Arc-Consistency and Arc-Consistency Again

Christian Bessière

LIRMM, University of Montpellier II
161, rue Ada
34392 Montpellier Cedex 5, FRANCE
Email: bessiere@lirmm.fr

Marie-Odile Cordier

IRISA, University of Rennes I
Campus de Beaulieu
35042 Rennes, FRANCE
Email: cordier@irisa.fr

Abstract

Constraint networks are known as a useful way to formulate problems such as design, scene labeling, temporal reasoning, and more recently natural language parsing. The problem of the existence of solutions in a constraint network is NP-complete. Hence, consistency techniques have been widely studied to simplify constraint networks before or during the search of solutions. Arc-consistency is the most used of them. Mohr and Henderson [Moh&Hen86] have proposed AC-4, an algorithm having an optimal worst-case time complexity. But it has two drawbacks: its space complexity and its average time complexity. In problems with many solutions, where the size of the constraints is large, these drawbacks become so important that users often replace AC-4 by AC-3 [Mac&Fre85], a non-optimal algorithm. In this paper, we propose a new algorithm, AC-6, which keeps the optimal worst-case time complexity of AC-4 while working out the drawback of space complexity. More, the average time complexity of AC-6 is optimal for constraint networks where nothing is known about the semantic of the constraints. At the end of the paper, experimental results show how much AC-6 outperforms AC-3 and AC-4.

1. Introduction

There is no need to show the importance of arc-consistency in Constraint Networks. Originating from Waltz [Waltz72], who developed it for vision problems, it has been studied by Mackworth and Freuder [Mackworth77], [Mac&Fre85], by Mohr and Henderson [Moh&Hen86] who have proposed an algorithm having an optimal worst-case time complexity: $O(ed^2)$, where e is the number of constraints (or relations) and d the size of the largest domain. In [Bessière91] its use has been extended to Dynamic constraint networks. Recently, Van Hentenryck, Deville and Teng [Dev&VanH91], [VanH&al92], have proposed a generic algorithm which can be implemented with all known techniques, and have extracted classes of networks on which there exist algorithms running arc-consistency in $O(ed)$. In 1992, Perlin [Perlin92] has given properties of arc-consistency on factorable relations.

Everybody now looks for arc-consistency complexity in particular classes of constraint networks because AC-4 [Moh&Hen86] has an optimal worst-case complexity and it is supposed that we cannot do better.

But AC-4 drawbacks are its average time complexity which is too much near the worst-case time complexity and more, its space complexity which is $O(ed^2)$. In applications with a large number of values in variables domains and with weak constraints, AC-3 is often used instead of AC-4 because of its space complexity. Such situations appear for example when domains encode discrete intervals and constraints are defined as arithmetic relations ($\geq$, $<$, $\neq$,…). Constraint Logic Programming (CLP) languages [Din&al88] which are big consumers of arc-consistency (arc-consistency has some good properties in CLP) are concerned by these problems.

In problems with many solutions, where the constraints are weak, AC-4 initialization step is very long because it requires to consider the relations in their whole to construct its data structure. In those cases, AC-3 [Mac&Fre85] runs faster than AC-4 in spite of its non-optimal time complexity.

In this paper we propose a new algorithm, AC-6, which while keeping $O(ed^2)$ optimal worst-case time complexity of AC-4, discards the problem of space complexity (AC-6 space complexity is $O(ed)$) and checks just enough data in the constraints to compute the arc-consistent domain. AC-4 looks for all the reasons for a value to be in the arc-consistent domain: it checks, for each value, all the values compatible with it (called its supports) to prove this value is viable. AC-6 only looks for one reason per constraint to prove that a value is viable: it checks, for each value, one support per constraint, looking for another one only when the current support is removed from the domain.

The rest of the paper is organized as follows. Section 2 gives some preliminaries on constraint networks and arc-consistency. Section 3 presents the algorithm AC-6. In section 4, experimental results

show how much AC-6 outperforms the algorithms AC-3 and AC-4[1]. A conclusion is given in section 5.

