Lazy Arc Consistency

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

Jean-Charles Régin
ILOG SA.
BP 85, Gentilly
F-94253 Cedex, France
regin@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 LAC_7, 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. The same 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 its average-case behavior is far better, especially when the domains are very 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 instanciate. 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 LAC_7, 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 $(\ldots, R_{ij}, \ldots)$ 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 $\textsc{Edges}(G)$ to refer to the set of directed edges in G and $\textsc{Neighbors}(i)$ to refer to the set of variables j such that

$(i, j) \in$ EDGES(G). In the remainder of the paper, $i, j, \ldots$ will be used to refer to variables, and $a, b, \ldots$ to refer to values.

Definition 2 *If $D = (D_1, \ldots, D_n)$ is a CSP domain, a sub-domain D' is a sequence $(D'_1, \ldots, D'_n)$, s.t. $\forall i, D'_i \subseteq D_i$.*

Arc consistency (AC) is a local consistency property, which uses the notion of *support* and *viability*:

Definition 3 *Let $D' = (D'_1, \ldots, D'_n)$ be a sub-domain, i be a variable, $a \in D_i$ be a value of i and $(i, j) \in$ EDGES(G); the value a is supported by D'_j along (i, j) iff there exists a value $b \in D'_j$ s.t. $R_{ij}(a, b)$; b is called a support for a along (i, j). Obviously, a is also a support for b along the inverse edge (j, i).*

Definition 4 *Let $D' = (D'_1, \ldots, D'_n)$ be a sub-domain, i be a variable and $a \in D'_i$ be a value of i; the value a is viable with respect to D' iff $\forall j \in$ NEIGHBORS(i), a is supported by D'_j along (i, j).*

Definition 5 *A sub-domain D' is arc consistent[1] iff it is non empty ($\forall i \in V, D'_i \neq \varnothing$), 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.*

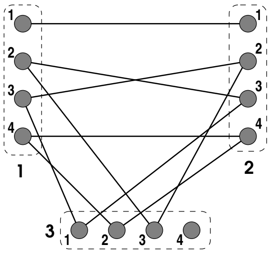
Property 2 *If a CSP is consistent, there exists a maximum arc consistent sub-domain and any value which takes part in a solution belongs to it.*

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 backtrack 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, which tries to exploit specific constraint properties, has been presented in (BFR95), but 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 $D_1 = D_2 = D_3 = \{1, 2, 3, 4\}$.



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 LAC_7 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 ($\forall a \in D_i, b \in D_j, R_{ij}(a, b) = R_{ji}(b, a)$).

Data structures: the data structures of LAC_7 contain all the data structures of AC7 plus some new data-structures for laziness (but LAC_7 may nevertheless need much less memory than AC7 because of its laziness).

Since LAC_7 tries to build an arc consistent sub-domain $D' \subseteq D$, it needs to remember, for each variable i, which values from the initial domain D_i 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 INFSUPPORT$[(i, j), a]$ contains, for each $\langle (i, j), a \rangle$ a value b such that no support for a on edge (i, j) can be found strictly before b. Precisely, the data structures of LAC_7 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 D_i is considered as the integer range $1 \ldots |D_i|$. The following constant time procedures are used to handle D_i lists: $last(D_i)$ returns the greatest value in D_i if $D_i \neq \varnothing$ or 0 else. If $a \in D_i - \{last(D_i)\}$, $next(a, D_i)$ returns the smallest value in D_i greater than a. $remove(a, D_i)$ removes value a from D_i.
- 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 seeked 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 noted S_{ija}.

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, INFSUPPORT$[(i,j),a]$ contains a value from D_j such that every value in D_j compatible with (i,a) is greater than or equal to INFSUPPORT$[(i,j),a]$.
- a single list, SUPPORTSEEKINGLIST is used to store demands for support. It contains edge-value pairs such as $\langle (i,j),a \rangle$ (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 INFSUPPORT$[(i,j),a]$ of AC7 are used by LAC_7 to guarantee that AC7 properties are still verified by LAC_7 (see (BFR95)).

Algorithm: the algorithm is embodied in the function LAC_7. 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 $\langle (i,j),a \rangle$ 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), LAC_7 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$[(j,i),b]$ (Def. 3).

