Using Inference to Reduce Arc Consistency Computation

Christian Bessiere
LIRMM (UMR 9928 CNRS), 161 rue Ada
34392 Montpellier cedex 5, France
phone: (33) 67 41 85 39
fax: (33) 67 41 85 00
e-mail: bessiereClimn.fr

Eugene C. Freuder
University of New Hampshire
Durham, NH 03824 USA
phone: 603-862-1867
fax: 603-862-3493
e-mail: ecfCcs.unh.edu

Jean-Charles Regin
GDR 1093 CNRS, LIRMM, 161 rue Ada
34392 Montpellier cedex 5, Prance
phone: (33) 67 41 85 80
fax: (33) 67 41 85 00
e-mail: reginClinnm.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 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 on real-world problems.

1 Introduction

1.1 Overview

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 we restrict our attention here to *binary* CSPs, where the constraints involve two variables.

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 *Dj.* 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.*

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. 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 make 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* 6 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 [Mackworth and Freuder, 1993]. 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 [Freuder, 1978].

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 [Mackworth, 1977] 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 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 [Bessiere and Cordier, 1993].

1.2 Relation to Previous Work

For some time the state of the art resided in two algorithms, AC-4 [Mohr and Henderson, 1986], which has optimal worst-case behavior, and AC-3 [Mackworth, 1977], which often exhibits better average-case behavior [Wallace, 1993]. Two AC-5 algorithms, one by Deville and Van Hentenryck [Deville and Hentenryck, 1991] and another by Perlin [Perlin, 1992], permit exploitation of certain specific constraint structures, but reduce to AC-3 or AC-4 in the general case. Recently Bessi6re and Cordier have developed AC-6, which retains the optimal worst-case behavior of AC-4 while improving on the average-case behavior of AC-3 [Bessiere and Cordier, 1993]. 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 by the authors [Bessiere and Regin, 1994; 1995; Freuder, 1995].

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 [Gaschnig, 1978] 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* 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

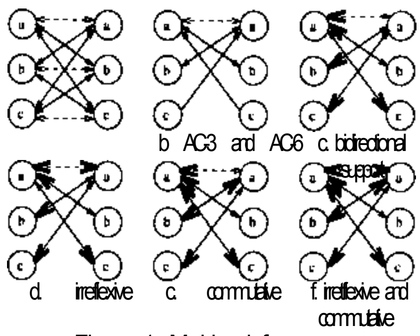

Figure 1: Making inferences.

demonstrated for AC-6, plus the additional advantage provided by inferences based on bidirectionality.

2 Inferring Support

2.1 Principle

We illustrate the principle of inferring support with a simple coloring problem example. Consider a trivial one for illustration purposes. 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. As a CSP, the countries are the variables, the colors the values. The constraint between the two variables specifics that the two countries cannot have the same color (i.e. it is a not-equal constraint).

A value, (z,a), is *arc inconsistent* if there is a variable for which there is no value that supports *(i,a).* We achieve *arc consistency* by removing all arc inconsistent values or in this simple coloring problem by 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 la. The values for country *X* are on the left, for country *Y* on the right. An arrow from a value *u* for A" to a value *v* for *Y* indicates that a check of the consistency of *(X,u)* with *(Y,v),* (denoted by *RXY(U,V)),* has been computed while seeking support for (A', *u).* The double ended arrows indicate that redundant 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 8 constraint checks here (Figure Ib), but some of them are redundant. 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. If the support it finds initially is deleted, 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 8 constraint checks.

AC-3 may check pairs once, twice, or more than twice. 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.) 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,* 6), and then separately checks that *(Y, b)* is supported by *(X,a).* These algorithms can not say: "oh, I already know *(Y,b)* is supported; I just found that out while looking for support for (X,a)".

The AC-7 algorithm that we propose here *can,* in effect, say 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-7 only requires 5 constraint checks to establish arc consistency. This is shown in Figure Ic. Large barbed arrowheads are used to indicate inferences. The double-ended arrows here, with a barb 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).