2. Background

A *network of binary constraints (CN)* is defined as a set of n variables $\{i, j,\ldots\}$, a domain $D=\{D_i, D_j,\ldots\}$ where D_i is the set of possible values for variable i, and a set of binary constraints between variables. A *binary constraint* (or relation) R_{ij} between variables i and j is a subset of the Cartesian product $D_i \times D_j$ that specifies the allowed pairs of values for i and j. Following from Montanari [Montanari74], a binary relation R_{ij} between variables i and j is usually represented as a (0,1)-matrix (or a matrix of booleans) with $|D_i|$ rows and $|D_j|$ columns by imposing an ordering on the domains of the variables. Value true at row a, column b, denoted $R_{ij}(a, b)$, means that the pair consisting of the ath element of D_i and the bth element of D_j is permitted; value false means the pair is not permitted. In all the networks of interest here $R_{ij}(a, b)=R_{ji}(b, a)$. In some applications (constraint logic programming, temporal reasoning,…), R_{ij} is defined as an arithmetic relation ($=, \neq, <, \geq,\ldots$) without giving the matrix of allowed and not allowed pairs of values.

A *graph* G can be associated to a constraint network, where nodes correspond to variables in the CN and an edge links nodes i and j every time there is a relation R_{ij} on variables i and j in the CN. For the purpose of this paper, we consider G as a symetric directed graph with arcs (i, j) and (j, i) in place of the edge $\{i, j\}$.

A *solution* of a constraint network is an instanciation of the variables such that all the constraints are satisfied.

Definition. Having the constraint R_{ij}, value b in D_j is called a *support* for value a in D_i if the pair (a, b) is allowed by R_{ij} (i.e. $R_{ij}(a, b)$ is true).

A value a for a variable i is *viable* if for every variable j such that R_{ij} exists, a has a support in D_j.

The domain D of a CN is *arc-consistent* if for every variable i in the CN, all the values in D_i are viable.

3. Arc-consistency with unique support

3.1. Preamble

As Mohr and Henderson underlined in [Moh&-Hen86], arc-consistency is based on the notion of support. As long as a value a for a variable i (denoted (i, a)) has supporting values on each of the other variables j linked to i in the constraint graph, a is considered a viable value for i. But once there exists a variable on which no remaining value satisfies the relation with (i, a), then a must be eliminated from D_i.

The algorithm proposed in [Moh&Hen86] makes this support explicit by assigning a counter $counter[(i, j), a]$ to each arc-value pair involving the arc (i, j) and the value a on the variable i. This counter records the number of supports of (i, a) in D_j. For each value (j, b), a set S_{jb} is constructed, where $S_{jb}=\{(i, a) / (j, b) \text{ supports } (i, a)\}$. Then, if (j, b) is eliminated from D_j, $counter[(i, j), a]$ must be decremented for each (i, a) in S_{jb}.

This data structure is at the origin of AC-4 optimal worst-case time complexity. But computing the number of supports for each value (i, a) on each constraint R_{ij} and recording all the values (i, a) supported by each value (j, b) implies an expensive space complexity of $O(ed^2)$ (the size of the support sets S_{jb}) and an average time complexity increasing with the number of allowed pairs in the relations since the number of supports is proportional to the number of allowed pairs in the relations.

The purpose of AC-6 is then to avoid the expensive checking of the relations to find all the supports for all the values. AC-6 keeps the same principle as AC-4, but instead of checking all the supports for a value, it only 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_{jb}, 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_{jb}. The only requirement in the use of AC-6 is to have a total ordering in all domains D_j. But this is not a restriction since in any implementation, a total ordering is imposed on the domains. This ordering is independent of any ordering computed in a rearrangement strategy for searching solutions.

3.2. The algorithm

The algorithm proposed here works with the following data structure:

• A table M of booleans keeps track of which values of the initial domain are in the current domain or not ($M(i, a)=\textbf{true} \Leftrightarrow a \in D_i$). In this table, each initial D_i is considered as the integer range $1..|D_i|$. But it can be a set of values of any type with a total ordering on these values. We use the following

[1]AC-5 [VanH&al92] is not discussed here since it is not an improvement but a generic framework in which all previous algorithms can be written.

constant time functions to handle D_i sets that are considered as lists:

- *first*(D_i) returns the smallest value in D_i.
- *last*(D_i) returns the largest value in D_i.
- *next*(a, D_i) returns the value a' in D_i such that every value a'' larger than a and smaller than a' is out of D_i.

