Non-Viability Deductions in Arc-Consistency Computation

Camilo Rueda* and Frank D. Valencia**

[1] Dept. of Computer Science, Universidad Javeriana Cali
Cali, Colombia
crueda@puj.edu.co
[2] Dept. of Information Technology, Uppsala University
Uppsala, Sweden
frankv@it.uu.se

Abstract Arc-Consistency (AC) techniques have been used extensively in the study of Constraint Satisfaction Problems (CSP). These techniques are used to simplify the CSP before or during the search for its solutions. Some of the most efficient algorithms for AC computation are AC6++ and AC-7. The novelty of these algorithms is that they satisfy the so-called *four desirable properties* for AC computation. The main purpose of these interesting properties is to reduce as far as possible the number of constraint checks during AC computation while keeping a reasonable space complexity. In this paper we prove that, despite providing a remarkable reduction in the number of constraint checks, the four desirable properties do not guarantee a minimal number of constraint checks. We therefore refute the minimality claim in the paper introducing these properties. Furthermore, we propose a *new desirable property* for AC computation and extend AC6++ and AC-7 to consider such a property. We show theoretically and experimentally that the new property provides a further substantial reduction in the number of constraint checks.

1 Introduction

Constraint satisfaction problems (CSP) occur widely in engineering, science and the arts. Applications are frequently reported in production planning, resource allocation [BLN01], music composition [AADR98], Verification [EM97], Security [BB01] Bioinformatics [GW94] and many others. In fact, a CSP is any problem that can be expressed as that of finding, from a finite set of possibilities, a collection of values satisfying some given particular properties. These properties are represented by relations called constraints.

In its general setting the constraint satisfaction problem has been proved to be NP-complete. Nevertheless, in many real world instances a solution can be found with reasonable time and space efficiency when appropriate techniques are applied. The most frequently used are so-called *consistency* techniques. The main idea in these techniques is to use constraints not only to test the validity of a solution but as a sort of devices

* The contribution of Camilo Rueda was supported by the **AVISPA** Project.
** The contribution of Frank D. Valencia was supported by the **PROFUNDIS** Project.

for detecting inconsistencies and for pruning from the original set of possibilities some values that cannot appear in a solution. The reduced CSP taking into account only the remaining values is said to satisfy a given (weak) notion of consistency. One such notion is *arc consistency*. Consistency procedures are usually invoked repeatedly during search so it is very important to have efficient consistency algorithms. Even savings by a constant factor can have important overall performance consequences in some situations.

Finding better arc consistency algorithms has thus been an ongoing research topic for more than two decades. Building from algorithm AC-3 [Mac77], improvements such as AC-4 [MH86], AC-6 [Bes94], AC6++, AC-7 (see [BF99]) have been proposed. The standard way to compare arc consistency algorithms is by the number of constraint checks they perform. In [BR95] the so-called *four desirable properties* (FDP) for AC algorithms were defined and shown to provide a remarkable reduction in the number of constraint checks. Moreover, in [BR95] it is claimed that algorithms (such as AC6++ and AC-7) satisfying the FDP are optimal in the number of constraint checks.

Our contributions are the following: we show that even when complying with the FDP an AC algorithm can still perform unnecessary constraint checks (e.g., AC-6++ and AC-7). We thus refute the above optimality claim. We prove that there is a family of CSP's for which these unnecessary constraint checks can be rather significant. We also define a new property and show how AC algorithms satisfying it can avoid those redundant checks. This property is parameterized in a set of inference rules. We give two such rules and show their validity. We give a general AC algorithm taking into account the new property and show its correctness. We then use it to orthogonally extend AC-6++ and AC-7 into algorithms maintaining the new property and show how they improve over the originals in some benchmark and randomly generated problems.

Recently, [van02] has proposed a particular constraint processing ordering heuristic that can lead to savings of constraint checks similar to ours. Our idea is independent of constraint ordering and so leaves more room to variations in constraint ordering heuristics. This is important theoretically because the optimality claim for FDP compliant algorithms is wrt to analysis that assume the same particular constraint ordering. It is important in practice because a particular ordering may encode useful knowledge about the problem domain. On the other hand, efficient implementations of our idea seem to require particular value orderings, so it may leave less room to value ordering heuristics.

2 CSP and AC: Concepts, Assumptions and Concerns

A *Constraint Satisfaction Problem* (CSP) consists of a given finite set of variables with their corresponding finite domain of values, and a given set of constraints over the variables. The constraints specify allowed value assignments for the corresponding variables. A CSP *solution* is a value assignment for the variables satisfying all the constraints. Since CSP's are NP-complete [GJ79], usually they are simplified by using pre-processing techniques, most notably Arc-Consistency (AC). This technique, also used *during* the search of CSP's solutions, involves the removal of some values that cannot be in any solution.

In AC we are only concerned with binary constraints, so we confine ourselves to CSP's where all the constraints are binary relations; i.e., *binary CSP's*.