Moreover, AC-7 is but one instance of the general principal 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 these constraints. Suppose we know that they are also *irreflexive: Rxy(v,v)* does not hold for any *v.* Then the number of checks can be reduced to 4. Irreflexivity allows us to immediately infer that *(X,a)* is inconsistent with *(Y,a),* making it unnecessary to check this (Figure Id). Suppose we know that the constraints are *commutative: Rxy(u,v) = RXY(V,U).* This permits us to reduce the number of constraint checks to 3. For example, after checking *Rxy(a,b)* we can infer *Rxy(b, a).* See Figure Ie. If we know they are irreflexive and commutative, we are down to two checks, Figure If.

Finally, suppose we know that the constraints are inequality constraints. What does it mean to know that they are inequality constraints? Well one thing it could mean is that we have an inference rule that says simply: if the constraints are inequality constraints, and the domains contain more than one element, forget arc consistency processing—the problem is already arc consistent. This reduces the processing to zero constraint checks. Less dramatically, but potentially of considerable significance, we may 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 they share common domains).

2.2 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 values need to be rechecked to see if they are still supported. AC-4 computes a total support count initially and then updates it as values are deleted. If AC-Inference 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-Inference looks for another supporting value later in the list.

The schema data structures are something of a hybrid of AC-4 and AC-6. Conceptually, for each value, *a,* for each variable, i, and each other variable *j,* which shares a constraint with i, AC-Inference maintains a *support set* of values from *Dj* that support *(i,a),* and the *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 neither set.)

Maintaining these sets enables the schema to remember inferred constraint check information. Positive results are remembered by adding 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; unlike AC-6 we can avoid some of those values via inference.

The support data structures of AC-4 accept additional support during the initialization phase and can be modified to reflect deletion of support. For the schema, support can be both added and deleted throughout the processing. AC-6 proceeds through the variable domains as necessary looking for support, ignoring deleted values. The schema does the same, ignoring also values whose support status has already been inferred. The number of supporting and unchecked values can be maintained to facilitate deletion decisions.

The schema can be expressed as a processor for two streams.

```
AC-Inference Schema:
  Initialize the seek-support stream
  Establish the initial support and unchecked
      sets, using any initial inferences
  Repeat until done:
     If process-deletion stream is not empty,
          process its first element
     else if seek-support stream is not empty,
          process its first element
     else done
```

The *seek-support* stream 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 stream. Processing an element of this stream involves looking for support for the value in the unchecked set of the variable. After each constraint check a set of inferences is made, based on the outcome. The specific set of inferences chosen instantiates the schema. The process-deletion stream consists of values whose support and unchecked sets at some variable are both empty. Processing a deletion involves removing it, at least conceptually, from the domain of values for its variable, and the support and unchecked sets in which it appears.

When both the support and unchecked sets for a value with respect to a variable become empty, that value enters the process-deletion stream. This can happen either as the result of seeking support or processing deletion. When only the support set for a value with respect to a variable becomes empty, that value enters the seek-support stream. This can happen as the result of processing a deletion. Notice that we respond to deletions immediately. Furthermore we recommend implementing the seek-support stream 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.

3 AC-7

AC-7 refines the AC-Inference 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 restricted our inferences to bidirectionality we are able to maintain a space complexity of the same order as AC-6, namely O(ed), for a problem with e constraints and maximum domain size d. 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. Swapping slows down running time. AC-7 avoids the space requirements of AC-Inference by using an AC-6-like data structure to avoid the constraint checks that would be inferred by AC-Inference.

AC-7 has the following desirable properties. AC-7:

1. never checks $Ri_{ij}a,b)$ if there exists b' still in D_j such that $R_{ij}(a,b')$ has already been successfully checked.
2. never checks Rij(a,b) if there exists b' still in D_j such that Rji(b',a) has already been successfully checked.
3. never checks $R_{ij}(a,b)$ if:
   a) it has already been checked
   or b) $R_{Jt}(b,a)$ has been checked.
4. has 0(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.

