Using Constraint Metaknowledge to Reduce Arc Consistency Computation

Christian Bessière
LIRMM-CNRS (UMR 5506), 161 rue Ada
34392 Montpellier cedex 5, France
e-mail: bessiere@lirmm.fr

Eugene C. Freuder
University of New Hampshire
Durham, NH 03824 USA
e-mail: ecf@cs.unh.edu

Jean-Charles Régin
ILOG, 1681 route des Dolines
06560 Valbonne, France
e-mail: regin@ilog.fr

Abstract

Constraint satisfaction problems are widely used in artificial intelligence. They involve finding values for problem variables subject to constraints that specify which combinations of values are consistent. Knowledge about properties of the constraints can permit inferences that reduce the cost of consistency checking. In particular, such inferences can be used to reduce the number of constraint checks required in establishing arc consistency, a fundamental constraint-based reasoning technique. A general AC-Inference algorithm schema is presented and various forms of inference discussed. A specific algorithm, AC-7, is presented, which takes advantage of a simple property common to all binary constraints to eliminate constraint checks that other arc consistency algorithms perform. The effectiveness of this approach is demonstrated analytically, and experimentally.

1 Introduction

Constraint satisfaction problems (*CSPs*) occur widely in artificial intelligence. They involve finding values for problem variables subject to constraints on which combinations are acceptable. For simplicity (and as CSP algorithms have been commonly introduced) we restrict our attention here to *binary* CSPs, where the constraints involve two variables. However, the ideas presented here can be extended to non-binary constraints. First results concerning this topic can be found in [BR97].

Binary constraints are binary relations. If a variable I has a *domain* of potential values D_I and a variable J has a domain of potential values D_J, the constraint on I and J, R_{IJ}, is a subset of the Cartesian product of D_I and

D_J. If the pair of values a for I (denoted by (I,a)) and b for J (denoted by (J,b)) is acceptable to the constraint R_{IJ} between I and J, i.e., a member of the Cartesian product, we will call the values *consistent* (with respect to R_{IJ}). Asking whether a pair of values is consistent is called a *constraint check*.

A value, (I,a), is *arc inconsistent* if there is a variable J for which there is no value in D_J consistent with (I,a) with respect to R_{IJ}. We achieve *arc consistency* by removing *all* arc inconsistent values, and *only* arc inconsistent values in order not to affect the set of satisfiable assignments of the CSP. The set of domains so obtained is called *maximal arc consistent* set of domains (or *arc consistent closure*), and is denoted by $AC(\mathcal{D})$ if $\mathcal{D} = \{D_I, D_J, \ldots\}$ was the set of domains of the variables of the CSP. If a domain *wipe out* is observed when achieving arc consistency, we say that the CSP is arc inconsistent.

In the rest of the paper, d will represent the size of the largest initial domain, and e the number of constraints of the CSP.

Constraints can be represented *implicitly*, where a computation, or a real-world process, is needed to answer constraint check questions, or *explicitly*, where the answer is already recorded in a data base, e.g., a Boolean matrix representation of a constraint R_{IJ}, where the matrix entry $R_{IJ}(a,b)$ is true if and only if the combination of a and b is acceptable to R_{IJ}. Implicit constraint checks could be very costly to compute. Even if the constraints are represented explicitly in a form that permits quick computation, there may be an enormous number to compute, along with associated decisions about which checks to do and how to use the results. As a result much of the work on constraint reasoning has focused on ways to reduce the number of constraint checks required.

Constraint algorithms often seek to establish *support* for a value $a \in D_I$, i.e., to find a value for a variable J that is consistent with (I,a) (or to determine that no such value exists). Traditionally constraint checks have been used to establish support. This paper proposes to reduce constraint checks by using metalevel knowledge to *infer* support. We will demonstrate that one such inference can save many additional constraint checks, ensuring that the benefits of these inferences can more than offset the costs associated with making and exploiting them.

We apply this approach to building a schema for arc consistency algorithms. Arc consistency is one of the most basic and useful constraint reasoning processes. Thus arc consistency algorithms have been the subject of much interest [MF93]. The new algorithm schema, AC-Inference, permits use of inferred support. We identify several properties of constraints that permit such inferences. We hope that AC-Inference will permit the exploitation of many other generic and problem domain specific properties of constraints. Furthermore, the metaknowledge inference approach should also be extensible to higher order consistency [Fre78].

We refine the schema to build a specific, new arc consistency algorithm AC-7. AC-7 is a *general* arc consistency algorithm, since it does not depend on special properties of a limited class of constraints, but simply utilizes the knowledge that support is *bidirectional*: (I,a) supports (J,b) if and only if (J,b) supports (I,a). (It is tempting to assume that a special class of undirected constraints is required here; but a careful reading of [Mac77] should demonstrate that this is not the case.)

AC-7's exploitation of bidirectionality gives it a computational advantage over other general purpose arc consistency algorithms, and restricting attention to bidirectionality also permits an implementation with space efficiency comparable to the best of these algorithms, which is not possible for the AC-Inference algorithm schema in general. Constraint programmers tell us that space can be a major issue for practical applications. On the other hand, AC-Inference permits us to obtain additional computational efficiency by taking advantage of knowledge about a restricted class of constraints. We demonstrate each of these advantages experimentally in real-world applications. We also provide analytical evaluation of AC-7, specifically refuting an optimality claim for an earlier algorithm, AC-6 [BC93].

We emphasize that the saving that AC-7 can achieve is not simply to avoid *inverse* checks, checking that (I, a) supports (J, b) when we have already checked that (J, b) supports (I, a). AC-7 may avoid *non-inverse* checks: it may never check whether (I, a) and (J, b) are consistent, while other algorithms do. We will show that there are problems for which AC-7 can avoid a quadratic number of such checks.

We compare AC-7 with AC-3, AC-4, and AC-6. (The AC-5 algorithms achieve their advantages for specific classes of constraints.) In terms of constraint checks, AC-7 is capable of considerable improvement over AC-3, AC-4, and AC-6.

Of course, constraint checks alone do not tell the whole story. However, if we either assume a large enough cost per constraint check, or demonstrate a large enough savings in the number of constraint checks, the constraint check count will dominate overhead concerns.

In summary, the contributions of this paper are:

- a new approach to utilizing constraint metaknowledge: inferring support
- an algorithm schema, AC-Inference, that supports this approach
- a general-purpose arc consistency algorithm, AC-7, that exploits this approach
- analytical evidence of the advantages of AC-7, specifically refuting the optimality of AC-6
- identification of additional, specific constraint properties that permit additional inferences

Section 2 recalls related work. Section 3 discusses the general theme of meta-level inference and presents the general algorithm schema. Section 4 focuses on the specific AC-7 algorithm. In Section 5, an experimental evaluation is given.

2 Relation to Previous Work

For some time the state of the art resided in two algorithms, AC-4 [MH86], which has optimal worst-case behavior, and AC-3 [Mac77], which often exhibits better average-case behavior [Wal93]. Two AC-5 algorithms, one by Deville, Van Hentenryck and Teng [DV91, VDT92] and another by Perlin [Per92], permit exploitation of certain specific constraint structures, but reduce to AC-3 or AC-4 in the general case. Bessière and Cordier developed AC-6, which retains the optimal worst-case behavior of AC-4 while improving on the average-case behavior of AC-3 [BC93, Bes94]. The new algorithm schema, AC-Inference, owes something to all these predecessors, but permits use of inferred support; AC-7 is most closely related to AC-6. This paper merges and extends previous independent work [BR95, Fre95] and common work [BFR95] by the authors.

The potential redundancy in processing of bidirectional support has been recognized before. When AC-3 removes a value from the domain of variable X because it has no support in variable Y, it realizes that this cannot cause a value of Y to become bereft of support at X. DEEB [Gas78] uses a "revise-both" procedure that more directly anticipates AC-7. After the values for X are checked for support at Y, values for Y are immediately checked for support at X, but only those Y values that have not just provided support for X values are checked. Gaschnig points out that this avoids unnecessary checks performed by AC-3 the first time the domains of X and Y are checked against each other; but he incorrectly, it seems to us, concludes that DEEB also avoids all the checks that AC-3 avoids by utilizing bidirectionality.