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

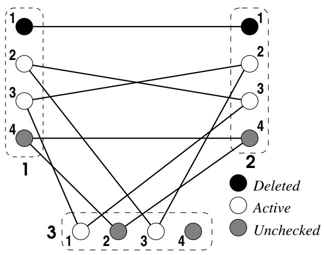
   If no support is found, the value is deleted and made inactive. If no active value remains in the domain, and if no unchecked value is available, wipe-out occurs. Else, an unchecked value is activated (see function EmptyDomain). Then, the pairs $\langle (j,i),b \rangle$ such that (j,b) was supported 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 LAC_7 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

```
   Function LAC7(): boolean
   SUPPORTSEEKINGLIST ← ∅
   for all i ∈ V do
       CARDACTIVE[i] = 0
       FOR all a ∈ D_i DO
           ACTIVE[i, a] ← false
           UNCHECKED[i, a] ← true

   for all (i, j) ∈ EDGES(G) do
       for all a ∈ D_i do
           SUPPORTED[(i, j), a] ← ∅
           INFSUPPORT[(i, j), a] ← 1

1  for all i ∈ 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 ← INFSUPPORT[(i, j), a]
               if SeekNextSupport((i, j), a, b) then
3                  if not ACTIVE[j, b] then ActivateValue(j, b)
                   Put a in SUPPORTED[(j, i), b]
                   INFSUPPORT[(i, j), a] ← b
               else
                   remove(a, D_i)
                   ACTIVE[i, a] ← false
                   CARDACTIVE[i] ← CARDACTIVE[i] − 1
4                  if EmptyDomain(i) then return false
                   for all j ∈ NEIGHBORS(i) do
                       for b ∈ SUPPORTED[(i, j), a] do
5                          Remove b from SUPPORTED[(i, j), a]
                           Put ⟨(j, i), b⟩ in SUPPORTSEEKINGLIST

   until SUPPORTSEEKINGLIST = ∅
   return true
```

with LAC_7 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 LAC_7 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 LAC_7 from line 2.



After the execution of LAC_7, using 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 $D^0 = (D_1^0, \ldots, D_n^0)$ the initial domain of the CSP, $D = (D_1, \ldots, D_n)$ the domain defined by unchecked or active values, $D^a = (D_1^a, \ldots, D_n^a)$ for active values only and $D^\top$ for the maximum arc consistent

```
Function SeekTrivialSupport(in (i,j): edge, in a: value,
out b: value) : boolean
while SUPPORTED[(i,j),a] ≠ ∅ do
    b ← an element of SUPPORTED[(i,j),a]
1   if ACTIVE(j,b) then return true
    else Remove b from SUPPORTED[(i,j),a]
return false

Function SeekNextSupport(in (i,j): edge, in a: value, in
out b: value) : boolean
while b ≤ last(D_j) do
    if (ACTIVE(j,b) or UNCHECKED(j,b)) then
        if INFSUPPORT[(j,i),b] ≤ a then
2           if R_ij(a,b) then return true
        b ← next(b,D_j)
    else
        b ← b + 1
return false

Procedure ActivateValue(in i: variable, in a: value)
ACTIVE[i,a] ← true
CARDACTIVE[i] ← CARDACTIVE[i] + 1
UNCHECKED[i,a] ← false
for all j ∈ NEIGHBORS(i) do
    Put ⟨(i,j),a⟩ in SUPPORTSEEKINGLIST

Function EmptyDomain(in i: variable) : boolean
if CARDACTIVE[i] = 0 then
    if D_i ≠ ∅ then
        a ← an element of D_i
        ActivateValue(i,a)
        return false
    else
        return true
else return false
```

sub-domain of the CSP (if any). The proof relies on three lemmas.

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

When a value is activated (see function ActivateValue), a demand for support on all incident edges is posted in SUPPORTSEEKINGLIST 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 LAC$_7$ returns *true*, SUPPORTSEEKINGLIST 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 SeekNextSupport, it is immediately activated on line 3 of function **LAC**$_7$, (2) SeekTrivialSupport seeks only active values (test on line 1 of the function), (3) when a value a is removed from D_i and made inactive, the set SUPPORTED$[(i,j),a]$ is emptied (line 5 of function **LAC**$_7$).

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

As in 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 loosing any support and we do not have to check $R_{ij}(a,b)$ for values b such that INFSUPPORT$[(j,i),b] > a$. 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 $D^\top$, then this value (i,a) is out of $D^\top$. Since, initially, $D^\top \subseteq D^0 = D$, by induction a value is removed only if it is not in $D^\top$ which proves the Lemma.

Lemma 3 *When LAC*$_7$ *returns,* $(D \neq \varnothing \Rightarrow D^a \neq \varnothing)$ *and* $(D = \varnothing \Rightarrow$ *LAC*$_7$ *returns* false*).*

After initialization and line 1, either a domain is empty and function EmptyDomain returns *true* so LAC$_7$ returns *false* or one value of each variable has been activated *i.e.*, $D_i^a \neq \varnothing$. Afterwards, when an active value (i,a) is deleted, the function EmptyDomain is called on line 4 of the function LAC$_7$ and either D_i is non empty and one value is active (or made active) or D_i is empty and EmptyDomain returns *true* so LAC$_7$ returns *false*.

