D$, by induction a value is removed only if it is not in $D^T$ which proves the lemma.

**Lemma 3** When $\text{LAC}_7$ ends, $D^a = \emptyset \Rightarrow \text{LAC}_7$ returned false.

After initialization and line 1, either $D$ and therefore $D_a$ is already empty and $\text{LAC}_7$ returns false or one value of each variable has been activated i.e., $D^a \neq \emptyset$. Afterwards, when an active value $(i, a)$ is deleted, the function $\text{EmptyDomain}$ is called on line 4 of the function $\text{LAC}_7$ and either $D_i$ is non empty and one value is active (or made active) or $D_i$ and therefore $D_a^a$ is empty and $\text{LAC}_7$ returns false.

Now, at the end of $\text{LAC}_7$, if $D^T$ exists, it is included in $D$ (by Lemma 2), which is therefore not empty, thus $\text{LAC}_7$ return true and $D^a$ is not empty (by Lemma 3) and arc consistent (by Lemma 1). Conversely, if $\text{LAC}_7$ returns true, $D^a$ is non empty (by Lemma 3) and arc-consistent (by Lemma 1) and therefore $D^T$ exists.

**Further Analysis**

First of all, the desirable properties of AC7 are simply inherited by $\text{LAC}_7$ because of the data-structures INFSUPPORT and SUPPORTED which are managed as in AC7. The worst-case space complexity of $\text{LAC}_7$ is also $O(ed)$ because the new data-structures $\text{CARDACTIVE}$ and $\text{UNCHECKED}$ are $O(n)$ and ($nd$) respectively. Thus the total space complexity remains $O(ed)$. In practice, it should be noticed that the SUPPORTED lists are empty for non active values, which may actually save a lot of space.

Very simply, one can observe that the time complexity of $\text{LAC}_7$ is bounded by the complexity of AC7 since the algorithm will stop as soon as any arc consistent sub-domain is built (or a wipe-out is detected). $\text{LAC}_7$ can save a lot of constraint checks on loose CSP. In the (unrealistic) case of a CSP entirely composed of constraints such that $R_{ij}(a, b)$ always holds, $\text{LAC}_7$ will perform $O(e)$ constraint checks to prove that no wipe-out can occur while AC7 will perform $O(ed)$ tests (to enforce arc consistency).

**Improving $\text{LAC}_7$:** if the CSP has only one connected component, the line 1 in $\text{LAC}_7$ is useless and applying $\text{EmptyDomain}$ on any variable suffices. This avoids the possibly useless activation of the first value of each variable. Still trying to minimize the arbitrary activation of values, one can observe that $\text{LAC}_7$ seeks support following the

---

3The sets SUPPORTED keep a reasonable $O(ed)$ space complexity as in AC7 because each edge-value pair $((i, j), a)$ has at most one current support (an element of SUPPORTED$(j, i, b)$).
initial domain order. A better idea would be to look for support among already active values first and then only among unchecked values. This is, however, not immediate because the domain order is used in LAC to avoid redundancies in constraint checks. We first need to remember the order of insertion of values in the active domain. This is done using a bounded stack of size d, a pointer to a value being pushed on the stack when this value is activated. Then, in order to avoid redundant constraint checks, we need a second InSupport-like data-structure, called InSupportAct. The original array InSupport[(i, j), a] contains, for each ((i, j), a) a value b such that no support for a on edge (i, j) can be found strictly before b using the initial domain order, while the array InSupportAct[(i, j), a] contains, for each ((i, j), a) the position p in the stack such that no support for a on edge (i, j) can be found before p.

Finally, the SeekNextSupport procedure is modified: a support is first sought in the corresponding bounded stack, starting at InSupportAct[(i, j), a] and then only among unchecked values. The two InSupport data-structures are also used to avoid unnecessarily failed constraint checks, as in AC7. The algorithm defined is noted LAC and has the same worst-case space/time complexities as LAC.

Finding the smallest domain variable: For a given variable v, we will respectively note |v|^T, |v|^a, |v|^u the number of values for v in the maximum arc consistent domain, the number of active values for v and the number of unchecked values for v. Obviously, after LAC has been executed, we have |v|^a \leq |v|^T \leq (|v|^a + |v|^u).

Let \( \alpha = \min_v (|v|^a) \), \( \beta = \min_v (|v|^T) \) and \( \gamma = \min_v (|v|^a + |v|^u) \). According to the previous inequality, we have \( \alpha \leq \beta \leq \gamma \). Therefore, if the condition \( \alpha = \gamma \) is met, we know that a variable \( v_i = \arg\min_v (|v|^a + |v|^u) \) is a minimum domain size variable in the maximum arc consistent domain without necessarily computing the whole maximum arc consistent domain. Otherwise, we can simply activate one unchecked value in all variables \( v \) such that \( |v|^a = \alpha \), launch LAC again, and loop until the condition is met. This will necessarily occur since when all unchecked values are exhausted, \( |v|^a = |v|^u + |v|^a \). This defines the MinLAC* algorithm.