3.1 Data structure

Every time a constraint check Rji(b,a) is performed, AC-7 needs to store this information to avoid future Rij(a, b) or Rij(a, b') checks forbidden by properties (3b) and (2). This storage must be done in a careful way to avoid falling into the trap of an $0(ed^2)$ space complexity. 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) 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 (7,6) is removed from Dj 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 are split into S[(i,j),a] sets, to support property (2): each value (i,a) has a set of values supported by it per constraint instead of one for all the constraints. b E S[(i,j),a] implies (i,a) is known as supporting (j',b) on R_{Ji} then (i,a) is compatible with (j,b). Hence, an $R_{ij}a.b)$ is checked if and only if S[(iJ),a] = 0, otherwise we know (i, a) has a support on R_{iJ} without any new constraint check. Also, arrays inf-support are added to support property (3b) for negative constraint checks and property (3a)[1]: inf-support[(i, j), a] represents the smallest value in D_j which may be compatible with (i,a).

In more detail the data structure of AC-7:

- A table M of Booleans keeps track of which values of the initial domain are in the current domain or not (M[i,a]=true ⇔ a € Di). In this table, each initial D_i is considered as the integer range I..$|D_i|$. We use the following constant time functions and procedures to handle D_i lists. last(Di) returns the greatest value in D_i if D_i = 0, else returns 0. If a € D_i\last(D_l), next(a,Di) returns the smallest value in D_i greater than a. remove(a,D_i) removes the value a from D_i. remove(a, D_i) can be extended to stop the arc consistency algorithm if a was the only remaining value in D_i. The CSP, indeed, has no solution.
- For all a in D_t, S[(i,j), a] contains all values b in Dj for which (i,a) is assigned as its current support. The current support is not necessary the smallest one, as opposed to AC-6.
- Arrays inf-support are updated by AC-7 to ensure that every b in D_j compatible with (i,a) is greater than or equal to inf-support[(i,j),a].
- Deletionstream and SeekSupportStream have the same behavior as in AC-Inference. Handling the SeekSupportStream as a stack seems to be an efficient heuristic because it propagates the consequences of deletions as soon as they appear, and then discovers empty domains earlier.

3.2 Algorithm

The algorithm AC-7 (see Fig. 2), has the same framework as AC-Inference. There are two main operations: processing the deletion of a value, and seeking a new support for a value. AC-7 processes the deletion of a value (j, b) by adding in the SeekSupportStream all the value-variable pairs [(i,a), j] such that (i,a) was supported by

[1] Property (3a) held in AC-6 without inf-support 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.

(j,b) (i.e. values (i,a) that were in the S[(j,i),b] sets). The search on R_{ij} for a new support for a value (i,a) is only done if S[(i,j),a] does not contain values in Dj\ otherwise, this search is useless: (i,a) has inferable supports on R_{ij} (since values supported by (i,a) are supports for (i,a)). If the search of support is done for a value (i,a) on R_{ij} and a value c is found, a is added to S[(j,i),c] since c is now the current support of (i,a). inf-support[(i, j),a] records c as the smallest value in D_j supporting (i, a) on R_{ij} since the search follows the order of Dj. During the whole algorithm, every time a value is found without support on a constraint, it is removed from D and put in the DeletionStream.

```
DeletionStream ← ∅;
SeekSupportStream ← ∅;
for each $(i,a) \in D$ do $M[i,a]$ ← true;
for each $(i,j)/R_{ij}$ exists do
    for each $a \in D_i$ do
        $S[(i,j),a] \leftarrow \emptyset$;
        inf-support$[(i,j),a] \leftarrow 1$ /* first value of $D_j$ */ ;
        put $[(i,a),j]$ in SeekSupportStream;
repeat until done
    if DeletionStream $\neq \emptyset$ then
            pick $(j,b)$ from DeletionStream;
            for each $i/R_{ji}$ exists do
                for each $a \in S[(j,i),b]$ do
                    delete $a$ from $S[(j,i),b]$;
                    put $[(i,a),j]$ in SeekSupportStream;
    else if SeekSupportStream $\neq \emptyset$ then
            pick $[(i,a),j]$ from SeekSupportStream;
            if $M[i,a]$ then
                if SeekInferableSupport$((i,j),a,c)$
                then put $a$ in $S[(j,i),c]$
                else $c \leftarrow$ inf-support$[(i,j),a]$;
                    if SeekNextSupport$((i,j),a,c)$
                    then put $a$ in $S[(j,i),c]$;
                         inf-support$[(i,j),a] \leftarrow c$
                    else remove$(a,D_i)$; $M[i,a]$ ← false;
                         put $(i,a)$ in DeletionStream;
    else done
```

Figure 2: AC-7.