Neither algorithm, however, has any long term memory of inferences based on bidirectionality. For example, suppose that the first value, x_1, and last value, x_{100}, among a hundred values for X are found to be supported by a value y for Y. If later x_1 is deleted during the constraint propagation process, neither AC-3 nor DEEB will remember that y still is supported by x_{100}. In fact they may need to look at the other 98 values for X before "rediscovering" that y is supported by x_{100}. More generally, both AC-3 and DEEB are "arc revision" oriented, while AC-7 is "support maintenance" oriented, in the spirit of AC-6. As a result, AC-7 will have the same sort of constraint check advantage over AC-3 that has been demonstrated for AC-6, plus the additional advantage provided by inferences based on bidirectionality.

3 Inferring Support

3.1 Principle

We illustrate the principle of inferring support with a simple coloring example. The problem is to assign a color, a (aquamarine), b (blue) or c (coral), to each of two countries, X and Y, such that X and Y have different colors. (In general countries which share a border require different colors.) As a CSP, the countries are the variables, the colors the values. The constraint between the two variables

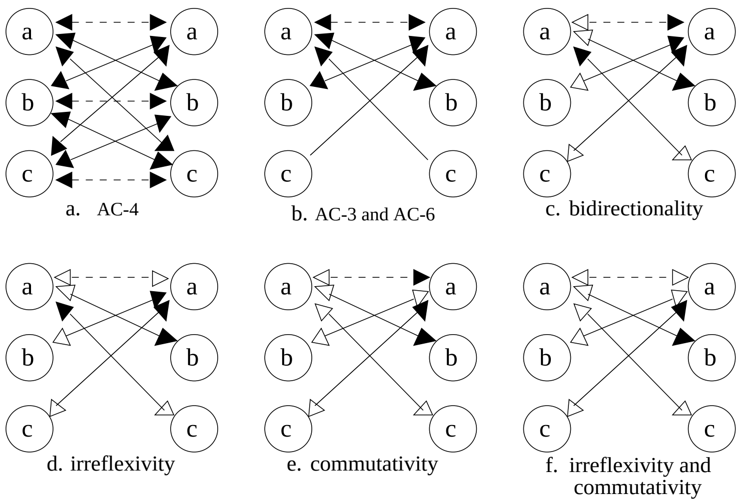

Figure 1: Making inferences.

specifies that the two countries cannot have the same color (i.e., it is a not-equal constraint).

Achieving arc consistency involves removing the arc inconsistent values—or in this simple coloring problem verifying that there are no arc inconsistent values. In general, removing one value may make another value arc inconsistent, so we say that achieving arc consistency can involve a *constraint propagation* process. However, in this simple example we do not have to worry about that.

The AC-4 arc consistency algorithm operates by first checking for all possible support. It stores summary information in support counters, and is later able to implement constraint propagation efficiently when propagation is required, by updating the counters rather than by performing further constraint checks. However, its brute force initial processing is costly; it performs all possible constraint checks twice. (This total can be reduced when values are deleted during initial processing.) *Eighteen* constraint checks are required for this little problem. These are shown in Figure 1a. The values for country X are on the left, for country Y on the right. An arrow from a value u for X to a value v for Y indicates that a check of the consistency of (X, u) with (Y, v), (denoted by $R_{XY}(u, v)$), has been computed while seeking support for (X, u). The double ended arrows indicate that inverse checks were made, while seeking support for both (X, u) and (Y, v). A solid arrow indicates that the check established consistency; a dashed arrow indicates that the check established inconsistency.

AC-3 seeks only to establish that each value has a supporting value (at every other variable). This only requires *eight* constraint checks here (Figure 1b), but some of them are inverse checks. In general, AC-3 propagation can involve additional redundant checking, but this does not enter in here. AC-6 can improve on AC-3 by avoiding some redundant checking during propagation.

As opposed to AC-3, it looks for another support if and only if the support it finds initially is deleted; and it proceeds on from that point to look for another support; it does not need to "start over" as AC-3 does. However, in this simple example, AC-6 has no opportunity to exhibit this additional intelligence. It, too, requires *eight* constraint checks.

AC-3 may check pairs once, twice, or up to $2d$ times (d times in each direction). AC-4 checks each pair at most twice. AC-6 was said to be optimal in the sense that it only made necessary checks. (The actual number of checks it makes is still subject to processing order effects [WF92].) However, support inference based on the bidirectionality of support can reduce constraint checks further. AC-6 (AC-3 and AC-4 as well) checks, for example, that (X, a) is supported by (Y, b), and then separately checks that (Y, b) is supported by (X, a). These algorithms cannot see that they already know (Y, b) is supported, whereas they just found that out while looking for support for (X, a).

The AC-Inference algorithm (AC-7 as well[1]) that we propose here *can*, in effect, see this. After it checks (X, a) against (Y, b) and finds that the latter supports the former, it *infers* that the former also supports the latter. This inference is based upon the simple metaknowledge that support is bidirectional. Utilizing such inferences, AC-Inference only requires *five* constraint checks to establish arc consistency. This is shown in Figure 1c. White arrowheads are used to indicate inferences. The double-ended arrows here, with a white arrowhead on one end, indicate a constraint check in one direction followed by a constraint inference in the other. Notice that negative support information can also be inferred: after performing a constraint check to determine that (Y, a) does not support (X, a), we can infer that (X, a) does not support (Y, a).

In this simple example, AC-Inference only saves inverse constraint checks; in fact, one might assume that at best it will require half the constraint checks of AC-6 on a given problem. However, in the next section (when analyzing the AC-7 savings), we will see that non-inverse constraint checks can be saved, and far fewer than half the constraint checks of AC-6 are required for appropriately structured problems.

Moreover, bidirectionality is but one instance of the general principle of using constraint metaknowledge to infer or avoid constraint checks. We will illustrate this point with a couple of further examples. Suppose we have further metaknowledge of this constraint. Suppose we know that it is also *irreflexive*: $R_{XY}(v, v)$ does not hold for any v. Then the number of checks can be reduced to *four*. Irreflexivity allows us to immediately infer that (X, a) is inconsistent with (Y, a), making it unnecessary to check this (Figure 1d). Suppose we know that the constraint is *commutative*: $R_{XY}(u, v) = R_{XY}(v, u)$. This permits us to reduce the number of constraint checks to *three*. For example, after checking $R_{XY}(a, b)$ we can infer $R_{XY}(b, a)$. (See Figure 1e.) If we know it is irreflexive and commutative, we are down to *two* checks, Figure 1f.

Finally, there is another type of constraint metaknowledge that is potentially

[1] In fact, we will see in the next section that AC-7, as opposed to AC-Inference, only infers support information when it needs to know the value of the corresponding constraint check.

of considerable significance. Suppose we have a problem in which some of the constraints appear *repeatedly* involving different variables. A consistency check involving one of these "repeated" constraints can permit us to infer support information for all these variables (to the extent to which their domains share common values).

Dealing with repeated constraints is particularly worthwhile in certain kinds of problems, such as configuration, resource allocation, etc. Imagine that you want to assign hostesses to flights, with some restrictions (*compatibility* constraints) on which hostesses can work together. (There can be any reason for that.) You can encode this problem as a CSP in which you have as many variables by flight as the number of hostesses needed on this flight. Each variable takes its values in the set of available hostesses. Any two variables corresponding to the same flight are linked by the compatibility constraint, specifying the allowed pairs of hostesses. We see that this compatibility constraint is repeated between many pairs of variables since there is a clique of such constraints for each set of variables associated with the same flight. (Obviously, the complete problem probably contains other constraints between the variables in different flights.)

In evaluating the significance of inferred support we should bear in mind several factors that could increase the importance of utilizing such inferences:

- computing constraint checks could be costly;
- inferring one support may permit us to avoid a number of additional constraint checks. We will discuss this further in connection with AC-7.

3.2 Schema

The key to the inference schema, as it is to AC-3, AC-4 and AC-6, is maintaining appropriate information. AC-3 determines which variables need to be rechecked to see if all their values are still supported. AC-4 computes a total support count initially and then updates it as values are deleted. If AC-6 needs to find support for value (I, a) at variable J, it looks through an ordered list of J values for a single supporting value, b. If b is deleted, AC-6 looks for another supporting value later in the list.

The schema data structures of AC-Inference are something of a hybrid of AC-4 and AC-6. For each variable, I, for each of its possible values, a, and each other variable, J, which shares a constraint with I, AC-Inference maintains:

- $CS[I, a, J]$, a *currently-supported set*, of values from D_J currently supported by (I, a). (As in AC-6[2], every value b of every variable J which shares a constraint with I must have a current support in D_I.)

