Arc-Consistency and Arc-Consistency Again

Christian Bessiere

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

[*] This paper is a revised version of [2].

[**] This work was partially supported by the CSPFlex and the BAHIA projects of the PRC-GDR IA (CNRS).

Abstract

There is no need to show the importance of arc-consistency in Constraint Networks. Mohr and Henderson [9] have proposed AC-4, an algorithm with optimal worst-case time complexity. But it has two drawbacks: its space complexity and average time complexity. In problems with many solutions, where constraints are large, these drawbacks become so important that users often replace AC-4 by AC-3 [8], 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. Moreover, the average time complexity of AC-6 is optimal for constraint networks where nothing is known about the constraint semantics.

1. Introduction

Constraint networks provide 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 for solutions. They can improve the efficiency of search procedures, but they are also useful in stand-alone implementations because they can yield solution sets or highly reduced problems [15]. Arc-consistency is the most commonly used consistency technique. Waltz [19] developed arc-consistency for vision problems, it was subsequently studied by Mackworth and Freuder [7] and [8], by Mohr and Henderson [9] who proposed an algorithm with 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 [1] its use was extended to dynamic constraint networks. Recently, Van Hentenryck, Deville and Teng [4] and [15], proposed a generic algorithm which can be implemented with all known techniques, and have extracted classes of networks in which there are algorithms that run arc-consistency in $O(ed)$. In 1992, Perlin [13] gave properties of arc-consistency for factorable relations.

Everyone now looks for arc-consistency complexity in particular classes of constraint networks since AC-4 [9] has optimal worst-case complexity and it is assumed that we cannot do better.

AC-4 drawbacks are its average time complexity, which is too near the worst-case time complexity, and even moreso, its space complexity which is $O(ed^2)$. In problems with many solutions, where arc-consistency removes few values, the AC-4 initialization step is long because whole relations must be considered to construct its data structure. In these cases, AC-3 [8] runs faster than AC-4 despite its non-optimal time complexity [17]. Moreover, in problems with a large number of values in variable domains and with weak constraints, AC-3 is often used instead of AC-4 because of its space complexity.

In this paper we propose a new algorithm, AC-6, which while keeping the $O(ed^2)$ 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: for each value, it checks all 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: for each value, it checks one support per constraint and looks 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 preliminary definitions on constraint networks and arc-consistency. Section 3 presents the algorithm AC-6. In section 4, experimental results show how much AC-6 outperforms algorithms AC-3 and AC-4[1]. Section 5 discusses the drawbacks of this new algorithm. A conclusion is given in Section 6.

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 finite 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. Beginning with Montanari [11], 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 order on the

[1]AC-5 ([4], [15]) is not discussed in this section since it is not an improvement for arbitrary constraints. It is a generic framework in which all previous algorithms can be written and with which we can produce good algorithms for particular constraint classes.

variable domains. 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 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 (=, ≠, <, ≥,…) without giving the matrix of allowed and not allowed pairs of values. When the constraint is very weak, R_{ij} can be defined by the set of forbidden combinations of values for i and j (called the negative form). In fact, R_{ij} can be represented as any function such that $R_{ij}(a, b)$ returns true iff the pair (a, b) is allowed.

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 symmetric directed graph with arcs (i, j) and (j, i) instead of the edge $\{i, j\}$.

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

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}. 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* for this CN if for every variable i in the CN, all values in D_i are viable. When a CN has a domain D, we call *maximal arc-consistent domain* of this CN domain D', defined as the union of all domains included in D and arc-consistent for this CN. D' is also arc-consistent and is the domain expected to be computed by an arc-consistency algorithm.

3. Arc-consistency with unique support

3.1. Preamble

As Mohr and Henderson underlined in [9], 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 is 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 [9] makes this support evident by assigning a *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*) 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 the origin of the 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 average time complexity and a space complexity both 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. If the constraints are given in extension (e.g. matrices of Booleans), this space complexity (bounded by $O(ed^2)$ with the size of the supports sets S_{jb}) is "only" proportional to the size of the problem. But in cases where constraints are defined by arithmetic relations, or given in negative form, the size of the AC-4 data structure may be dramatically larger than the size of the problem.

The purpose of AC-6 is then to avoid expensive checking and storage of all supports for all values. AC-6 keeps the same principle as AC-4, but instead of checking all 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}. Using AC-6 only requires total ordering in all domains D_j. But this is not a restriction since in any implementation 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 ($M(i, a)$=**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 total ordering on these values. We use the following constant time functions and procedures to handle D_i sets, which are considered as lists:

  - $last(D_i)$ returns the greatest value in D_i if $D_i \neq \emptyset$, else returns 0.

  - if $a \in D_i \setminus last(D_i)$, $next(a, D_i)$ returns the smallest value in D_i greater than *a*.