The function SeekNextSupport (Fig. 3), which looks for the smallest value in Dj not smaller than b and supporting (i,a) on Rij, is the same as in AC-6 except the addition of the test of line 3 to avoid checking Rij(a,b) when Rj_t(b,a) has already been checked as false (i.e. when we know that (j, b) has no support in Di until a value greater than a).

Another function, SeekInferableSupport (Fig. 3), is used in AC-7 to check whether a value (i, a) has inferable supports on R_{ij} or not (i.e. whether S[(i,j),a] contains values in Dj or not). Every time SeekInferableSupport unsuccessfully checks if a value b in S[(i,j),a] is in D_j, b is deleted from S[(i,j),a] to avoid checking it again in the following.

3.3 Analysis

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

By the same principle as with AC-6, properties (1)

```
function SeekNextSupport(in $(i,j)$ : arc; in $a$ : integer;
                            in out $b$ : integer):Boolean;
    /* returns true iff a value $b$ is found */
1   if $b >$ last$(D_j)$ then return false;
2   while not $M[j,b]$ do $b \leftarrow b+1$;
3   if inf-support$[(j,i),b] \leq a$ then
4           if $R_{ij}(a,b)$ then return true;
5   if $b <$ last$(D_j)$ then
            $b \leftarrow$ next$(b,D_j)$;
            goto line 3;
6   return false;

function SeekInferableSupport(in $(i,j)$ : arc; in $a$ : integer;
                                 out $b$ : integer):Boolean;
    /* returns true iff a value $b$ is found */
1   if $S[(i,j),a] = \emptyset$ then return false;
2   $b \leftarrow$ an element of $S[(i,j),a]$;
3   if $M[j,b]$ then return true;
4   delete $b$ from $S[(i,j),a]$;
5   goto line 1;
```

Figure 3: Subprocedures.

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 $S[(i,j),a]$. Since AC-7 only looks for a support for (i,a) on R_{ij} if $S[(i,j),a] \cap D_j = \emptyset$, we are sure that $R_{ij}(a,b)$ is only checked 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 $S[(i,j),a]$ or out of D, and $R_{ij}(a,b)$ cannot be checked. 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 $b \in D_j$, *inf-support*$[(j,i),b]$ is greater than a, and line 3 of function *SeekNextSupport* avoids checking $R_{ij}(a,b)$. Property (3b) holds.

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 $S[(j,i),b]$ sets. This does not increase their total size: each arc-value pair $[(i,j),a]$ has at most one current support (j,b) (i.e. a belonging to $S[(j,i),b]$). Hence, the total size of the $S[(j,i),b]$ sets is bounded above by $O(ed)$. We have to add arrays *inf-support*, which take a $2ed$ space. Thus, the total space complexity of AC-7 remains $O(ed)$.

Note that AC-7 can do much more than avoid the computation of *inverse* checks that establish whether (i,a) supports (j,b) after establishing whether (j,b) supports (i,a). Inferring one support may permit us to avoid a number of additional constraint checks.

We will demonstrate the savings that are possible for AC-7 with a simple example. Figure 4 shows the microstructure of the example in the form of a *consistency graph*, where each consistent pair of values is joined by an edge. The problem involves just two variables, as-

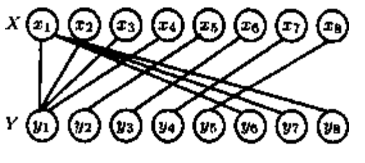

Figure 4: Analysis.

suming the variables are ordered from top to bottom. In this example the number of checks for AC-7 is 21; for AC-3 or AC-6, 48, for AC-4, 64.