[2]The $S[I, a]$ sets in AC-6 correspond in AC-Inference to the union of the $CS[I, a, J]$ sets for each variable J sharing a constraint with I. In fact, as in AC-6, only one currently-supported set per value would be sufficient in AC-Inference. We use this presentation to stay homogeneous with the support and unchecked sets, and with the currently-supported sets of AC-7, which will need this implementation. (See Section 4.)

- $S[I, a, J]$, a *support set*, of values from D_J that support (I, a)
- $U[I, a, J]$, an *unchecked set*, of values from D_J that have not yet been checked to see if they support (I, a)
- Values that have been checked and found not to support (I, a) appear in none of these sets.

Maintaining the S and U sets enables the schema to remember inferred constraint check information. Positive results are remembered by moving from the unchecked sets to the support sets. Negative results are remembered by deleting from the unchecked sets. Like AC-6, we only have to work through the unchecked sets once; like AC-6, we only have to look for a single support; unlike AC-6, we can avoid checking some of the values via inference. We remark that as soon as bidirectionality of support is inferred, the currently-supported set of each value-variable pair is included in its support set.

The algorithm schema can be expressed as Algorithm 1.

If no specific metaknowledge is known about the constraints, for each value (I, a), and each variable J sharing a constraint with I, the support set is initialized empty, and the unchecked set is initialized with the domain of the variable (i.e., $U[I, a, J] = D_J$). When some initial knowledge is available, some values of the variable J can directly be added to the support set or ignored, instead of being put in the unchecked set (e.g., if R_{IJ} is reflexive, value a for J will directly be put in $S[I, a, J]$ instead of $U[I, a, J]$; if R_{IJ} is irreflexive, a for J will appear neither in $S[I, a, J]$ nor $U[I, a, J]$).

The *seek-support set* contains value-variable pairs. If a is a value for a variable I and the variable J shares a constraint with I, $[(I, a), J]$ will appear on the initial seek-support set. Processing an element $[(I, a), J]$ of this set (see subprocedure `SeekCurrentSupport`) involves looking for a current support for the value in the support set of the variable (i.e., $S[I, a, J]$). If none of the values of this set are still in the domain D_J, the values of the unchecked set still in D_J are checked for consistency with (I, a). After each constraint check, a set of inferences is made, based on the outcome. The specific set of inferences chosen instantiates the schema. If, during the processing of an element, the unchecked set becomes empty without finding a support, then the deletion of the value is processed. (See subprocedure `ProcessDeletion`.) The value is removed from the domain, and for all the values in its currently-supported sets, the appropriate value-variable pair enters the seek-support set.

We recommend implementing the seek-support set as a stack, so that the effects of deletions will propagate immediately. We believe this to be an improvement on the "two-pass" organization of AC-4 and AC-6.

At this point, we should note that the schema given in [BFR95] is not the same as the one presented here. The schema given in [BFR95] was written in an AC-4 like spirit, while the one given here is closer to an AC-6 spirit. Indeed, the schema in [BFR95] does not use the currently-supported sets, and so, needs to update the support and unchecked sets after each deletion (by deleting the removed value from all the sets it belonged to) as AC-4 needs to update its

Algorithm 1: `AC-Inference` algorithm schema

`AC-Inference`;

begin

Initialize the SeekSupportSet with all the $[(I, a), J]$;
1 Establish the initial S sets and U sets using initial inferences;
2 Initialize the CS sets with the empty set;
while *SeekSupportSet* $\neq \emptyset$ **do**
  pick $[(I, a), J]$ from SeekSupportSet ;
  if $a \in D_I$ **then**
    $c \leftarrow$ `SeekCurrentSupport`(I, a, J);
    if $c \neq$ ***nil*** **then**
      put a in $CS[J, c, I]$;
    else `ProcessDeletion`$(I, a,$ SeekSupportSet$)$;

end

function `SeekCurrentSupport` **(in** I: variable; **in** a: value; **in** J: variable **):** value

/* returns a value b supporting (I, a), or **nil** if not found */ ;
/* updates S and U if necessary */ ;

begin

found $\leftarrow$ **false**;
while $(S[I, a, J] \neq \emptyset)$ ***and*** $(\neg$ *found*$)$ **do**
  $b \leftarrow$ an element of $S[I, a, J]$;
  if $b \in D_J$ **then** found $\leftarrow$ **true**;
  else delete b from $S[I, a, J]$;
while $(U[I, a, J] \neq \emptyset)$ ***and*** $(\neg$ *found*$)$ **do**
  pick b from $U[I, a, J]$;
  if $b \in D_J$ **then**
    found $\leftarrow R_{IJ}(a, b)$;
3   make inferences from $((I, a)(J, b))^3$;
if *found* **then return** b;
else return nil ;

end

procedure `ProcessDeletion` **(in** I: variable; **in** a: value;
in out SeekSupportSet: set**)**

begin

remove a from D_I;
if $D_I = \emptyset$ **then exit** ("wipe out");
for each J/R_{IJ} *exists* **do**
  for each $b \in CS[I, a, J]$ **do**
    delete b from $CS[I, a, J]$;
4   put $[(J, b), I]$ in SeekSupportSet ;

end

counters. A counter equal to zero was the signal to remove a value in AC-4. Support and unchecked sets both empty on the same constraint is the signal to remove a value in the AC-Inference of [BFR95]. On the other hand, the AC-Inference algorithm schema presented here uses the currently-supported sets. Therefore, it does not need updated support and unchecked sets. As in AC-6, a value is viable as long as it has a current support on each constraint. Thus, support and unchecked sets can contain values no longer in the domain. This latter version of the AC-Inference algorithm schema, avoiding many updates, has shown experimental superiority. The experimental results presented in [BFR95] were already based on this latter version, although it was not pointed out in the paper.

3.3 Analysis

A sketch of proof of correctness.

We do not give an exhaustive proof here. Indeed, AC-Inference has the same structure as AC-7, which is a bit more technical, and for which a complete proof is given in Section 4.3.

An invariant property of AC-Inference is that, for any value (I, a) in $\mathcal{D}$, any constraint R_{IJ}, and any value b in D_J, $R_{IJ}(a, b)$ is true only if b belongs to $S[I, a, J] \cup U[I, a, J]$. A value (I, a) is removed from $\mathcal{D}$ only if these two sets are empty. Thus, if $\mathcal{D}^0$ denotes the set of initial domains (i.e., the value of $\mathcal{D}$ at the beginning of AC-Inference), a value removed from $\mathcal{D}$ by AC-Inference cannot belong to $AC(\mathcal{D}^0)$. Furthermore, when AC-Inference finishes, the SeekSupportSet is empty, and so, all the values still in $\mathcal{D}$ have a current support on each constraint. $\mathcal{D}$ is then arc consistent. If $\mathcal{D}$ is arc consistent and the values of $\mathcal{D}^0 \setminus \mathcal{D}$ were out of $AC(\mathcal{D}^0)$, we can conclude that $\mathcal{D} = AC(\mathcal{D}^0)$ at the end of AC-Inference.

Space and time complexities.

AC-Inference needs to represent explicitly the currently-supported sets CS, the support sets S, and the unchecked sets U. Each value belongs to at most one CS set on each constraint since a value has only one current support per constraint. Thus, the total size of the CS sets is in $O(ed)$. For each value (I, a) and each constraint R_{IJ}, the size of $S[I, a, J]$ plus the size of $U[I, a, J]$ is bounded above by the size of D_J. Then, the total size of S and U sets is bounded above by $2ed^2$. The SeekSupportSet has a maximal size in $O(ed)$. Thus AC-Inference has an $O(ed^2)$ space complexity.

The inner loops of AC-Inference are in `SeekCurrentSupport`. Each loop is executed at most d times for each $S[I, a, J]$ or $U[I, a, J]$ set, for total work of at most d deletions in each $S[I, a, J]$ and $U[I, a, J]$ set. Furthermore, the total work performed by line 3 cannot exceed $2ed^2$ inferences since once a support or

[3] Making inferences involves updating the suitable S and U sets, depending on the kind of inferences the constraint concerned permits.

a non-support is inferred, it cannot be inferred a second time. (Lists S and U are updated.) Thus, the time complexity of AC-Inference is in $O(ed^2)$.

Implementation details.

In this subsection we present a few data structures in order to show how certain implementation problems can be solved.