Now, at the end of LAC$_7$, if $D^\top$ exists, it is included in D (by Lemma 2), which is therefore not empty, thus D^a is not empty (by Lemma 3) and arc consistent (by Lemma 1). Otherwise, $\exists D_i^a = \varnothing$ (or else D^a would be non empty, arc consistent and $D^\top$ would exist) and LAC$_7$ returns *false* (Lemma 3).

Further Analysis

First of all, the desirable properties of AC7 are simply inherited by LAC$_7$ because of the data-structures INFSUPPORT and SUPPORTED, which are managed as in AC7. The worst-case space complexity of LAC$_7$ is also $O(ed)$ because the new data-structures CARDACTIVE and UNCHECKED are $O(n)$ and (nd) respectively[3]. 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 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). 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, 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 LAC$_7$: if the CSP has only one connected component, the line 1 in **LAC**$_7$ is useless and applying 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 LAC$_7$ seeks support following the

[3] The sets SUPPORTED keep a reasonable $O(ed)$ space complexity as in AC7 because each edge-value pair $\langle(i,j),a\rangle$ 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_7 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 INFSUPPORT-like data-structure, called INFSUPPORTACT. The original array INFSUPPORT$[(i,j),a]$ contains, for each $\langle(i,j),a\rangle$ 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 INFSUPPORTACT$[(i,j),a]$ contains, for each $\langle(i,j),a\rangle$ 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 seeked in the corresponding bounded stack, starting at INFSUPPORTACT$[(i,j),a]$ and then only among unchecked values. The two INFSUPPORT data-structures are also used to avoid necessarily failed constraint checks, as in AC7. The algorithm defined is noted LAC_7^+ and has the same worst-case space/time complexities as LAC_7.

Finding the smallest domain variable: For a given variable v, we will respectively note $|v|^\top$, $|v|^a$, $|v|^u$ 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_7 has been executed, we have $|v|^a \leq |v|^\top \leq (|v|^a + |v|^u)$.

Let $\alpha = \min_v(|v|^a)$, $\beta = \min_v(|v|^\top)$ 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 $v_i = argmin_v(|v|^a + |v|^u)$ 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_7 again, and loop until the condition is met. This will necessarily occur since when all unchecked values are exhausted, $|v|^a = |v|^a + |v|^u$. This defines the MinLAC_7^+ algorithm.

In the spirit of the A_ε^* 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_7, LAC_7^+ and MinLAC_7^+ 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_7 and LAC_7^+ the problem is to compute any arc consistent sub-domain or to stop when a wipe-out occurs. For MinLAC_7^+, 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_7^{+D}) and minimum domain size by degree ratio (noted Min$\mathsf{LAC}_7^{+\frac{D}{d}}$). We report the number of constraint checks or cc. (testing a constraint R_{ij} on a pair of values, see function SeekNextSupport at line 2).

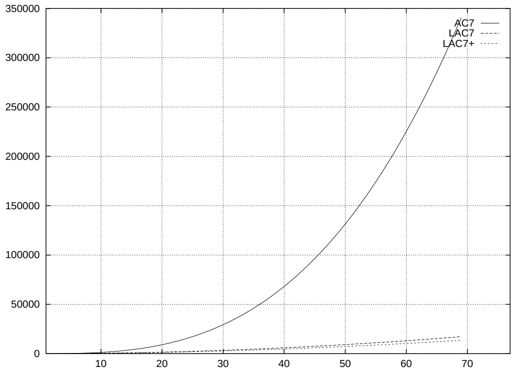


Figure 1: The n queens problem (#cc)

Academic problems: For the Zebra problem, the results obtained using random orderings for variables and domains are 899 cc. for AC7, 408 cc. for LAC_7 and 452 cc. for LAC_7^+. The results for MinLAC_7^{+D} and Min$\mathsf{LAC}_7^{+\frac{D}{d}}$ are identical to the results of LAC_7^+ (there exist variables with cardinality one initially).

Fig. 1 presents the number of constraint checks performed on the n queens problem. MinLAC_7^+ 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 $(n-1)+\lfloor\frac{(n-1).(n-2)}{4}\rfloor$. The constraint tightness goes from 5% to 100% in 5% steps. Fifty problems are solved at each point. The number of cc. for all algorithms are given in Fig. 2. Since all LAC_7 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_7^{+D}: when the CSP is already arc consistent, and since all domains have the same size, MinLAC_7^{+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 variable diminishes and MinLAC_7^{+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%. Min$\mathsf{LAC}_7^{+\frac{D}{d}}$ can immediatly take advantage of the variability in the degree and immediatly saves cc.