We can define a (binary) CSP as a tuple $\langle V, D, C\rangle$ where $V = \{x_1, \dots, x_n\}$ is a set of *variables*, $D = \{D_1, \dots, D_n\}$ is a set of *domains* with each D_i specifying the domain of $x_i \in V$, and C is a set of *constraints* $C_{ij} \subseteq D_i \times D_j$. We define the predicate $C_{ij}(v, w)$ to be true iff $(v, w) \in C_{ij}$. Without loss of generality, for each pair (x_i, x_j) of variables in V, *we assume that there is at most one constraint* $C_{ij} \in C$. A tuple $(v_1, \dots, v_n) \in D_1 \times \dots \times D_n$ is a *solution* iff for each $C_{ij} \in C$, $C_{ij}(v_i, v_j)$.

Example 1. Let $V = \{x_1, x_2, x_3\}$, $D = \{D_1, D_2, D_3\}$ with $D_1 = D_3 = \{1, 2\}$ and $D_2 = \{0, 1, 2\}$. Define $C_{12} = \{(v, w) \in D_1 \times D_2 \mid v \leq w\}$ and $C_{23} = \{(v, w) \in D_2 \times D_3 \mid v \leq w\}$. Consider the CSP $\langle V, D, \{C_{12}, C_{23}\}\rangle$. The tuples $(1, 1, 1), (1, 1, 2), (1, 2, 2), (2, 2, 2)$ are solutions, but no tuple having 0 as its second component (i.e., of the form $(\_, 0, \_)$) can be a solution. $\square$

2.1 Bidirectionality

Let $\langle V, D, C\rangle$ be a CSP. Notice that if $C_{ij} \in C$, augmenting the CSP with a constraint C_{ji} which is the *converse* of C_{ij} (i.e., $C_{ji} = C_{ij}^{-1} = \{(w, v) \mid (v, w) \in C_{ij}\}$) does not restrict any further the CSP, i.e., the CSP's solutions remain the same. Intuitively, C_{ij} and its converse C_{ji} represent exactly the same constraint except that C_{ij} can be viewed as a constraint going from x_i to x_j while C_{ji} as going from x_j to x_i. The reader may care to augment the CSP's constraints in Example 1 with the converses C_{21} and C_{32} and verify that the resulting CSP's solutions are the same as the ones to the original CSP.

If a CSP has a converse C_{ji} for each of its constraints C_{ij} then it is said to satisfy the *bidirectionality* property. Without loss of generality, we shall confine our attention to *CSP's satisfying the bidirectionality property* as usually done for AC.

2.2 Arc-Consistency and Viability

As mentioned before, the idea behind AC computation is to eliminate from the domains of a given CSP some values that cannot be in any of its solutions. We say that such values are not *viable*.

Definition 1 (Support and Viability). *Let $P = \langle V, D, C\rangle$ be a CSP where $D = \{D_1, \dots, D_n\}$. Let $D'_1 \subseteq D_1 \dots D'_n \subseteq D_n$. Suppose that $C_{ij} \in C$, $v \in D'_i$ and $w \in D'_j$.*

We say that w is a **support** *for v iff $C_{ij}(v, w)$. Also, we say that v is* **viable** *wrt D'_j iff there exists a support for v in D'_j. Furthermore, we say that v is* **viable** *wrt $D'_1 \times \dots \times D'_n$ iff for all $C_{ik} \in C$, v is viable wrt D'_k.*

Example 2. Let P be the CSP $\langle V, D, C\rangle$ with V and D as in Example 1 and C as the set containing the constraints C_{12} and C_{23} in Example 1 plus its converses C_{21} and C_{32}.

Notice that $2 \in D_2$ is a support for $1, 2 \in D_1$. Also notice that $0 \in D_2$ is not viable wrt D_1, so it cannot be in any solution to P. We shall see that in AC computation, $0 \in D_2$ must be removed. $\square$

The AC algorithms use a graph whose nodes and arcs correspond to the variables and constraints, respectively, of the input CSP. Given a CSP $P = \langle V, D, C \rangle$, define G_P as the graph with nodes $Nodes(G_P) = \{i \mid x_i \in V\}$ and arcs $Arcs(G_P) = \{(i, j) \mid C_{ij} \in C\}$. Let us recall the definition of arc-consistency:

Definition 2 (AC Graphs). *Let $P = \langle V, D, C \rangle$ be a CSP where $D = \{D_1, \ldots, D_n\}$ and let $D'_1 \subseteq D_1 \ldots D'_n \subseteq D_n$.*

An arc (i, j) in G_P is said to be **arc-consistent** *wrt D'_i and D'_j iff every $v \in D'_i$ is viable wrt D'_j. Also G_P is said to be* **arc-consistent** *wrt $D'_1 \times \ldots \times D'_n$ iff every arc (i, j) in G_P is arc-consistent wrt D'_i and D'_j.*

Furthermore, we say that G_P is **maximal arc-consistent** *wrt $\rho = D'_1 \times \ldots \times D'_n$ iff G_P is arc-consistent wrt ρ and there are no $D''_1 \supset D'_1, \ldots, D''_n \supset D'_n$ such that G_P is arc-consistent wrt $D''_1 \times \ldots \times D''_n$.*

Example 3. Let $P = \langle V, D, C \rangle$ as in Example 2. Notice that G_P is not arc-consistency wrt $D_1 \times D_2 \times D_3$ but it is wrt $\emptyset \times \emptyset \times \emptyset$. Verify that G_P is maximal arc-consistent wrt $D_1 \times D'_2 \times D_3$ where $D'_2 = D_2 - \{0\}$. □

Computing Arc-Consistency.