Most of the time, the algorithms we present work with the indices of the values in a domain. E.g., a domain D_I initially containing the values *aquamarine*, *blue*, and *coral* will probably be represented by an array containing the values 0, 1, and 2. But the constraint "$\neq$" holds between colors, not between indices. (See the example of Section 3.1.) This means that we need to reach an original value from its index in $O(1)$. This can easily be done by a table of correspondence, which for each index of a domain gives the original value it represents. (E.g., for the index 1 of D_I it gives *blue*.)

However, it can arise that in some domains we need to access the index from the original value. Suppose R_{IJ} is commutative[4]. Then, after checking $R_{IJ}(aquamarine, blue)$ we must infer the value of $R_{IJ}(blue, aquamarine)$. Thus we need to reach the index of *blue* in D_I and the index of *aquamarine* in D_J. Note that *blue* in D_I (resp. *aquamarine* in D_J) has not necessarily the same index as *blue* in D_J (resp. as *aquamarine* in D_I) since the initial domain of variable J (denoted by D^0_J) is not necessarily equal to the initial domain of variable I (denoted by D^0_I). (E.g., if $D^0_J = \{light\text{-}aquamarine, aquamarine, light\text{-}blue, blue\}$, *blue* will have the index 3 in D_J while its index is 1 in D_I.) The problem that arises when the algorithm is implemented is: how can the value *aquamarine* of variable J and the value *blue* of variable I be accessed?

The first response is to use a hash table to do so. In this case, and under reasonable assumptions, the expected time to search for an element is close to a small constant [VDT92]. But it is impossible to ensure an $O(1)$ complexity.

Another approach is to define a data structure in which the same value in two different domains has the same index. A value in D_I (or D_J) is associated with the same index as the one it would have in the virtual domain D^0_{IJ} that is the union of D^0_I and D^0_J. (Indices corresponding to values in $D^0_{IJ} \setminus D^0_I$, resp. in $D^0_{IJ} \setminus D^0_J$, are set to be out of the domain D^0_I, resp. D^0_J.) In our example, D^0_{IJ} being the set $\{light\text{-}aquamarine, aquamarine, light\text{-}blue, blue, coral\}$, the indices of the values *aquamarine*, *blue*, and *coral* of D_I would respectively be 1, 3, and 4, while those of the values *light-aquamarine*, *aquamarine*, *light-blue*, and *blue* of D_J would be 0, 1, 2, and 3. This method has a drawback: to implement a domain of a variable I, it is necessary to take the union of the domains of all the variables that are involved in a commutative (or repeated) constraint on I. For certain networks, this leads to the union of all the domains of the network. Nevertheless, it is important to note that dealing with commutative (or repeated) constraints is useful only if the initial domains are almost equal. This method ensures a complexity in $O(1)$.

[4]The same problem arises with repeated constraints.

Thirdly, another simple method exists. For each constraint R_{IJ}, it consists of storing with each value of D_I^0 its index in the initial domain of J, and conversely. (A dummy can be used if a value is not present in a domain.) This method also ensures a complexity in $O(1)$.

From a general point of view, the first method is the best because no assumptions are made about the network, and its practical performance is good. In the case where the warranty of a complexity in $O(1)$ is required, the second method should be prefered to the third one.

Another problem of implementation in AC-Inference is the direct access needed to remove the value a from the list $U[J, b, I]$ after the deletion of a value b from the list $U[I, a, J]$. It is clear that a double link between the value a of $U[J, b, I]$ and the value b of $U[I, a, J]$ is necessary. Then, two approaches are possible when, for instance, the value b is removed from $U[I, a, J]$. With the first approach, we access the value a of $U[J, b, I]$ and we immediately remove it from this list. In this case, the list $U[J, b, I]$ must be doubly linked in order to remove any element in $O(1)$. With the second approach, we access the value a of $U[J, b, I]$, and we simply mark it as "removed." An element marked as "removed" will be deleted from the list when it is reached by a traversal of this list. In this case, the list $U[J, b, I]$ does not need to be doubly linked, and complexity does not change: it remains in $O(1)$.

4 AC-7

AC-7 refines the AC-Inference algorithm schema, while restricting inferences to those based on bidirectionality. Since bidirectionality is a general property of constraints, AC-7 is a general purpose arc consistency algorithm. Also by restricting our inferences to bidirectionality, we are able to maintain a space complexity of the same order as AC-6, namely $O(ed)$, while AC-Inference (and AC-4) space complexity is quadratic in d. In some applications, the size of the problem (number of variables, size of the domains, number of constraints) is so high that it is impossible to store in memory all the support sets and unchecked sets needed by AC-Inference. In order to avoid the space requirements of AC-Inference, AC-7 does not use the unchecked and support sets of AC-Inference, but stays closer to an AC-6-like data structure. In fact, AC-7 does not infer all the supports and non-supports AC-Inference does, but *deduces* them only when it needs to know their value, thus performing a kind of "lazy inference." But, even without the support and unchecked sets, AC-7 guarantees performing as few constraint checks as AC-Inference, having the following desirable properties [BR95]. AC-7:

1. never checks $R_{IJ}(a, b)$ if there exists b' still in D_J such that $R_{IJ}(a, b')$ has already been successfully checked.

2. never checks $R_{IJ}(a, b)$ if there exists b' still in D_J such that $R_{JI}(b', a)$ has already been successfully checked.

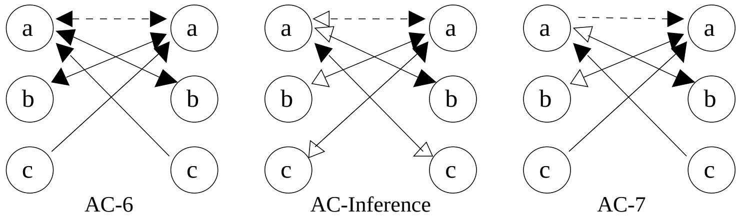

Figure 2: Bidirectionality

3. never checks $R_{IJ}(a, b)$ if:
   a) it has already been checked
   or b) $R_{JI}(b, a)$ has been checked.

4. has $O(ed)$ space complexity.

AC-3 lacks properties (1), (2), (3a), and (3b). AC-4 lacks (1), (2), (3b), and (4). AC-6 lacks (2), and (3b), the properties resulting from bidirectionality. AC-Inference lacks property (4).

Look again at the problem of Fig. 1 to illustrate the differences between AC-6, AC-Inference, and AC-7. Among the constraint checks performed by AC-6 and not by AC-7 (or AC-Inference) there are $R_{YX}(a, b)$ and $R_{YX}(b, a)$, which relate to property (3b), and $R_{YX}(a, a)$, which relates to both properties (2) and (3b).

AC-Inference needed *five* constraint checks and *five* inferences to achieve arc consistency. AC-7 needs *five* constraint checks also, but only deduces *two* of the inferences AC-Inference performed. It deduces only those it needs to know for the arc consistency computation. (With the same variable, value, and arc orderings, we could say that AC-7 never deduces a support or a non-support if AC-6 did not perform the corresponding constraint check.) In Fig. 2, we see that $R_{XY}(a, c)$, $R_{YX}(a, a)$, and $R_{YX}(a, c)$ are not deduced by AC-7 because it does not need them to achieve arc consistency.

4.1 Data structure

In AC-7, the support sets $S[I, a, J]$, and the unchecked sets $U[I, a, J]$ of AC-Inference cannot be represented explicitly because of the space complexity limitation of property (4). But, every time a constraint check $R_{JI}(b, a)$ is performed, AC-7 needs to store this information to avoid future $R_{IJ}(a, b)$ or $R_{IJ}(a, b')$ checks forbidden by properties (3b) and (2). AC-7 does this by adding two refinements to the data structures of AC-6.

AC-6 assigns an ordering of the values in every D_I, checks one support (the *first* one or *smallest* one with respect to the ordering) for each value (I, a) on each constraint R_{IJ} to prove that (I, a) is currently viable. When (J, b) is found as the smallest support of (I, a) on R_{IJ}, (I, a) is added to $S[J, b]$, the list of values currently having (J, b) as smallest support. If (J, b) is removed from D_J then AC-6 looks for the next support in D_J for each value (I, a) in $S[J, b]$. In AC-7, $S[I, a]$ sets of AC-6 are split into $CS[I, a, J]$ sets (as in AC-Inference). Thus, it is easy to ensure property (2): $b \in CS[I, a, J]$ implies (I, a) is known as supporting (J, b) on R_{JI}, then (I, a) is compatible with (J, b). So, an $R_{IJ}(a, b)$ is checked if and only if $CS[I, a, J] = \emptyset$; otherwise, we know (I, a) has a support on R_{IJ} without any new constraint check.