• S_{jb}={(i, a) / (j, b) is the smallest value in D_j supporting (i, a) on R_{ij}} while in AC-4 it was containing all the values supported by (j, b).

• Counters for each arc-value pair in AC-4 are not used in AC-6.

• A list *List* contains values deleted from the domain but for which the propagation of the deletion has not been processed yet.

In AC-4, when a value (j, b) was deleted, it was added to *List* waiting for the propagation of the consequences of its deletion. These consequences were to decrement *counter*[(i, j), a] for every (i, a) in S_{jb} and to delete (i, a) when *counter*[(i, j), a] becomes equal to zero. In AC-6, the use of *List* is not changed but the consequence of (j, b) deletion is now to find another support for every (i, a) in S_{jb}. Having an ordering on D_j we look after b (the old support) for another value c in D_j supporting (i, a) on R_{ij} (we know there is no such value before b). When such a value c is found, (i, a) is added to S_{jc} since (j, c) is the new smallest support for (i, a) in D_j. If no such value exists, (i, a) is removed and put in *List*.

AC-6 uses the following procedure to find the smallest value in D_j not smaller than b and supporting (i, a) on R_{ij}:

```
procedure nextsupport(in i, j, a : integer; in out b : integer;
                                     out emptysupport : boolean);
  begin
   {search of the smallest value as large as b that
   belongs to Dj ; this part is not needed in the call of
   the procedure done in the initialization step since b
   already belongs to Dj}
   while not M(j, b) and b < last(Dj) do b ← b + 1 ;
   emptysupport ← not M(j, b) ;

   {search of the smallest support for (i, a) in Dj}
   while not Rij(a, b) and not emptysupport do
      if b < last(Dj) then b ← next(b, Dj)
      else emptysupport ← true
  end;
```

The algorithm AC-6 has the same framework as AC-4. In the initialization step, we look for a support for every value (i, a) on each constraint R_{ij} to prove that (i, a) is viable. If there exists a constraint R_{ij} on which (i, a) has no support, it is removed from D_i and put in *List*.

In the propagation step, values (j, b) are taken from *List* to propagate the consequences of their deletion: finding another support (j, c) for values (i, a) they were supporting (values (i, a) in S_{jb}). When such a value c in D_j is not found, (i, a) is removed from D_i and put in *List* at its turn.

```
{initialization}
for (i, a) ∈ D do Sia ← ∅ ; M(i, a) ← true ;
for (i, j) ∈ arcs(G) do
   for a ∈ Di do
    begin
    if Dj = ∅
    then emptysupport ← true
    else  b ← first(Dj) ;
          nextsupport(i, j, a, b, emptysupport) ;
    if emptysupport
    then  Di ← Di \ {a} ; M(i, a) ← false ;
          Append(List, (i, a))
    else Append(Sjb, (i, a))
    end

{propagation}
while List ≠ ∅ do
   begin
   choose (j, b) from List and remove (j, b) from List ;
   for (i, a) ∈ Sjb do {before its deletion (j, b) was the
    begin       smallest support in Dj for (i, a) on Rij}
    remove (i, a) from Sjb ;
    if M(i, a) then
       begin
       c ← b ; nextsupport(i, j, a, c, emptysupport) ;
       if emptysupport
       then  Di ← Di \ {a} ; M(i, a) ← false ;
             Append(List, (i, a))
       else Append(Sjc, (i, a))
       end
    end
   end
```

3.3. Correctness of AC-6

Here are the key steps for a complete proof of the correctness of AC-6. In this section we denote *maxAC* the maximal arc-consistent domain which is expected to be computed by an arc-consistency algorithm.

• In AC-6, value (i, a) is removed from D_i only when it has no support in D_j on a constraint R_{ij}. If all previously removed values are out of *maxAC* then (i, a) is out of *maxAC*. *maxAC* was trivially included in D when AC-6 started. Then, by induction, (i, a) is out of *maxAC*. Thus, *maxAC* ⊆ D is an invariant property of AC-6.