This example can be generalized from domain sizes of 8 to domain sizes of d, where $0(d^2)$ checks are saved. Observe that it is constructed so that d/2 of the values in the second domain (y_2 to y_5) 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. Thus we have established the following:

Theorem 1 AC-7 can save $0\{d^2)$ non-inverse constraint checks for domain size d.

Notice that we have established this theorem without even considering constraint propagation, where further savings may be realized. Bear in mind that inferring non-support and utilizing it immediately can justify the deletion of values. Early deletions of values can lead to further savings, as we no longer have to seek support for these deleted values, or consider them as possible supports for other values. It may even lead to early discovery of unsolvability. 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.

We would expect AC-7 to show to best advantage on problems with structures such as those above, that can especially benefit from the undirected constraint inferences. Changing the order in which values are considered can increase (or reduce) the advantage of AC-7, but since these advantages are tied to the structure of the consistency graph, and since determining consistency graph structure involves performing the very constraint checks we are trying to avoid, it is difficult to argue strongly for or against AC-7 based on such ordering considerations. However, in specific cases, we might be able to utilize metaknowledge of the consistency graph structure for effective value-ordering heuristics.

We have been focusing on constraint checks, which are a standard measure of CSP algorithm performance. 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 concems. AC-7 can even permit us to cope with some situations where some constraint checks cannot be computed at all.

4 Experimental results

We tested AC-Inference and AC-7 on two real-world problems. (Since AC-Inference is an algorithm that takes advantage of constraint semantics, we would not expect it to do best on random problems.) The first is 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 maintained by Michael Jampel (ftp.cs.city.ac.uk:/pub/constraints/csp-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. The second is a RNA problem in molecular biology. The goal is to forecast the structure of RNA molecules given the sequence of nucleotides. The CSPs involved are exceptionally large: 70-1542 variables, 40-550 values per domain, and a complete graph of loose constraints which are irreflexive. The task is to achieve arc consistency alone. We also tested on the Zebra problem, a common CSP benchmark (also available in the benchmarks archive).

We compared several versions of the MAC algorithm [Sabin and Freuder, 1994] on RLFAP and Zebra. MAC is a backtrack search algorithm where arc consistency is dynamically maintained as search choices are made. The versions of MAC we tested differ in the type of arc consistency algorithm they use. *MAC4* is based on AC-4[2], *MAC6* on AC-6 (two-pass organization), *MAC6+heuristic* on an AC-6 version implemented with our heuristic (processing deletions first), *MAC7* on AC-7, *MAC-I* on AC-Inference when reflexivity, commutativity and irreflexivity are used, and finally, *MAC-I+repeated*, based on AC-Inference, when, in addition, repeated constraints are taken into account[3]. On the RNA problem, we compared AC-4, AC-6, AC-7 and AC-Inference. All the results are given in Table 1.

Experimental results give us some interesting information. First, the number of constraint checks is strongly correlated with cpu time, even on this kind of problem where constraint checks are cheap to compute. This means that, generally, the benefit of inferences not only offsets, but also overcomes their cost. (Otherwise, "#ccks" would have decreased from *MAC4* to *MAC-I+repeated* while "time" would not). The very good running times of *MAC-I+repeated* confirm the advantage of using the most metaknowledge we can. Second, the results of *MAC6* and *MAC6+heuristic* show the effectiveness of our heuristic (processing deletions first) on problems with a large propagation at the first arc consistency call[4] (e.g. RLFAP #5—12046 values deleted, RLFAP #8—arc inconsistent, as opposed to RLFAP #3—0 values deleted, RLFAP #11—0 values deleted). Finally, the

[2]The number of constraint checks for MAC4 is the same as the number of constraint checks for the first AC-4 call, since all support information is then in the support sets.

[3]MACS was not used because it has a behavior really worse than the other algorithms, except on very easy instances where it can outperform MAC4-

[4]because MAC6 contains the expensive initialization phase of AC-6 (which is only performed at the first call), while MAC6+heuristic already benefits from the good behavior of the heuristic.