Arrays last are added to ensure property (3a)[5], and property (3b) for negative constraint checks: $\mathsf{last}[I, a, J]$ represents the last value of D_J which has been checked when looking for support for (I, a). In other words, it is the smallest value that may be compatible with (I, a) on R_{IJ}. In fact, for any b in D_J, we can say that $b < \mathsf{last}[I, a, J]$ means that b would not belong to $S[I, a, J] \cup U[I, a, J]$ in AC-Inference, while $b \geq \mathsf{last}[I, a, J]$ means that b would belong to $S[I, a, J] \cup U[I, a, J]$ in AC-Inference.

In more detail, this is the data structure of AC-7:

- Like AC-6, AC-7 needs to represent the domain D_I of a variable I as an ordered table of Booleans to answer in constant time whether a value of the initial domain is in the current domain or not. In this table, the initial D_I is considered as the integer range $1..|D_I|$. Furthermore, in this table we need to store the highest value in D_I, and for each value in D_I, the indexes of the previous and next values in D_I, in order to use the following constant time functions and procedure:
  - $highest(D_I)$ returns the greatest value in D_I if $D_I \neq \emptyset$.
  - $next(a, D_I)$ returns the smallest value in D_I greater than a if $a \in D_I \backslash highest(D_I)$; if $a = highest(D_I)$, **nil** is returned.
  - $remove(a, D_I)$ removes the value a from D_I.
- For all a in D_I, $CS[I, a, J]$ contains all values b in D_J for which (I, a) is assigned as its current support. The current support is not necessarily the smallest one, as opposed to AC-6.
- For all a in D_I, $\mathsf{last}[I, a, J]$ is updated by AC-7 to be the last value of D_J already checked when looking for support for (I, a). So, the last arrays ensure that every b in D_J such that $R_{IJ}(a, b)$ has already been checked is less than or equal to $\mathsf{last}[I, a, J]$, and that every b in D_J compatible with (I, a) is greater than or equal to $\mathsf{last}[I, a, J]$.
- The SeekSupportSet has the same behavior as in AC-Inference. Handling the SeekSupportSet as a stack seems to be an efficient heuristic because it propagates the consequences of deletions as soon as they appear, and thus discovers empty domains earlier.

[5] Property (3a) held in AC-6 without last arrays because we knew that (I, a) was always the smallest support for values in $S[I, a]$. In AC-7 we lose this strong property.

4.2 Algorithm

The algorithm AC-7 (see Algorithm 2), has the same framework as AC-Inference. There are two main operations: seeking a current support for a value, and processing the deletion of a value. Initially, for each value (I, a), for each constraint R_{IJ}, the value-variable pair $[(I, a), J]$ is put in the SeekSupportSet. For each value-variable pair $[(I, a), J]$ in the SeekSupportSet, AC-7 seeks a support for (I, a) in D_J by means of function `SeekCurrentSupport`. If a value c is found by `SeekCurrentSupport`, a is added to $CS[J, c, I]$ since c is now the current support of (I, a) on R_{IJ}. During the whole algorithm, every time a value (I, a) is found without support on a constraint, it is removed from D_I, and this deletion is processed by adding in the SeekSupportSet all the value-variable pairs $[(J, b), I]$ such that (J, b) was supported by (I, a) (i.e., values (J, b) that were in the $CS[I, a, J]$ sets).

When `SeekCurrentSupport` is called to find a support for (I, a) in D_J, it first checks (lines 1 to 2) whether $CS[I, a, J]$ contains values still in D_J or not. Indeed, if $CS[I, a, J]$ contains values in D_J, the search on R_{IJ} for a support for the value (I, a) is useless: (I, a) has inferable supports on R_{IJ} (values supported by (I, a) are supports for (I, a)). Every time `SeekCurrentSupport` unsuccessfully checks if a value b in $CS[I, a, J]$ is in D_J, b is deleted from $CS[I, a, J]$ to avoid checking it again in the following.

If no inferable support is found, the search for support for (I, a) on R_{IJ} is performed (lines 3 to 5). This second part of `SeekCurrentSupport`, which looks for the smallest value in D_J not smaller than $\mathsf{last}[I, a, J]$ and supporting (I, a) on R_{IJ}, looks like the function `nextsupport` of AC-6. The first difference is the addition of the test of line 4 to avoid checking $R_{IJ}(a, b)$ when $R_{JI}(b, a)$ has already been checked as false (i.e., when we know that (J, b) has no support in D_I until a value greater than a). The second difference is the need to record in $\mathsf{last}[I, a, J]$ the (smallest) value found in D_J as supporting (I, a) on R_{IJ}, if any (line 5).

4.3 Analysis

Proof of correctness.

AC-7 is similar to AC-6 except that it does not always know the smallest support for a value on a constraint, but it knows only the existence of a support and an inferior bound below which no support exists. After the initialization step (lines 1 to 4 in AC-7), the following property holds until the end of the algorithm:

$$(\alpha)\ \forall (I, a) \in \mathcal{D}, \forall R_{IJ}, \forall (J, b) \in \mathcal{D} : R_{IJ}(a, b) \Rightarrow b \geq \mathsf{last}[I, a, J]$$

Thus, when (I, a) has no current support and no inferable support in D_J ($CS[I, a, J] \cap D_J = \emptyset$), we can start looking for another one at $\mathsf{last}[I, a, J]$

[6]To improve the algorithm slightly, we can replace "$b \leftarrow b + 1$" by "$b \leftarrow$ *latest-next*(b, D_J)" where *latest-next*(b, D_J) is the value b' that was equal to $next(b, D_J)$ when b was removed from D_J (*latest-next*$(0, D_J) = 1$). This improvement can be significant on large domains.

Algorithm 2: AC-7

```
AC-7;
  begin
1   SeekSupportSet ← ∅;
2   for each (I, J)/R_IJ exists do
      for each a ∈ D_I do
        CS[I, a, J] ← ∅;
3       last[I, a, J] ← 0 /* first value of D_J minus 1*/ ;
4       put [(I, a), J] in SeekSupportSet ;

    while SeekSupportSet ≠ ∅ do
      pick [(I, a), J] from SeekSupportSet ;
      if a ∈ D_I then
        c ← SeekCurrentSupport(I, a, J);
        if c ≠ nil then
          put a in CS[J, c, I];
5       else ProcessDeletion(I, a, SeekSupportSet);

  end

function SeekCurrentSupport (in I: variable; in a: integer; in J: variable): integer
  /* returns a value supporting (I, a), or nil if not found */ ;
  /* updates last[I, a, J] if necessary */ ;
  begin
    found ← false;
1   while (CS[I, a, J] ≠ ∅) and (¬ found) do
      b ← an element of CS[I, a, J];
      if b ∈ D_J then found ← true;
2     else delete b from CS[I, a, J];
    if found then return b;

3   b ← last[I, a, J];
    if b > highest(D_J) then return nil ;
    while b ∉ D_J do b ← b + 1[6];
    while (b ≠ nil) and (¬ found) do
4     if (last[J, b, I] ≤ a) and then (R_IJ(a, b)) then
        found ← true;
      else b ← next(b, D_J);
5   last[I, a, J] ← b;
    return b;
  end
```

without forgetting any support of (I, a). Furthermore, we do not need to check $R_{IJ}(a, b)$ for values b in D_J such that $\mathsf{last}[J, b, I] > a$ because of property (α). So, despite the first test of line 4 of the function `SeekCurrentSupport`, a support of (I, a) cannot be forgotten. Thus, value (I, a) is only removed from D_I (line 5 of AC-7) when it has no support in D_J on a constraint R_{IJ}. If all previously removed values are out of $AC(\mathcal{D}^0)$ ($\mathcal{D}^0$ being the initial set of domains), then (I, a) is out of $AC(\mathcal{D}^0)$. $AC(\mathcal{D}^0)$ is trivially included[7] in $\mathcal{D}$ when AC-7 started (since $\mathcal{D}$ was equal to $\mathcal{D}^0$). So, by induction, (I, a) is out of $AC(\mathcal{D}^0)$. Thus, $AC(\mathcal{D}^0) \subseteq \mathcal{D}$ is an invariable property of AC-7.