• Every time a value (j, b) is removed, it is put in *List* until the values it was supporting are checked for new supports. Every time a value (i, a) is found without support on a constraint, it is removed from D. Thus, every value (i, a) in D has at least one support in D∪*List* on each constraint R_{ij}. AC-6 terminates with *List* empty. Hence, after AC-6, every

value in D has a support in D on each constraint. Thus, D is arc-consistent.

• $maxAC/D$ and D arc-consistent at the end of AC-6 imply that D is the maximal arc-consistent domain at the end of AC-6. ❑

3.4. Time and space complexity

In both the initialization step and the propagation step, the inner loop is a call to the procedure *nextsupport* which compute a support for a value on a constraint, starting at the current value. Hence, for each arc-value pair $[(i, j), a]$, each value in D_j will be checked at most once. There are *ed* arc-value pairs, thus $O(ed^2)$ is the worst-case time complexity for AC-6, as for AC-4.

The matrix M has a size proportional to the number of values in D, $O(nd)$. Arc-value pairs $[(i, j), a]$ have at most one support (j, b) with (i, a) belonging to S_{jb}; hence the total size of the S_{jb} sets is at most equal to the number of arc-value pairs: $O(ed)$. Therefore, the worst-case space complexity of AC-6 is $O(ed)$. The problem of the space complexity of AC-4 is worked out.

Having no information on the semantic of the constraints this algorithm is the best in time we can expect. It stops the processing of a value just when it has the proof it is viable (i.e. the first support). If we know something about the constraint (e.g. functional) we can locally improve the search of the support (e.g. for a functional constraint finding a support for a value is in constant time) but this is not in the topic of this paper.

4. Experimental results

Having produced an algorithm making just enough processing to ensure that each value is viable, we expect it to outperform AC-3 and AC-4 on all the problems.

We have tested the performances of the three algorithms on a large spectrum of problems. For each problem, we have counted the number of atomic operations and tests done by each algorithm.

The first comparison has been done on the zebra problem (see Appendix) which has strong similitudes with real-life problems. With the representation of this problem given in [Dechter88], we obtain the following results:

AC-3: 4008
AC-4: 3824
AC-6: 1998

Afterwards, in fig. 1, we have compared the three algorithms on a problem often used for algorithms comparisons: the n-queens (i.e. a nxn chessboard on which we want to put n queens, none of them being attacked by any other). We can encode it in a CN by representing each column by a variable which values are the rows. The graph associated to the CN is complete, each pair $\{i, j\}$ of variables being linked by a constraint that specifies the allowed positions for the two queens in the columns i and j. This CN is very particular since it is extremely symetrical and all the constraints are weak (note that arc-consistency does not discard any value in this CN). So, results obtained here cannot be generalized to other kinds of CNs. However, this CN is interesting to illustrate the behavior of the algorithms on CNs with weak constraints where arc-consistency discards few values. On these CNs, AC-4 fails while AC-3 and AC-4 have an *O(ed)* average time complexity.

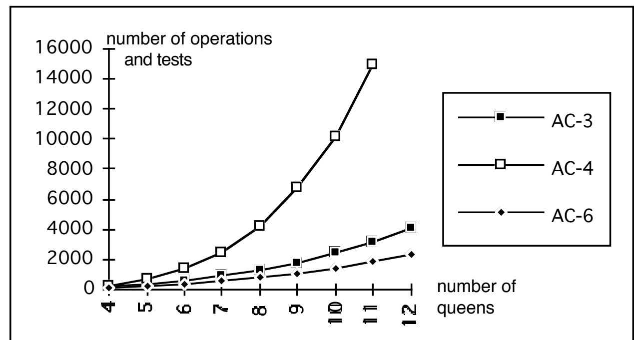

Figure 1. Comparison of AC-3, AC-4 and AC-6 on the n-queens problem

Finally, we defined classes of randomly generated constraint networks and we showed in fig. 2, 3 and 4 the behavior of the three algorithms on these different types of constraint networks. Four parameters were taken into account: n the number of variables, d the number of values per variable, pc the probability that a constraint R_{ij} between two variables exists and pu the probability in existing relations R_{ij} that a pair of values $R_{ij}(a, b)$ be allowed. The result given for each class is the average for ten instances of problems in the class, to be more representative of the class.