Given a $P = \langle V, D, C \rangle$ where $D = \{D_1, \ldots, D_n\}$, the outcome of an AC algorithm on input P, is a $P' = \langle V, D', C \rangle$ with $D' = \{D'_1, \ldots, D'_n\}$, $D'_k \subseteq D_k$ $(1 \leq k \leq n)$ such that G_P is maximal arc-consistent wrt $D'_1 \times \ldots \times D'_n$.

Usually, an AC algorithm takes each arc (i, j) of G_P and removes from D_i those values that are not viable wrt D_j (i.e., not having support in D_j). This may cause the viability of some values, previously supported by the removed ones from D_i, to be checked again by the algorithm.

Constraint Checks. The standard comparison measure for the various AC algorithms is the *number of constraint checks* performed (i.e., checking whether $C_{ij}(v, w)$ for some C_{ij} and $v \in D_i$, $w \in D_j$) [Bes94,BR95,BFR95]. It has been shown analytically and experimentally [BFR95] that if we assume a large cost per constraint check or *demonstrate* large enough savings in the number of constraint checks, the constraint checks count will dominate overhead concerns.

In the next section we shall see several properties aiming at reducing substantially the number of constraint checks from simple but useful observations.

Domain Ordering $\prec$. Henceforth, we presuppose a total underlying order $\prec$ on the CSP's domains as typically done for AC computation [Bes94,BR95,BFR95]. In practice, $\prec$ corresponds to the ordering on the data structure representing the domains. In our examples, we shall take $\prec$ to be the usual "less" relation $<$ on the natural numbers.

We can now recall the general notion of support lower-bound. Such a notion denotes a value before which no support for a given value can be found.

Definition 3 (Support Lower-Bound). *Let $P = \langle V, D, C \rangle$ be a CSP where $D = \{D_1, \ldots, D_n\}$ and let $D'_1 \subseteq D_1 \ldots D'_n \subseteq D_n$. For all $C_{ij} \in C$, the value $w \in D'_j$ is a* **support lower-bound** *in D'_j for $v \in D'_i$ iff for every $w' \in D'_j$ with $w' \prec w$, $C_{ij}(v, w')$ does not hold.*

Example 4. Let P be as in Example 2. Assume that the total ordering $\prec$ on the domains is $<$. Then $1 \in D_2$ is a support lower-bound in D_2 for $1, 2 \in D_1$. $\square$

(Notice that a support lower-bound for v is not necessarily a support of v.)

In the next sections, we shall see that simple and general notions, such as support lower-bound and bidirectionality (which are usually assumed in AC), can reduce substantially the number of constraints checks.

3 Four Desirable Properties of AC computation

Modern AC algorithms satisfy the so-called *four desirable properties* of a AC computation given in [BR95,BFR95]. These are very simple and practical properties aiming at reducing the number of constraint checks while keeping a reasonable space complexity. In practice, algorithms satisfying these properties have shown to be very successful [BR95,BFR95].

In the following we assume that $D_1, \dots, D_n$ represent the current CSP domains of during AC computation. The desirable properties require (of an AC algorithm) that:

1. $C_{ij}(v, w)$ should not be checked if there is w' still in D_j such that $C_{ij}(v, w')$ was already successfully checked.
2. $C_{ij}(v, w)$ should not be checked if there is w' still in D_j such that $C_{ji}(w', v)$ was already successfully checked.
3. $C_{ij}(v, w)$ should not be checked if:
   a. $C_{ij}(v, w)$ was already checked, or
   b. $C_{ji}(w, v)$ was already checked.
4. The space complexity should be $O(ed)$ where e, d are the cardinalities of the set of constraints and the largest domain, respectively, of the input CSP.

The properties can be justified as follows. An AC algorithm checks $C_{ij}(v, w)$ when establishing the viability of v wrt D_j (i.e., the algorithm needs to find a support for v in D_j if any, otherwise it should remove v from D_i). Now, the value v in (1) has already a support, i.e., it is viable, if such a w' still exists in D'_j ; so there is no need to check whether $C_{ij}(v, w)$. Property (2) can be explained similarly by using bidirectionality. Property 3.a states that there is no need of doing the same constraint check more than once, and 3.b states that, by bidirectionality, if we have checked $C_{ji}(w, v)$ then we already know the result of checking $C_{ij}(v, w)$. Property (4) states a restriction on the space that can be used (see [BR95] for further details).

The AC algorithm AC-3 does not satisfy Properties (1-3); AC-4 does not satisfy Properties 1,2,3.b, and 4; AC-6 does not satisfy Properties 2 and 3b (the ones using bidirectionality); AC-Inference does not comply with Property 4. The modern algorithms AC6++ and AC-7 preserve the four properties and hence they are said to be optimal [BR95].