Every time a value (I, a) is found without support on a constraint, it is removed from $\mathcal{D}$. Every time a value (I, a) is removed, all the values (J, b) it was supporting are added to the SeekSupportSet in order to be checked for other support (line 4 of `ProcessDeletion`—see Algorithm 1). Thus, every value (J, b) that belongs to $\mathcal{D}$ and that does not appear in the SeekSupportSet has at least one support in $\mathcal{D}$ on each constraint R_{IJ}. AC-7 terminates with an empty SeekSupportSet. As a result, after AC-7, every value in $\mathcal{D}$ has a support in $\mathcal{D}$ on each constraint. Thus, $\mathcal{D}$ is arc-consistent.

$AC(\mathcal{D}^0) \subseteq \mathcal{D}$ and $\mathcal{D}$ arc-consistent at the end of AC-7 imply that $\mathcal{D}$ is the maximal arc-consistent domain $AC(\mathcal{D}^0)$ at the end of AC-7.

Desirable properties.

We first show that AC-7 has all of the desirable properties enumerated earlier.

By the same principle as with AC-6, properties (1) and (3a) hold. Indeed, we stop looking for supports for a value (I, a) on R_{IJ} as soon as we find the first support, and an $R_{IJ}(a, b)$ cannot be checked twice since we start looking for a new support for (I, a) on R_{IJ} where we stopped the previous time.

Every time a constraint check $R_{JI}(b, a)$ is successfully performed (when looking for a support for (J, b)) AC-7 records b in $CS[I, a, J]$. Since AC-7 only looks for a support for (I, a) on R_{IJ} if $CS[I, a, J] \cap D_J = \emptyset$, we are sure that $R_{IJ}(a, b)$ is checked only when none of the previously done constraint checks on R_{JI} can prove that (I, a) has a support in D_J. Property (2) holds.

Next, we have to prove that when we check $R_{IJ}(a, b)$, $R_{JI}(b, a)$ has never been checked before. If $R_{JI}(b, a)$ has already been successfully checked, b must be in $CS[I, a, J]$ or out of D_J, and $R_{IJ}(a, b)$ cannot be checked (see above). If $R_{JI}(b, a)$ has been checked as false, that means that we have looked for a support for (J, b) in D_I and that (I, a) was not a support. So, if $b \in D_J$, $\mathsf{last}[(J, I), b]$ is greater than a, and line 4 of function `SeekCurrentSupport` avoids checking $R_{IJ}(a, b)$. Property (3b) holds.

Space and time complexities.

The worst-case space complexity of AC-6 was $O(ed)$ because of the size of $S[J, b]$ sets. In AC-7, $S[J, b]$ sets of AC-6 are split into $CS[J, b, I]$ sets. This does not

[7]"included" is loosely used here since the inclusion is on the respective elements of the two sets.

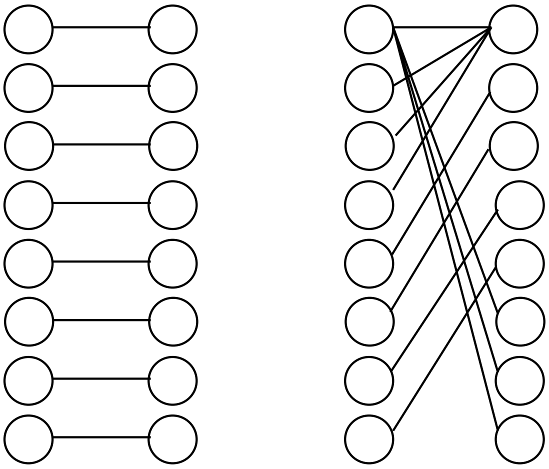

Figure 3: Analysis

increase their total size: each arc-value pair $[(I, J), a]$ has at most one current support (J, b) (i.e., a belonging to $CS[J, b, I]$). Hence, the total size of the $CS[J, b, I]$ sets is in $O(ed)$. We have to add arrays last, which take a $2ed$ space, and the size of the SeekSupportSet, which is in $O(ed)$. Thus, the total space complexity of AC-7 remains in $O(ed)$.

The two inner loops of AC-7 are in `SeekCurrentSupport` (lines 1 to 2, and lines 3 to 5). Since each $R_{IJ}(a, b)$ is checked at most once, in the second loop of `SeekCurrentSupport`, complexity due to the calls to this loop is in $O(ed^2)$, as in AC-6. During the whole algorithm, a value $a \in D_I$ is put at most once in $CS[J, b, I]$: when (J, b) is assigned as the current support of (I, a). `SeekCurrentSupport`, and its first inner loop, is called at most d times for each $CS[I, a, J]$ (when a value b current support of (I, a) is removed from D_J), for a total work of at most d deletions in each $CS[I, a, J]$. That loop is in $O(d)$ on each $CS[I, a, J]$, and so, complexity due to the calls to this loop is bounded above by $2ed \times O(d)$ i.e., $O(ed^2)$. Thus, $O(ed^2)$ is the worst-case time complexity for AC-7, as for AC-6.

Savings.

We will now look at two sample problems, problem structures really, to demonstrate the different levels of savings that are possible for AC-7 as compared to the other algorithms. Figure 3 shows the microstructure of these problems in the form of a *consistency graph*, where each consistent pair of values is joined by an edge. Each of these problems will involve only two variables, and we will put the values for each variable in a separate column, assuming the unlabeled values are ordered lexicographically from top to bottom, and the unlabeled variables are ordered lexicographically from left to right.

Note first that AC-7 is unlikely to make matters worse. AC-4 always computes all checks twice, AC-7 at most twice. (Both can avoid some checks by deleting unsupported values.) AC-3 and AC-6 look for a support and then look for additional support only as needed when current support is deleted, AC-6 avoiding some redundant checks made by AC-3. The same is true of AC-7, including the avoidance of redundant checks, but inferences further reduce checking and may reduce the need to look for support.

AC-4 and AC-6 complete an initialization phase before turning their attention to deletions ("two-pass" organization). A queue-based implementation of AC-3 will as well. The stack-oriented implementation of the inference schema is potentially profitable in its priority handling of deletions. (As well as more elegant in expression.)

Observation 1 *AC-7 can cut in half the number of constraint checks.*

Consider the problem in Figure 3a. (This could be regarded as representing an equality constraint, as opposed to the inequality constraints we focused on earlier in the context of coloring problems.) The support for the ith value of the first variable requires i constraint checks to find. The total number of constraint checks for the first variable is then $d(d+1)/2 = 36$ for AC-3, AC-6, and AC-7. The support for the ith value of the second variable, that is deduced by AC-7, requires also i constraint checks to find for the other algorithms. The total number of constraint checks performed by AC-7 on this constraint is thus 36, half the number needed by AC-3 or AC-6.

It might seem at first glance that AC-7 can only avoid computing inverse checks that establish whether (J, b) supports (I, a) after establishing whether (I, a) supports (J, b). (See property (3b) of Section 4.)

Observation 2 *AC-7 can avoid non-inverse constraint checks.*

In fact, the problem in Figure 3a also demonstrates this. The savings noted earlier included avoiding checks that were not computed earlier—see property (2) of Section 4. (Checks between the i^{th} value of the first variable and the j^{th} value of the second variable are never performed by AC-7 when $i < j$, while they are performed by AC-3 or AC-6 when looking for support for the j^{th} value of the second variable.)

Even in this example, however, the number of checks for AC-7 is only half the number of checks for AC-3 and AC-6.

We can go further, moreover, and show that AC-7 can, in fact, more than cut in half the number of constraint checks.

Observation 3 *AC-7 can more than cut in half the number of constraint checks.*

Figure 3b presents an example in which the number of checks for AC-7 is 21; for AC-3 or AC-6, 48.

Observe that this example generalizes to show that AC-7 can save a quadratic number of non-inverse checks. (The total number of different possible checks is only quadratic itself.)

Observation 4 *AC-7 can save $O(d^2)$ non-inverse constraint checks.*

The previous example can be generalized from domain sizes of 8 to domain sizes of d, where $O(d^2)$ checks are saved. Observe that it is constructed so that $d/2$ of the values in the second domain must be checked (and fail) against from $d/2$ to $d-1$ of the values in the first domain with AC-3 or AC-6, while AC-7 avoids these checks entirely. In fact, on this example, AC-7 asymptotically requires four times fewer constraint checks than AC-3 or AC-6 (when d grows).

Notice that we have established these various observations without even considering constraint propagation, where further savings may be realized. We would expect AC-7 to show to better advantage on more tightly constrained problems, where one has to look harder to find support, and thus can benefit more when inferences preclude the need to look for support.