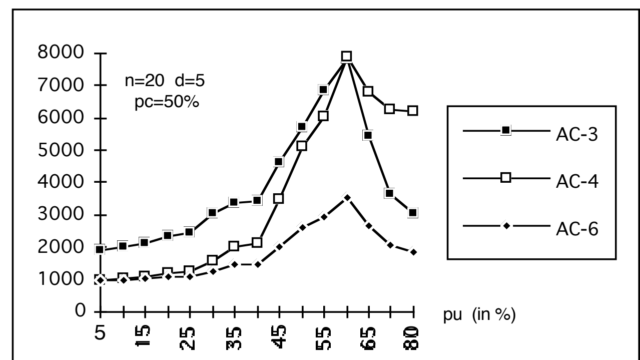

Figure 2. AC-3, AC-4 and AC-6 on randomly generated CNs with 20 variables having 5 possible values and where the probability pc to have a constraint between two variables is 50 per cent

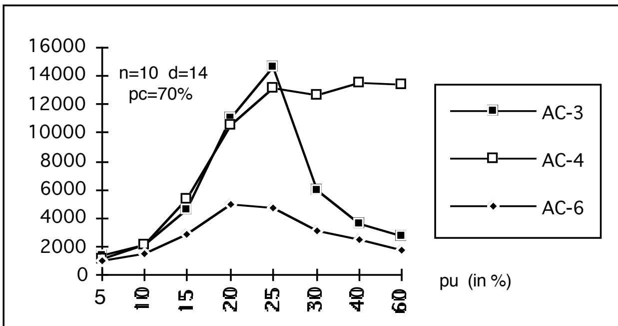

Figure 3. AC-3, AC-4 and AC-6 on randomly generated CNs with 10 variables having 14 possible values and where the probability *pc* to have a constraint between two variables is 70 %

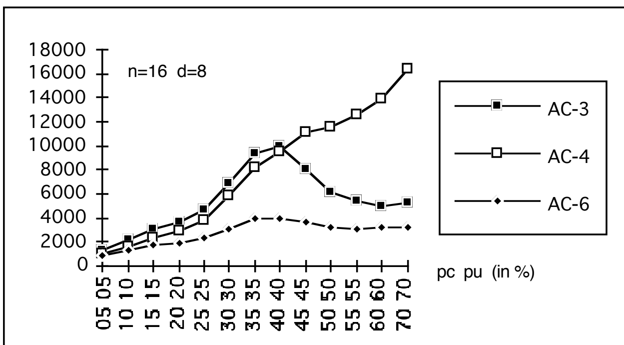

Figure 4. AC-3, AC-4 and AC-6 on randomly generated CNs with 16 variables having 8 possible values

Let summarize roughly those results:

• AC-4 performances decrease when *d* or *pu* grow. The larger the domains and the weaker the constraints, the worse AC-4. When we take a look at those figures, AC-4 seems to be not interesting. But, being good when arc-consistency discards many values, randomly generated CNs are not favourable to it. In practical cases, constraints are less homogeneous than in randomly generated CNs and AC-4 is better. Many applications, like SYNTHIA [Jan&al90] to design peptide synthesis plans, prefer AC-4 to AC-3.

• AC-3 is never very bad but it can check several times a pair of values because of its non-optimal time complexity. So, when propagation of deletions is long: in "middle" CNs (i.e. not too much constrained and not under-constrained), AC-3 becomes inefficient. However, CNs treated in practice are often "middle" CNs since under-constrained CNs and too much constrained CNs are easy to solve, a solution or a contradiction being quickly found.

• AC-6 has kept the optimal worst-case time complexity of AC-4 while working out the problem of considering the relations in their whole. Hence, it is very good on CNs with weak constraints contrary to AC-4, and remains efficient on CNs where the constraints are tighten or on "middle" CNs, contrary to AC-3.