The AC6++ and AC-7 algorithms differ mainly in the order that values and arcs are considered during AC computation. The latter propagates the effects of removing a value as soon as possible (i.e., to reconsider the viability of the values supported by the removed one). In practice, this heuristic seems to save unnecessary constraint checks. Experimentally, AC-7 has shown to outperform AC6++.

In [BR95] it is also claimed that the four desirable properties guarantee a minimal number of constraint checks. This claim is in the context of CSP's where nothing is known about the particular semantics of the constraints and wrt the order in which values, variables and arcs are considered during AC computation. Hence, AC6++ performs a minimal number of constraint checks according to the order used by this algorithm, but still it may perform more constraint checks than AC-7 which uses a different order.

The above four properties are of important practical significance. Nevertheless, we believe that they are not enough to guarantee the minimal number of constraint checks, thus contradicting the claim mentioned above. In the next section, we shall show that even when complying with the four desirable properties, an AC algorithm can still perform a substantial number of unnecessary constraint checks.

4 New Desirable Property and and Non-Viability Deductions

A drawback of the four desirable properties is that they allow checking $C_{ij}(v, w)$ even when the non-viability of v or w could have been deduced by using only the general notions of bidirectionality and support lower-bound, and information about previous constraint checks —i.e., without using any particular semantic properties of the constraints under consideration. In our view, the check of $C_{ij}(v, w)$ under the above conditions would be unnecessary.

Let us illustrate the above with the following example:

Example 5. Let P be the CSP defined in Example 4. Suppose that during AC computation an algorithm satisfying the four desirable properties checks, first of all, the viability of the values in D_1 and immediately after the viability of the values in D_3. Furthermore, suppose that the search for support in D_2 is done according to $\prec$.

After establishing the viability of all the values in D_1, the algorithm has checked that for no value $v \in D_1$, $C_{12}(v, 0)$ holds. Similarly after establishing the viability of the values in D_3, the algorithm has checked that for no value $w \in D_3$, $C_{32}(w, 0)$ holds.

Nevertheless, notice that for any $w \in D_3$ checking $C_{32}(w, 0)$ is really unnecessary, because after checking for the viability of the values in D_1 one can deduce that $0 \in D_2$ is not viable.

Here is a proof of the non-viability of $0 \in D_2$: Recall from Example 4 that $1 \in D_2$ is a support lower-bound in D_2 for $1, 2 \in D_1$. Now $0 \prec 1$, so after checking for the viability of $1, 2 \in D_1$, we can conclude from Definition 3 that $\neg C_{12}(1, 0)$ and $\neg C_{12}(2, 0)$. By bidirectionality $\neg C_{21}(0, 1)$ and $\neg C_{21}(0, 2)$. Hence we can *deduce*, from Definition 1, that $0 \in D_2$ is not viable. □

4.1 Unnecessary Constraint Checks

One can verify that both AC6++ and AC-7 may indeed perform the unnecessary constraint checks mentioned in the above example. Also notice that the number of unnecessary constraint checks in the above example is $d = |D_3|$. However, as shown below, one can generalize Example 5 to a family of CSP's for which the numbers of unnecessary constraint checks is about ed^2, where e is the number of constraints and d is the size of the largest domain.

In the following theorem, by *unnecessary constraint check* we mean that the check can be avoided by using only bidirectionality, the notion of support lower-bound, and information about previous constraint checks.

Theorem 1. *There is family of CSP's for which the number of unnecessary constraint checks during AC computation, even when complying with the four desirable properties, can be $\Omega(ed^2)$, where e is the number of constraints and d the size of the largest domain.*

Proof (Outline). Let $P = \langle V, D, C \rangle$ where $D = \{D_1, \ldots, D_n\}$ with all the domains being of a same even size. Decree that for any $(i, j) \in Arcs(G_P)$, $i < j$, the first half of the values in D_j (according to the domain ordering $\prec$) are not viable wrt D_i.

Let us suppose that we have an AC algorithm satisfying the four desirable properties. Assume that the algorithm checks first of all the viability of the values in D_1 wrt D_n (i.e., it searches supports in D_n for the values in D_1), then D_2 wrt D_n , $\ldots$, D_{n-1} wrt D_n, and then D_1 wrt D_{n-1}, D_2 wrt D_{n-1} and so on. Furthermore, suppose that the search for support is done according to $\prec$.

After establishing the viability of the values in D_1, it is possible to deduce, by using the notion of bidirectionality and support lower-bound (as in Example 5), that the values in the first half of D_n are not viable. Now, for each $k = 2, \ldots, n-1$, the four desirable properties *do not prevent* the algorithm from checking unnecessarily $C_{kn}(v, w)$ for every $v \in D_k$ and every w in the first half of D_n. The same happens for $k = 2, \ldots, n-2$ wrt D_{n-1}, and so on. It then follows that the algorithm can perform $\Omega(ed^2)$ unnecessary constraint checks. □

4.2 New Desirable Property.

In order to avoid unnecessary constraint checks of the kind above, we could suggest the following new desirable property: $C_{ij}(v, w)$ should not be checked if it can be deduced via bi-directionality and the notion of support-lower bound that v or w is not viable. We shall use "deduce" in a loose sense of the word: We mean that one can conclude, without performing further constraint checks, that v (or w) is not viable.