5 Experimental results

We tested AC-Inference and AC-7 on several types of problems. In this section we first show a few results on randomly generated problems using the generator available in [FBDR96]). In that case, AC-7 (and not AC-Inference) is compared to the other generic algorithms achieving arc consistency. (Since AC-Inference is an algorithm that takes advantage of constraint semantics, we would not expect it to do best on random problems, for which no metaknowledge can be extracted from the constraints.) Second, we compare both AC-7 and AC-Inference to the other algorithms on the Radio Link Frequency Assignment problem (RLFAP). We took problems from a set that stem from real instances, which is available in a CSP benchmarks archive (ftp://ftp.cs.unh.edu/pub/csp/archive/code/benchmarks/). In this problem, all the constraints are commutative and irreflexive since for every pair of variables X_I and X_J, only two types of constraints are possible: $|X_I - X_J| > k$, or $|X_I - X_J| = k$, k being a positive number. The number of values for k are limited, so that many constraints in the network are repeated. Finally, we report the results obtained on the Zebra problem, a common CSP benchmark (also available in the benchmarks archive).

Though the general task in these problems is not only to achieve arc consistency, we will limit our comparisons to it, since it is the central aim of our paper. (We will see the limitations this involves.) See [BFR95] and [Rég95] for a comparison of the behavior of search algorithms using different arc consistency algorithms to maintain arc consistency during search. (This search technique was proposed in [SF94], and called *MAC*.)

We compared several versions of arc consistency algorithms on the problems presented above. Each table corresponds to one of the classes of problems, and the different versions of arc consistency algorithms applied to them are the following:

- AC-3: the classical version [Mac77]
- AC-4: the classical version [MH86]

- AC-6: the classical version [Bes94] (two-pass organization)
- AC-6p: a version of AC-6 organized as AC-7 and AC-Inference, propagating deletions as soon as they occur
- AC-7: the one presented in this paper
- AC-I: the AC-Inference presented in this paper, provided with the knowledge of reflexive, commutative, irreflexive, etc., constraints, but not with repeated constraints
- AC-Ir: AC-Inference provided with all the possible knowledge (Reflexivity, commutativity, repeated constraints, etc.)

		$<150, 50, 500/0.045, 1250/0.5>$ (under-constrained)		$<150, 50, 500/0.045, 2350/0.94>$ (over-constrained)	
		#ccks	time	#ccks	time
AC-3		100,010	0.08	514,973	0.37
AC-4		2,500,000	2.39	1,384,874	1.17
AC-6		100,010	0.15	492,704	0.40
AC-6p		100,010	0.15	223,077	0.24
AC-7		94,030	0.19	205,070	0.29
		$<150, 50, 500/0.045, 2296/0.918>$ (phase transition)		$<50, 50, 1225/1.0, 2188/0.875>$ (phase transition)	
		#ccks	time	#ccks	time
AC-3	ac	2,353,669	1.69	2,932,326	1.91
	inc	4,865,777	3.67	8,574,903	5.46
AC-4	ac	2,238,076	1.89	5,748,678	4.56
	inc	2,224,428	1.92	5,700,125	4.66
AC-6	ac	639,550	0.67	1,022,691	1.10
	inc	747,045	0.82	1,231,452	1.38
AC-6p	ac	625,845	0.65	1,010,731	1.10
	inc	672,406	0.75	1,138,472	1.30
AC-7	ac	481,878	0.77	820,814	1.37
	inc	535,095	0.92	912,795	1.60

Table 1: Random problems: Arc consistency results in mean number of constraint checks (#ccks) and mean cpu time in seconds (time) on a PC Pentium 200MHz (50 instances generated for each set of parameters).

In Table 1, we present the results of several types of randomly generated problems with different sizes, densities, and tightness. $<N, K, C/p1, T/p2>$ represents the class of problems generated with N variables having K possible values in their domain, and $C = p1 \cdot N(N-1)/2$ constraints with $T = p2 \cdot K^2$ forbidden pairs of values in each. The different values of the parameters correspond to under-constrained problems (easily checked as being arc consistent), over-constrained problems (easily found arc inconsistent —a domain wipe out is observed), and problems at the phase transition of arc consistency [GS96a, GMP+97] (i.e., problems for which establishing arc consistency sometimes succeeds and sometimes not).

On under-constrained problems, a single pass is sufficient to find that all the values are arc consistent. No propagation is needed. AC-3, which has the simplest data structure, is obviously the best one in cpu time, performing the same number of constraint checks as AC-6 or AC-6p. AC-4 is the worst one because it builds its very expensive lists of supported values, which will be completely useless since no propagation occurs.

On over-constrained problems, the arc consistency process also terminates very quickly because a domain wipe out quickly occurs. The data structures of AC-6, AC-6p, and AC-7 do not significantly pay off. AC-3 is indeed comparable to them in cpu time. The differences in performance between AC-6 and AC-6p show the effectiveness of our implementation (processing deletions first) to detect a domain wipe out earlier. AC-4 is less bad here than on the under-constrained problems because it finds a domain wipe out before its expensive initialization phase is completed.

On problems at the phase transition of arc consistency, where some propagation is necessary to find the arc consistent closure or to prove arc inconsistency, the results are different. Even if the propagation phase remains very short compared to what happens during a search process maintaining arc consistency, we can see that AC-6, AC-6p, and AC-7 are faster than AC-3, which begins to be lost in its great amount of propagation. On these problems, we separate the performance on instances where an arc consistent closure is found (denoted by *ac*) from those where arc inconsistency is proved (denoted by *inc*). Thus, we can see that AC-3 is worse on arc inconsistent problems (where much propagation is necessary to prove inconsistency) than on those where an arc consistent closure is found. We suspect that these not trivially arc inconsistent problems are typically the kind of situations that are generated many, many times during a search process maintaining arc consistency. Finally, AC-7 does not save enough constraint checks to have an advantage over AC-6p in cpu time. This is not surprising since the kind of constraints favoring AC-7, such as those described in Section 4.3, are rarely generated randomly.

Experimental results on RLFAPs (see Table 2) give us some interesting information on the significance of inferring support. First, if we except AC-3, the number of constraint checks is correlated with cpu time, even on this kind of problem where constraint checks are cheap to compute. This means that, on these problems, the benefit of inferences not only offsets, but also overcomes their cost. (Otherwise, "#ccks" would have decreased from AC-4 to AC-Ir while "time" would not.) The very good running times of AC-Ir confirm the advantage of using the most metaknowledge we can. Second, as on random problems, the results of AC-6 and AC-6p show the effectiveness of processing deletions first on problems with many arc inconsistent values (e.g., RLFAP #5—12046 values deleted, RLFAP #8—arc inconsistent), as opposed to problems with limited or no propagation (RLFAP #3—0 values deleted, RLFAP #11—0 values deleted). Third, AC-3 shows surprisingly good cpu times. This means that when constraint checks are cheap to compute and when arc consistency is only used as a preprocessing step, AC-3 is not a bad choice. It has the advantage of its simple data structure. But, bear in mind that the final goal is almost always to

		RLFAP#3	RLFAP#5	RLFAP#8	RLFAP#11	Zebra
AC-3	#ccks	615,371	1,735,239	2,473,269	971,893	1090
	time	0.25	0.63	0.84	0.36	0.00
AC-4	#ccks	8,654,624	4,793,910	11,292,774	11,684,448	2299
	time	4.75	3.06	5.88	6.34	0.00
AC-6	#ccks	615,371	1,114,185	2,347,840	971,893	990
	time	0.69	1.27	2.68	1.18	0.00
AC-6p	#ccks	615,371	1,054,602	771,535	971,893	780
	time	0.78	1.19	0.86	1.19	0.00
AC-7	#ccks	412,594	848,438	654,086	638,932	582
	time	0.69	1.16	0.84	1.06	0.00
AC-I	#ccks	214,023	561,115	451,212	330,671	324
	time	0.55	1.09	0.87	0.84	0.00
AC-Ir	#ccks	3,543	145,080	133,528	3,950	42
	time	0.19	0.47	0.34	0.27	0.00

Table 2: RLFAPs and Zebra problem: Arc consistency results in number of constraint checks (#ccks) and in cpu time in seconds (time) on a PC Pentium 200MHz.

find a solution to the CSP, and thus to maintain arc consistency during search. In that case, even with cheap constraint checks, the cost of the initialization of the data structures in the first arc consistency call (before search) becomes negligible compared to the cost of the search process. And a great part of the cpu time of the search process is spent propagating deletions of values and discovering domain wipe out. (The more a problem is difficult to solve, the more a search algorithm will try values for variables, and therefore will spend time propagating deletions and finding domain wipe out.) As an example, the time needed to find a solution in the RLFAP#11 is 16.59 sec. for MAC3, 36.31 sec. for MAC4, 5.05 sec. for MAC6p[8], 4.84 sec. for MAC7, and 2.98 sec. for MACIr, with dom/deg as variable ordering heuristic [BR96]. (MACx means MAC with AC-x as the arc consistency algorithm.)