5. Conclusion

We have provided an algorithm, AC-6, to achieve arc-consistency in binary constraint networks. It keeps the $O(ed^2)$ optimal worst-case time complexity of AC-4 while working out the two drawbacks of this algorithm: its space complexity ($O(ed^2)$), and its average time complexity on constraint networks with weak constraints. AC-6 has an $O(ed)$ space complexity and its running-time decreases when the weakness of the constraints grows. Experimental results are given, showing that AC-6 outperforms AC-3 and AC-4 (the two other best algorithms to achieve arc-consistency) on all the problems tested.

Acknowledgments

We want to thank Amit Bellicha for his useful comments on the previous draft.

Appendix: The zebra problem

1. There are five houses, each of a different color and inhabited by men of different nationalities, with different pets, drinks, and cigarettes.
2. The Englishman lives in the red house.
3. The Spaniard owns a dog.
4. Coffee is drunk in the green house.
5. The Ukranian drinks tea.
6. The green house is immediatly to the right of the ivory house.
7. The Old-Gold smoker owns snails.
8. Kools are being smoked in the yellow house.
9. Milk is drunk in the middle house.
10. The Norwegian lives in the first house on the left.
11. The Chesterfield smoker lives next to the fox owner.
12. Kools are smoked in the house next to the house where the horse is kept.
13. The Lucky-Strike smoker drinks orange juice.
14. The Japanese smokes Parliament.
15. The Norwegian lives next to the blue house.

The query is: Who drinks water? and who owns the Zebra?

This problem can be represented as a binary constraint network involving 25 variables, one for each of the five houses, five nationalities, five pets, five drinks, and five cigarettes. Each of the variables has domain values {1, 2, 3, 4, 5}, each number

corresponding to a house position (e.g. assigning the value 2 to the variable *horse* means that the horse owner lives in the second house) [Dechter88].

References

Bessière, C. 1991. "*Arc-Consistency in Dynamic Constraint Satisfaction Problems*"; Proceedings 9th National Conference on Artificial Intelligence, Anaheim CA, 221-226

Dechter, R. 1988. "*Constraint Processing Incorporating Backjumping, Learning, and Cutset-Decomposition*"; Proceedings 4th IEEE Conference on AI for Applications, San Diego CA, 312-319

Deville, Y. and Van Hentenryck, P. 1991. "*An Efficient Arc Consistency Algorithm for a Class of CSP Problems*"; Proceedings 12th International Joint Conference on Artificial Intelligence, Sydney, Australia, 325-330

Dincbas, M., Van Hentenryck, P., Simonis, H., Aggoun, A., Graf, T. and Berthier, F. 1988. "*The constraint logic programming language CHIP*"; Proceedings International Conference on Fifth Generation Computer Systems, Tokyo, Japan

Janssen, P., Jégou, P., Nouguier, B., Vilarem, M.C. and Castro, B. 1990. "*SYNTHIA: Assisted Design of Peptide Synthesis Plans*"; New Journal of Chemistry, 14-12, 969-976

Mackworth, A.K. 1977. "*Consistency in Networks of Relations*"; Artificial Intelligence 8, 99-118

Mackworth, A.K. and Freuder, E.C. 1985. "*The Complexity of Some Polynomial Network Consistency Algorithms for Constraint Satisfaction Problems*"; Artificial Intelligence 25, 65-74

Mohr, R. and Henderson, T.C. 1986. "*Arc and Path Consistency Revisited*" ; Artificial Intelligence 28, 225-233

Montanari, U. 1974. "*Networks of Constraints: Fundamental Properties and Applications to Picture Processing*"; Information Science 7, 95-132

Perlin, M. 1992. "*Arc-consistency for factorable relations*"; Artificial Intelligence 53, 329-342

Van Hentenryck, P., Deville, Y. and Teng, C.M. 1992. "*A generic arc-consistency algorithm and its specializations*"; Artificial Intelligence 57, 291-321

Waltz, D.L. 1972. "*Understanding Line Drawings of Scenes with Shadows*"; in: The Psychology of Computer Vision, McGraw Hill, 1975, 19-91 (first published in: Tech.Rep. AI271, MIT MA, 1972)