Nevertheless, there could be many other ways of deducing non-viability (e.g., special properties of constraints, domains, etc). Hence, we find it convenient to define the new desirable property wrt to fixed *non-viability deduction system* $\mathcal{S}$; i.e, a set of *inference rules* that allows us to deduce the non-viability of some values. We assume that whether a given value can be deduced in $\mathcal{S}$ as non-viable can always be decided. The fifth desirable property wrt a fixed $\mathcal{S}$ can be stated as follows:

5. $C_{ij}(v, w)$ should not be checked if it can be deduced, in the underlying non-viability inference system $\mathcal{S}$, that v or w are not viable.

Of course some deduction systems may be of little help. For example if $\mathcal{S}$ is the empty set of rules, in which case both AC-6++ and AC-7 would trivially satisfy the fifth property. Other example is a deduction system in which deciding the non-viability of a given value cannot be done with $O(ed)$ in space —see the fourth desirable property. Next we give more helpful but general deduction systems (inference rules).

Non-Viability Deductions

In the following properties, we give two simple and general inference rules for non-viability deduction to avoid unnecessary constraint checks of the kind illustrated in Example 5 and stated in Theorem 1.

Property 1 (Support LOWEST-Bound). Let $P = \langle V, D, C \rangle$ be a CSP where $D = \{D_1, \ldots, D_n\}$ and let $D'_1 \subseteq D_1 \ldots D'_n \subseteq D_n$. Suppose that SLB_{ij} is the **least** value (wrt $\prec$) in a given set containing a support-lower bound in D'_j for each $v \in D'_i$. The following is a valid non-viability inference rule:

$$\text{If } w \in D'_j \text{ and } w \prec SLB_{ij} \text{ then } w \text{ is not viable wrt } D'_i.$$

Proof. By using the notions of support lower-bound (Definition 3) and bidirectionality. □

The above property says that a value can be deduced as non-viable if it is less than every support-lower bound for (all the values of) a given domain. The property can be implemented by using an array SLB such that $SLB[i, j]$ keeps the least support-lower bound in D_j for the values in D_i. We shall discuss this issue in Section 5.1.

Property 2 (Support Upper-Bound Cardinality). Let $P = \langle V, D, C \rangle$ be a CSP where $D = \{D_1, \ldots, D_n\}$ and let $D'_1 \subseteq D_1 \ldots D'_n \subseteq D_n$. Suppose that $sub_{ij}(v)$ denotes an upper bound on the number of supports of $v \in D'_i$ in D'_j. The following is a valid non-viability inference rule:

$$\text{If } sub_{ij}(v) = 0 \text{ then } v \text{ is not viable wrt } D'_j.$$

Proof. Immediate □

We can implement the above property by having counters of the form $sub_{ij}(v)$ initially set to $|D_j|$. Then counter $sub_{ij}(v)$ decreases each time a check of $C_{ij}(v, w)$ is found to be false, a support $w' \in D_j$ for v is eliminated, or some value supported by v is eliminated. Once $sub_{ij}(v) = 0$ we can proceed as if v did not exist in D_i. We shall discuss this in Section 5.1.

In the next sections we shall also illustrate experimentally that despite its simplicity, the above deduction rules indeed provide a substantial reduction in the number of constraint checks for CSP's where nothing is known about the particular semantics of the constraints.

5 AC Algorithms with Non-Viability Deductions

In this section we first present a new generic AC algorithm, here called AC[$\mathcal{S}$], which is parametric in an underlying non-viability deduction system $\mathcal{S}$. The algorithm is based on AC-5 and it can be instantiated to produce other AC algorithms such as AC-4, AC-5, AC-3, AC6++ and AC-7.

The generic AC algorithm removes the values deduced as being non-viable immediately. This can be justified as follows: If propagating the consequences of removing a value as soon as possible is a good heuristic (as shown by AC-7 [BFR95]) then it is reasonable to perform removals as soon as possible. The non-viability deductions can also help to detect promptly values that must be removed.

In the following we assume that $P = \langle V, D, C\rangle$ represents the CSP on input of which AC[$\mathcal{S}$] is to perform AC. Furthermore, we assume that $D_1, \dots, D_n$ represent the current CSP's domains during the AC computation.

Most AC algorithms use a waiting list Q containing elements that have been removed and for which we need to propagate the effects of their elimination. In AC[$\mathcal{S}$], Q contains elements of the form $\langle(i,j), w\rangle$, where (i,j) is an arc and w is value which has been removed from D_j, thus making us reconsider the viability of some values in D_i supported by w.