Finally, it should be noted that on very large instances of non-random problems, where the constraints are not given in extension (i.e., the size of the encoding is in $O(nd + e)$ instead of $O(nd + ed^2)$), the $O(ed^2)$ space complexity of AC-4 and AC-Inference can prevent their use.

6 Conclusion

Metaknowledge about constraints can be used to infer, rather than compute, support information, and to both infer and avoid constraint checks. A variety of basic properties of constraints can be exploited in this way. The AC-Inference schema permits exploitation of inferences in establishing arc consistency. AC-7 is a refinement of that schema that exploits only the bidirectionality of support; thus is a fully general arc consistency algorithm. AC-7 can exhibit significant

[8]Note that MAC6 and MAC6p are almost the same, differing only in the first arc consistency phase.

savings over previous general arc consistency algorithms. AC-Inference may prove especially useful when constraints have strong structural properties, when some constraints appear repeatedly, or when support is costly to compute.

Acknowledgements

This material is based in part on work supported by the National Science Foundation under Grant No. IRI-9207633 and No. IRI-9504316, and by an "action incitative CNRS/NSF" under Grant No. 0690. We thank Thomas Schiex and the French Centre d'Electronique de l'Armement for making RLFAP available, and Alan Mackworth for a helpful discussion.

Appendix

In this paper, we presented two algorithms, AC-Inference and AC-7, in their basic version, i.e., when the goal is to achieve arc consistency as a preprocessing step in a given constraint network. However, it has been shown recently that maintaining arc consistency during search is a worthwhile approach [SF94, BFR95, BR96, GS96b]. In that latter case, if we except the first pass before any instantiation, maintaining arc consistency only involves propagating deletions of values. And then, a "value-deletion" oriented presentation of the algorithms would be easier to use than the "seek-support" oriented presentation we have chosen in this paper, even if the whole algorithm is much simpler in the latter case.

In this appendix, we just give a "value-deletion" oriented framework for AC-Inference and AC-7. The procedure `ProcessDeletion` is no longer used, but the function `SeekCurrentSupport` is not changed. This is not a "two-pass" organization as in AC-4, or AC-6. As in the basic versions of AC-Inference and AC-7 (with the SeekSupportSet handled as a stack), the deletions are propagated before the whole initialization process is achieved.

The main algorithm (see Algorithm 3) is written to be used only at the very beginning of a MAC procedure [SF94, Rég95]. The function `Propagation` (see Algorithm 4) is called at each instantiation during search to propagate the deletion of the non-chosen values of the variable instantiated, and after each failure, to propagate the deletion of the previously chosen value since it leads to a dead-end ("refutation").

References

[BC93] C. Bessière and M.O. Cordier. Arc-consistency and arc-consistency again. In *Proceedings AAAI'93*, pages 108–113, Washington D.C., 1993.

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

Algorithm 3: `AC-Inference` and AC-7: "value-deletion" oriented

```
begin
    DeletionSet ← ∅;
    Initialize the data structures as in the basic versions of AC-Inference (lines
    1 and 2), and of AC-7 (lines 2 to 3) ;
    for each variable I do
        for each a ∈ D_I do
            for each J/R_IJ exists while a ∈ D_I do
                c ← SeekCurrentSupport(I, a, J);
                if c ≠ nil then
                    put a in CS[J, c, I];
                else
                    remove a from D_I;
                    put (I, a) in DeletionSet ;
        if (D_I = ∅) or (¬Propagation(DeletionSet)) then
            exit ("wipe out");
end
```

Algorithm 4: Subprocedure

```
function Propagation (in out DeletionSet: set): Boolean
    /* returns true iff the network does not have any empty domains */ ;
    begin
        while DeletionSet ≠ ∅ do
            pick (J, b) from DeletionSet ;
            for each I/R_JI exists do
                for each a ∈ CS[J, b, I] do
                    delete a from CS[J, b, I];
                    if a ∈ D_I then
                        c ← SeekCurrentSupport(I, a, J);
                        if c ≠ nil then
                            put a in CS[J, c, I];
                        else
                            remove a from D_I;
                            if D_I = ∅ then return false ;
                            put (I, a) in DeletionSet ;
        return true ;
    end
```

[BFR95] C. Bessière, E. C. Freuder, and J. C. Régin. Using inference to reduce arc consistency computation. In *Proceedings IJCAI'95*, pages 592–598, Montréal, Canada, 1995.

[BR95] C. Bessière and J.C. Régin. Using bidirectionality to speed up arc-consistency processing. In M. Meyer, editor, *Constraint Processing*, volume 923 of *LNCS*, pages 157–169. Springer–Verlag, Heidelberg, 1995.

[BR96] C. Bessière and J.C. Régin. MAC and combined heuristics: two reasons to forsake FC (and CBJ?) on hard problems. In *Proceedings CP'96*, pages 61–75, Cambridge MA, 1996.

[BR97] C. Bessière and J.C. Régin. Arc consistency for general constraint networks: preliminary results. In *Proceedings IJCAI'97*, pages 398–404, Nagoya, Japan, 1997.

[DV91] Y. Deville and P. Van Hentenryck. An efficient arc consistency algorithm for a class of csp problems. In *Proceedings IJCAI'91*, pages 325–330, Sydney, Australia, 1991.

[FBDR96] D. Frost, C. Bessière, R. Dechter, and J.C. Régin. Random uniform csp generators. URL: http://www.ics.uci.edu/~dfrost/csp/generator.html, 1996.

[Fre78] E.C. Freuder. Synthesizing constraint expressions. *Communications of the ACM*, 21(11):958–966, Nov 1978.

[Fre95] E.C. Freuder. Using metalevel constraint knowledge to reduce constraint checking. In M. Meyer, editor, *Constraint Processing*, volume 923 of *LNCS*, pages 171–184. Springer-Verlag, Heidelberg, 1995.

[Gas78] J. Gaschnig. Experimental case studies of backtrack vs waltz-type vs new algorithms for satisficing assignment problems. In *Proceedings CCSCSI'78*, pages 268–277, 1978.

[GMP+97] I.P. Gent, E. MacIntyre, P. Prosser, P. Shaw, and T. Walsh. The constrainedness of arc consistency. In *Proceedings CP'97*, pages 327–340, Linz, Austria, 1997.

[GS96a] S.A. Grant and B.M. Smith. The arc and path consistency phase transitions. In *Proceedings CP'96*, pages 541–542, Cambridge MA, 1996.

[GS96b] S.A. Grant and B.M. Smith. The phase transition behavior of maintaining arc consistency. In *Proceedings ECAI'96*, pages 175–179, Budapest, Hungary, 1996.

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

[MF93] A.K. Mackworth and E.C. Freuder. The complexity of constraint satisfaction revisited. *Artificial Intelligence*, 59:57–62, 1993.

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

[Per92] M. Perlin. Arc consistency for factorable relations. *Artificial Intelligence*, 53:329–342, 1992.

[Rég95] J.C. Régin. *Développement d'outils algorithmiques pour l'Intelligence Artificielle. Application à la chimie organique.* PhD thesis, University of Montpellier II, December 1995. in French.

[SF94] D. Sabin and E.C. Freuder. Contradicting conventional wisdom in constraint satisfaction. In *Proceedings PPCP'94*, Seattle WA, 1994.

[VDT92] P. Van Hentenryck, Y. Deville, and C.M. Teng. A generic arc-consistency algorithm and its specializations. *Artificial Intelligence*, 57:291–321, 1992.

[Wal93] R.J. Wallace. Why ac-3 is almost always better than ac-4 for establishing arc consistency in csps. In *Proceedings IJCAI'93*, Chambéry, France, 1993.

[WF92] R.J. Wallace and E.C. Freuder. Ordering heuristics for arc consistency algorithms. In *Proceedings 9th Canadian Conference on Artificial Intelligence*, pages 163–169, Vancouver, Canada, 1992.