LAC_7 appears especially useful on under-constrained CSP. MinLAC_7^{+D} and Min$\mathsf{LAC}_7^{+\frac{D}{d}}$ improve AC7 performances, but in limited way because of random CSP artificial uniformity. We therefore tried the same algorithms on ran-

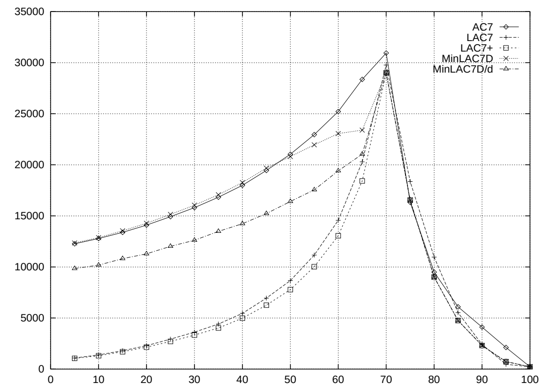


Figure 2: Random CSP (# cc)

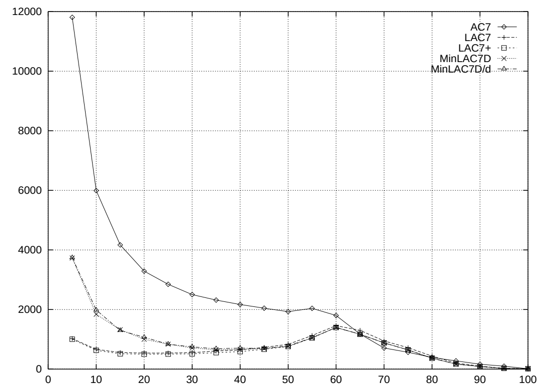


Figure 3: Random CSP, domain variability (# cc)

dom 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: wipe-outs 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$_7^{+\frac{D}{d}}$ save a lot of constraint checks: an important variability in the criteria minimized by MinLAC$_7^{+}$ 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'Électronique de l'Armement" in (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$_7^{+\frac{D}{d}}$.

		Pb. 3	Pb 5	Pb. 9	Pb. 11
AC7	#ccks	412 594	696 221	6 833	638 932
	# del.	0	12 046	499	0
LAC$_7$	#ccks	38 247	877 375	7 754	55 837
	# del.	0	11 452	278	0
LAC$_7^{+}$	#ccks	28 047	338 961	5 572	58 287
	# del.	0	9 898	231	0
MinLAC$_7^{+D}$	#ccks	49 737	338 961	5 572	76 389
	# del.	0	9 898	231	0

The last set of test problems comes from molecular biology (RNA secondary structure prediction). These problems have a complete constraint graph with loose constraints. Beyond the good results in terms of cc., which were quite predictable, another good point of LAC$_7$-style algorithms lies in their low memory consumption when few values are checked. The results of LAC$_7^{+}$ and MinLAC$_7^{+}$ algorithms are the same as LAC$_7$ results.

	tRNAthrTcoli	HIV1	rnasePcoli
AC7 #ccks	146 276	650 839	Memory
# del.	370	497	exhausted
LAC$_7$#ccks	2 850	7 626	70 876
# del.	0	0	0

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 AC7 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

C. Bessière. Arc-consistency and arc-consistency again. *Artificial Intelligence*, 65(1):179–190, 1994.

C. Bessière, E.C. Freuder, and J.C. Régin. Using inference to reduce arc-consistency computation. In *Proc. of the 14th IJCAI*, Montréal, Canada, August 1995.

CELAR. RLFAP benchmarks. URL ftp://ftp.cs.city.ac.uk/pub/constraints/csp-benchmarks/celar, 1994.

M. Dent and R. Mercer. Minimal Forward Checking. In *Proc. of the 6th IEEE International Conference on Tools with Artificial Intelligence (TAI94)*, pages 432–438, New Orleans, LA, 1994.

P. Hubbe and E. Freuder. An Efficient Cross-Product Representation of the Constraint Satisfaction Problem Search Space. In *Proc. of AAAI-92*, pages 421–427, San Jose, CA, 1992.

A. Mackworth. Consistency in networks of relations. *Artificial Intelligence*, 8(1):99–118, 1977.

R. Mohr and T. Henderson. Arc and path consistency revisited. *Artificial Intelligence*, 28(2):225–233, 1986.

B. A. Nadel. Constraint satisfaction algorithms. *Comput. Intell.*, 5(4):188–224, November 1989.

D. Sabin and G. Freuder. Contradicting conventional wisdom in constraint satisfaction. In *Proc. of ECAI-94*, pages 125–129, Amsterdam, 1994.

P. van Hentenryck, Y. Deville, and C. Teng. A generic arc-consistency algorithm and its specializations. *Artificial Intelligence*, 57:291–321, 1992.

M. Zweben and M. Eskey. Constraint Satisfaction with Delayed Evaluation. In *Proc. of IJCAI-89*, pages 875–880, Detroit, 1989.