As AC-5, AC[$\mathcal{S}$] is parametric in the procedures ArcCons and LocalArcCons (Figure 1) whose implementation can give rise to various AC algorithms. The procedure ArcCons$(i, j, \Delta_i, \Delta_j)$ computes the set of values $\Delta_i \subseteq D_i$ without support in D_j and the set of values $\Delta_j \subseteq D_j$ deduced, wrt $\mathcal{S}$, as being non-viable. The procedure LocalArcCons$(i, j, w, \Delta_i, \Delta_j)$ is similar except that it computes the set of values $\Delta_i \subseteq D_i$ without support in D_j which were previously supported by a value w removed from D_j.

```
**procedure** ArcCons(**in** $i, j,$ **out** $\Delta_i, \Delta_j$)
 Pre: $(i,j) \in Arcs(G_P)$
 Post: $\Delta_i = \{v \in D_i \mid \forall w \in D_j : \neg C_{ij}(v,w)\}$
       $\Delta_j = \{w \in D_j \mid P \vdash_{\mathcal{S}} w$ is not viable $\}$

**procedure** LocalArcCons(**in** $i, j, w,$ **out** $\Delta_i, \Delta_j$)
 Pre: $(i,j) \in Arcs(G_P) \wedge w \notin D_j$
 Post: $\Delta_i = \{v \in D_i \mid C_{ij}(v,w) \wedge \forall w' \in D_j : \neg C_{ij}(v,w')\}$
       $\Delta_j = \{w' \in D_j \mid P \vdash_{\mathcal{S}} w'$ is not viable $\}$
```

Figure1. The ArcCons and Local ArcCons Procedures. Notation $P \vdash_{\mathcal{S}} E$ means that E can be deduced in the inference system $\mathcal{S}$ from the current information about P

The AC[$\mathcal{S}$] algorithm (see Figure 2) has two phases. In the first one, called initialization phase (Lines 1-7), AC[$\mathcal{S}$] enforces each arc (i,j) to be arc-consistent wrt to the current D_i and D_j. In the second one, called propagation phase (Lines 8-15), it propagates the effects of all the removed values. Notice that the removed values are put in Q and they stay in there until the effects of their elimination are propagated.

The following theorem states that the outcome of AC[$\mathcal{S}$] on a CSP $P = \langle V, D, C\rangle$ where $D = \{D_1, \dots, D_n\}$, is a CSP $P' = \langle V, D', C\rangle$ with $D' = \{D'_1, \dots, D'_n\}$, $D'_k \subseteq D_k$ ($1 \leq k \leq n$) such that G_P is maximal arc-consistent wrt $D'_1 \times \dots \times D'_n$.

Theorem 2 (Correctness of AC[$\mathcal{S}$]). *The algorithm AC[$\mathcal{S}$], Figure 2, is correct wrt its precondition and postcondition.*

```
**Algorithm** AC[$\mathcal{S}$] (**in-out** $P$)
 Pre: $P$ is a CSP $\langle V, D, C\rangle$ with $D = \{D_1, \ldots, D_n\}$
 Post: $G_{P_0}$ is maximal arc-consistency wrt $D_1 \times \ldots \times D_n$
1. $Q \leftarrow \emptyset$
2. for each $(i,j) \in Arcs(G_P)$ do
3.   ArcCons$(i, j, \Delta_i, \Delta_j)$
4.   $Q \leftarrow Q \cup \{\langle (k,i), v\rangle \mid (k,i) \in Arcs(G_P) \wedge v \in \Delta_i\}$
5.   $Q \leftarrow Q \cup \{\langle (k,j), w\rangle \mid (k,j) \in Arcs(G_P) \wedge w \in \Delta_j\}$
6.   $D_i \leftarrow D_i - \Delta_i$
7.   $D_j \leftarrow D_j - \Delta_j$
8. while $Q \neq \emptyset$ do
9.   choose $\langle (i,j), w\rangle \in Q$
10.  LocalArcCons$(i, j, w, \Delta_i, \Delta_j)$
11.  $Q \leftarrow Q - \{\langle (i,j), w\rangle\}$
12.  $Q \leftarrow Q \cup \{\langle (k,i), v\rangle \mid (k,i) \in Arcs(G_P) \wedge v \in \Delta_i\}$
13.  $Q \leftarrow Q \cup \{\langle (k,j), w\rangle \mid (k,j) \in Arcs(G_P) \wedge w \in \Delta_j\}$
14.  $D_i \leftarrow D_i - \Delta_i$
15.  $D_j \leftarrow D_j - \Delta_j$
```

Figure2. The generic AC[$\mathcal{S}$] algorithm. Notation P_0 denotes the CSP P when input to the algorithm.

Proof (Outline). Suppose that the AC[$\mathcal{S}$] algorithm runs on input $P = \langle V, C, D\rangle$ with $D = \{D_1, \ldots, D_n\}$. Let $\rho_f = D_{1_f} \times \ldots \times D_{n_f}$ be such that G_P is maximal arc-consistency wrt ρ_f.

Let D_{i_0} be the initial D_i and D_{i_k} with $k > 0$ be the current D_i after the k-th elimination of a Δ_m (Lines 6-7 and 14-15) from some $D_{m_{k-1}}$. Let $\rho_k = D_{1_k} \times \ldots \times D_{n_k}$. It is sufficient to prove that AC[$\mathcal{S}$] terminates with a final ρ_k , $k \geq 0$, such that $\rho_k = \rho_f$.