		RLFAP#3	RLFAP#5	RLFAP#8	RLFAP#11	Zebra
MAC4	#ccks	8 922 424	4 897 940	12 350 516	13 050 704	2421
	time	49.08	26.27	59.20	227.33	0.02
MAC6	#ccks	995 261	1 119 334	2 477 957	1 666 632	1117
	time	8.70	6.65	14.25	28.47	0.00
MAC6 + heuristic	#ccks	995 261	903 902	713 167	1 666 632	1117
	time	8.70	5.95	4.73	28.47	0.00
MAC7	#ccks	714 362	685 248	585 021	1 167 695	756
	time	7.82	5.42	4.18	27.80	0.00
MAC-I	#ccks	442 935	451 529	394 631	712 399	403
	time	7.28	5.00	3.78	27.25	0.00
MAC-I + repeated	#ccks	19 661	134 384	122 579	24 467	42
	time	3.43	2.80	2.22	21.37	0.00

		tRNAthrTcoli	HIV1	rnasePcoli
AC-4	#ccks	$26.9\ 10^6$	$188\ 10^6$	???
AC-6	#ccks	$2.57\ 10^6$	$15.0\ 10^6$	$501\ 10^6$
AC-7	#ccks	$2.32\ 10^6$	$14.1\ 10^6$	$478\ 10^6$
AC-Inference	#ccks	$2.27\ 10^6$	$13.8\ 10^6$	???

Table 1: Results in number of constraint checks (#ccks) and in cpu time in seconds (time) on a SPARC 10 machine with 32 megabytes of memory.

RNA problem shows the consequences of tackling space complexity: AC-4 and AC-Inference could not process sequences larger than "HIV1" (124 variables, an average of 47 values per domain, 7626 constraints), while AC-6 and AC-7 could process "rnasePcoli" (377 variables). On the RNA problem we can also remark on the small differences in performance between AC-6, AC-7 and AC-Inference, explained by the looseness of the constraints (finding a support is easy).

5 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 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. We thank Thomas Schiex and the French Centre d'Electronique de l'Armement for making RLFAP available, Christine Gaspin for providing the RNA problem and Alan Mackworth for a helpful discussion.

References

[Bessiere and Cordier, 1993] C. Bessiere and M.O. Cordier. Arc consistency and arc consistency again. In Proc. AAAI'98.

[Bessiere and Regin, 1994] C. Bessiere and J.C. Regin. An arc-consistency algorithm optimal in the number of constraint checks. In Proc. IEEE TAI'94-

[Bessiere and Regin, 1995] C. Bessiere and J.C. Regin. Using bidirectionality to speed up arc-consistency processing. In M. Meyer, editor, Constraint Processing: Selected Papers from the ECAI'94 Workshop. Springer-Verlag, 1995. to appear.

[Deville and Hentenryck, 199l] Y. Deville and P. Van Hentenryck. An efficient arc consistency algorithm for a class of csp problems. In Proc. IJCAI'91.

[Freuder, 1978] E.C. Freuder. Synthesizing constraint expressions. CACM, 21:958-966, 1978.

[Freuder, 1995] E.C. Freuder. Using metalevel knowledge to reduce constraint checking. In M. Meyer, editor, Constraint Processing: Selected Papers from the ECAI'94 Workshop. Springer-Verlag, 1995. to appear.

[Gaschnig, 1978] J. Gaschnig. Experimental case studies of backtrack vs waltz-type vs new algorithms for satisficing assignment problems. In Proc. CCSCSI'78.

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

[Mackworth, 1977] A. Mackworth. Consistency in networks of relations. Artificial Intelligence, 8, 1977.

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

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

[Sabinand Freuder, 1994] D. Sabin and E.C. Freuder. Contradicting conventional wisdom in constraint satisfaction. In Proc. ECAI'94.

[Wallace, 1993] R. Wallace. Why AC-3 is almost always better than AC-4 for establishing arc consistency in CSPs. In Proc. IJCAI'93.