In the spirit of the A* algorithms, one could also identify a variable which is guaranteed to be close to the optimum by using the new condition \((1 + \varepsilon)\alpha \geq \gamma\). More generally, instead of using domain size, we may consider any criteria \( f \) that depends monotonically on the domain size, for example the domain by degree ratio, usually much more efficient.

Experiments

We have compared LAC, LAC + and MinLAC + with AC7 (BFR95). For AC7, the problem considered is the computation of the maximum arc consistent sub-domain. The algorithm is modified as in (BFR95) to stop as soon as a wipe-out occurs. For LAC and LAC + the problem is to compute any arc consistent sub-domain or to stop when a wipe-out occurs. For MinLAC +, the problem is both to compute an arc consistent sub-domain and to find an optimal variable or to stop when a wipe-out occurs. In the sequel, two criteria will be considered: minimum domain size (noted MinLAC + D) and minimum domain size by degree ratio (noted MinLAC + R). We report the number of constraint checks or ccks. (testing a constraint \( R_{ij} \) on a pair of values, see function SeekNextSupport at line 2).

Academic problems: For the Zebra problem, the results obtained using random orderings for variables and domains are 899 ccks. for AC7, 408 ccks. for LAC and 452 ccks. for LAC +. The results for MinLAC + D and MinLAC + R are identical to the results of LAC (there exist variables with cardinality one initially).

Fig. 1 presents the number of constraint checks performed on the n queens problem. MinLAC + algorithms have the same performances as AC7 since these problems are already arc consistent with uniform domain size and degree.

Random problems have been generated as in (HF92), with 40 variables, 15 values per domain and a number of constraints equal to \((n - 1) + \left\lfloor \frac{(n-1)(n-2)}{2} \right\rfloor\). The constraint tightness goes from 5\% to 100\% in 5\% steps. Fifty problems are solved at each point. The mean number of constraint checks for all algorithms are given in Fig. 2. Since all LAC algorithms use the AC7 heuristics that consists in propagating deletions immediately, it obtains the good results of AC7 when wipe-out occurs. When no wipe-out occurs, large savings are obtained by laziness.

Things are more subtle with MinLAC + D: when the CSP is already arc consistent, and since all domains have the same size, MinLAC + D carries out all the work done by AC7 to locate a minimum domain variable. But as soon as some values get deleted, the domain of some of the variables diminishes and MinLAC + D saves constraints checks while still locating the minimum domain size variable. This is visible just before the "wipe-out" threshold, which occurs at a constraint tightness of 70\%. MinLAC + R can immediately take advantage of the variability in the degree and immediately saves constraint checks.

LAC seems especially useful on under-constrained CSP. MinLAC + R and MinLAC + D improve AC7 performances, but in limited way because of random CSP artificial
uniformity. We therefore tried the same algorithms on random CSP with a domain size randomly chosen between 5 and 25, with uniform probability. Twenty problems are solved at each point. The results are given in Fig. 3. There is no clear “wipe-out” threshold as in the usual model: wipes out appear for a tightness of 10% but it is only at a tightness of 65% that all the CSP generated actually “wipe-out”. We can see that MinLAC_7^D and MinLAC_7^F save a lot of constraint checks: an important variability in the criteria minimized by MinLAC_7^F seems to help.

Real life problems: we conclude our test with some large problems (up to 680 variables, several thousands of constraints and domain sizes above 70). Eleven radio-link frequency assignment problems have been made available by the French “Centre d’Electronique de l’Armement” (CEL94). It is not surprising that enormous savings are achieved on problem 3 and 11 since these problems are rather under-constrained. For problems with a large number of deleted values (problem 5) or immediate wipe-out (problem 9), the performances of both algorithms are very similar (the differences are due to different orderings induced by different behaviors). Similar results are obtained on other instances or by using MinLAC_7^D.

Further research: The next step is to incorporate LAC_7 or MinLAC_7 algorithms inside a backtrack search algorithm, such as the MAC algorithms (SF94; BFR95) and to evaluate the savings that can be achieved more precisely. For MinLAC_7, all the usual services of AC_7 are still offered: domain wipe-out detection and best variable choice. For LAC_7, larger savings are achieved, but the loss of the domain size information could be costly.

References


Data Consistency 221