From the specification of ArcCons and LocalArcCons, it is easy to verify that any value is removed from $D_{i_{k-1}}$ only if it is found non-viable wrt ρ_{k-1}; either it did not have a support or it was deduced in $\mathcal{S}$ as being non-viable. Now one can prove by induction on k that if $v \notin D_{i_k}$ then $v \notin D_{i_f}$. So, the first invariant of AC[$\mathcal{S}$] is the following:

$$\rho_f \subseteq \rho_k \subseteq \rho_{k-1} \subseteq \ldots \subseteq \rho_0. \qquad (1)$$

Also, from the specification of ArcCons and LocalArcCons, one can verify that after the initialization phase (Lines 1-8) every value has a support in the current domains or in the waiting list Q. More precisely, let $Val(Q)$ be the set of domain values appearing in Q; during the propagation phase, the second invariant of AC[$\mathcal{S}$] is:

$$\forall (i,j) \in G_P, \forall v \in D_{i_k}, \exists w \in D_{j_k} \cup Val(Q) : C_{ij}(v, w). \qquad (2)$$

The algorithm terminates when $Q = \emptyset$. Hence, from Definition 2 and the second invariant, G_P is arc-consistency wrt the ρ_k at termination time. Furthermore, from the first invariant we have $\rho_f \subseteq \rho_k$. It then follows that G_P is maximal arc-consistent wrt the ρ_k at termination time, as wanted.

It only remains to prove termination; i.e., that the propagation phase terminates (Lines 8-15). Observe that once an element $\langle (i,j), w\rangle$ is taken from Q it is never put back in Q. In each iteration in the propagation phase, an element is taken from Q. Furthermore, there can be no more than ed elements in Q, where $e = |C|$ and d is the size of the largest domain. Hence, ed is an upper-bound on the number of iterations of the propagation phase. □

5.1 Implementation: AC6-3+ and AC-7+

We have implemented the non-viability inference rules given in Properties 1 and 2 for AC6++ and AC-7. We call AC6-3+ the algorithm that (orthogonally) extends AC6++ with the support *lowest*-bound inference rule (Property 1) and AC-7+ the one that extends (orthogonally) AC-7 with the support upper-bound cardinality rule (Property 2).

The algorithm AC6-3+ has a three-dimensional array slb used exactly as in AC6++. Each entry $slb[i,j,v]$ represents a support *lower*-bound for $v \in D_i$ in D_j—in fact the greatest one so far found by the algorithm; see [BR95] for more details. In addition, AC6-3+ has a two-dimensional array SLB. Each entry $SLB[i,j]$ keeps the *least* of all $slb[i,j,v]$ for all $v \in D_i$. Justified by Property 1, every value in $w \in D_j$ less than $SLB[i,j]$ is removed from D_j before checking any constraint of the form $C_{kj}(u,w)$ or $C_{kj}(w,u)$.

As for AC-7+, we use an additional three-dimensional array sub. Each array entry $sub[i,j,v]$ represents a lower-bound on the number of supports in D_j for $v \in D_i$. Initially, each $sub[i,j,v]$ is set to $|D_j|$. Then the algorithm decreases $sub[i,j,v]$ each time $C_{ij}(v,w)$ is found to be false, a support $w' \in D_j$ for v is eliminated, or some value supported by v is eliminated. Justified by Property 2, v is removed from D_i whenever $sub[i,j,v]$ becomes zero.

Both extended algorithms have the same worst-case complexities of their predecessors AC6++ and AC-7. More precisely, both AC6++ and AC-7+ have $O(ed^2)$ worst-case time complexity and $O(ed)$ worst-case space complexity, where e is the number of constraints and d the size of the largest domain [BR95,BFR95]. They also satisfy the four desirable properties as a result of being orthogonal extensions of AC6++ and AC-7. Furthermore, they satisfy the new desirable property wrt their underlying non-viability deduction rules, thus they can save some unnecessary constraint checks. In the next section, we shall show experimental evidence of these savings.

6 Experimental Results

Here we show some of our experimental results obtained from CSP's typically used to compare AC algorithms [Bes94,BR95,BFR95]. We compared AC6++ vs AC6-3++ and AC-7 vs AC-7+ in benchmark CSP's [Van89] as well as randomly generated CSP's [Bes94,BR95,BFR95]. Each comparison was performed wrt fifty instances of each problem.

For the ZEBRA problem [Van89] we obtained the following results in terms of constraint checks (ccs):

AC6++ : 717 ccs	AC-7 : 640 ccs
AC6-3+ : 639 ccs	AC-7+ : 594 ccs

As for the combinatorial problem suggested in [Van89], we obtained:

AC6++ : 977 ccs	AC-7 : 966 ccs
AC6-3+ : 783 ccs	AC-7+ : 826 ccs

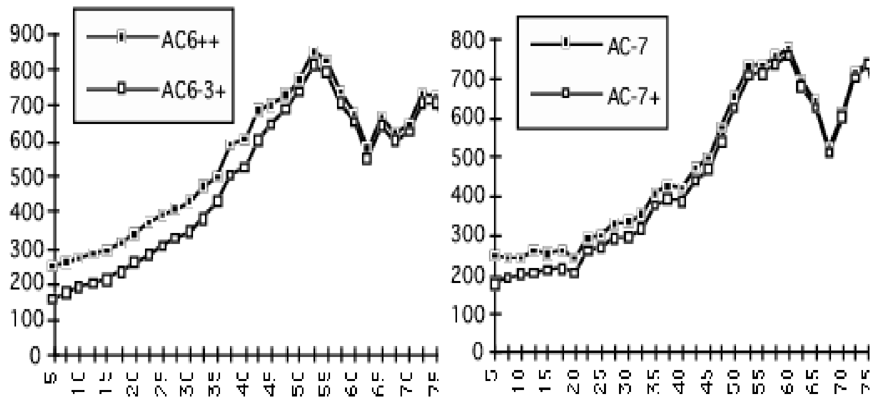

Figure3. AC6++ vs AC6-3+ (left) and AC-7 vs AC-7+ (right) on random generated problems with 20 variables, at most 5 values per domain, and 30% probability of having a constraint between two variables. The horizontal axis represents the probability percentage that two values support each other. The vertical axis represents the number of constraint checks.

For the randomly generated problems, following [Bes94,BR95,BFR95], we took the following as parameters of the generation: the number of variables, the size of the domains, the probability of having a constraint between any two variables, and the probability for any two values to be support of each other. In Figure 3 we show some results corresponding to the values of the parameters used in experiments of [BR95]. On average, we obtained that the reduction in the number of constraint checks by AC6-3+ and AC-7+ wrt AC6++ and AC-7 (respectively), was about 10%. We also observed that the numbers of values deduced as being non-viable was proportional to the reduction in the number of constraint checks. Moreover, even when the number of non-viability deductions was small, the number of constraint checks was significantly reduced.

7 Concluding Remarks

We have shown that, despite providing a remarkable reduction in the number of constraint checks, the four desirable properties of AC computation still allow a substantial number of unnecessary constraint checks—in the sense that the checks could have been avoided by deducing, only from general constraint properties and previous constraint-checks, the non-viability of some values. We also suggested a new desirable property which provides a further substantial reduction in the number of constraint checks. We modified some of the best known AC algorithms to satisfy the property and showed experimentally the benefits of the modified algorithms.

Since the reduction in the number of constraint checks by the new property depends on the non-viability of values, we believe it is practical for problems with strong structural properties (i.e., strong constrains, large domains, etc). As future work, we plan to identify and implement more inference rules to deduce non-viability efficiently.

Acknowledgements. Many thanks to Christian Bessière and Marc van Dongen for helpful comments on this work.

References

[AADR98] C. Agon, G. Assayag, O. Delerue, and C. Rueda. Objects, time and constraints in openmusic. In *ICMC98*, pages 1–12. ICMA, 1998.

[BB01] G. Bella and S. Bistarelli. Soft constraints for security protocol analysis: Confidentiality. *LNCS*, 1990, 2001.

[Bes94] C. Bessiére. Arc-consistency and arc-consistency again. *Artificial Intelligence*, 65(1):179–190, 1994.

[BF99] C. Bessiére and E. C. Freuder. Using constraint metaknowledge to reduce arc consistency computation. *Artificial Intelligence*, 107(1):125–148, 1999.

[BFR95] C. Bessière, E. C. Freuder, and J. C. Régin. Using inference to reduce arc consistency computation. In *ICAI-95*, pages 592–599, 1995.

[BLN01] P. Baptiste, C. Le Pape, and W. Nuijten. *Constraint-Based Scheduling. Applying Constraint programming to Scheduling Problems*. Kluwer, 2001.

[BR95] C. Bessière and J. C. Régin. Using Bidirectionality to Speed Up Arc-Consistency Processing. In *Constraint Processing, Selected Papers*, volume 923 of *LNCS*, pages 157–169. Springer-Verlag, 1995.

[EM97] J. Esparza and S. Melzer. Model checking LTL using constraint programming. In *18th International Conference on Application and Theory of Petri Nets*, volume 1248, pages 1–20. Springer-Verlag, 1997.

[GJ79] R. Garey and D.S. Johnson. *Computers and Intractability. A Guide to the Theory of NP-Completeness*. W. H. Freeman and Company, 1979.

[GW94] C. Gaspin and E. Westhof. The determination of secondary structures of RNA as a constraint satisfaction problem. In *Advances in molecular bioinformatics*. IOS Press, 1994.

[Mac77] A. K. Mackworth. Consistency in Networks of Relations. *Artificial Intelligence*, 8:99–118, 1977.

[MH86] R. Mohr and T. C. Henderson. Arc and path consistency revisited. *Artificial Intelligence*, 28(2):225–233, 1986.

[Van89] P. Van Hentenryck. *Constraint Satisfaction in Logic Programming*. MIT Press, Cambridge, MA, 1989.

[van02] M.R.C. van Dongen. Ac-3d an efficient arc-consistency algorithm with a low space-complexity. In *CP'2002*, LNCS, pages 755–760. Springer-Verlag, 2002.