  - $remove(a, D_i)$ removes the value *a* from 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 contains all values supported by (j, b).

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

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

In AC-4, when a value (j, b) is deleted, it is added to the *Waiting-List* to await 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 zero. In AC-6, use of the *Waiting-List* is not changed but the consequence of (j, b) deletion is now to find another support for every (i, a) in S_{jb}. With ordering on D_j, after b (the old support) we look 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 deleted.

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
    if b ≤ last(Dj) then
      begin
      emptysupport ← false ;
      {search of the smallest value greater (or equal) than b that belongs to Dj}
      while not M(j, b) do b ← b + 1 ; [2]
      {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
    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

[2]To improve slightly AC-6, we can replace "$b \leftarrow b + 1$" by "$b \leftarrow$ latest-next(j, b)" where latest-next(j, b) is the value b' that was equal to next(b, D_j) when b was removed from D_j (if b was not last(D_j)). This improvement is significant only with large domains.

constraint R_{ij} to prove that (i, a) is viable. If there is a constraint R_{ij} on which (i, a) has no support, it is removed from D_i and put in the *Waiting-List*.

In the propagation step, values (j, b) are taken from the *Waiting-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 the *Waiting-List* at its turn.

```
{initialization}
Waiting-List ← Empty-List ;
for (i, a) ∈ D do S_ia ← ∅ ; M(i, a) ← true ;
for (i, j) ∈ arcs(G) do
   for a ∈ D_i do
      begin
      b ← 1 ; nextsupport(i, j, a, b, emptysupport ) ;
      if emptysupport
      then remove(a, D_i) ; M(i, a) ← false ; Add-to(Waiting-List, (i, a))
      else Add-to(S_jb, (i, a))
      end

{propagation}
while Waiting-List ≠ ∅ do
   begin
   pick (j, b) from Waiting-List ;
   for (i, a) ∈ S_jb do                {before its deletion (j, b) was the smallest
      begin                                  support in D_j for (i, a) on R_ij}
      delete (i, a) from S_jb ;
      if M(i, a) then
         begin
         c ← b ; nextsupport(i, j, a, c, emptysupport ) ;
         if emptysupport
         then remove(a, D_i) ; M(i, a) ← false ; Add-to(Waiting-List, (i, a))
         else Add-to(S_jc, (i, a))
         end
      end
   end
```

3.3. Correctness of AC-6

Here are the key steps for complete proof of the correctness of AC-6. In this section we denote *DmaxAC* 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 *DmaxAC*, then (i, a) is out of *DmaxAC*. *DmaxAC* is trivially included in D when AC-6 is started. Then, by induction, (i, a) is out of *DmaxAC*. Thus, *DmaxAC/D* is an invariable property of AC-6.

• Every time a value (j, b) is removed, it is put in the *Waiting-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 $D \cup$ *Waiting-List* on each constraint R_{ij}. AC-6 terminates with an empty *Waiting-List*. Hence, after AC-6, every value in D has a support in D on each constraint. Thus, D is arc-consistent.

• *DmaxAC/D* and D arc-consistent at the end of AC-6 implies that D is the maximal arc-consistent domain at the end of AC-6. ❑

3.4. Space and time complexity

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 $O(ed)$ (instead of $O(ed^2)$ for AC-4). Moreover, the counters of AC-4 are not used. In spite of their $O(ed)$ space complexity, they are an expensive part of the AC-4 data structure. Needing a direct access, an implementation with arrays is necessary, taking space for $2ed$ counters, even if some of them will never be used.

In both the initialization step and the propagation step, the inner loop is a call to the procedure *nextsupport* which computes 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. Furthermore, with no information on the constraint semantics, this algorithm is the best we can expect in time. It stops

processing of a value just when it has proof that it is viable (i.e. the first support)[3].

4. Experimental results

After producing an algorithm with just enough processing to ensure that each value is viable, we expect that it will outperform AC-3 and AC-4 on all problems.

We tested the performances of the three algorithms on a large spectrum of problems. For each problem, we counted the number of atomic operations and tests done by each algorithm (we counted one for each *O(1)* set of instructions). This measure is more precise than counting the number of constraint checks since it is directly proportional to running-time (while being independent of the implementation of the structures).

The first comparison was done on the zebra problem ([3] or [14]) which has similarities with some real-life problems. With the representation of this problem given in [3], we obtain the following results:

AC-3: 3692
AC-4: 3802
AC-6: 1995

In Fig. 1, we then compared the three algorithms on a problem often used for algorithm comparisons: the n-queens (i.e. a nxn chessboard on which we want to put n queens, none of them attacked by any other). We can encode it in a CN by representing each column by a variable whose 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

[3]The only possible improvements are variable ordering heuristics [18].

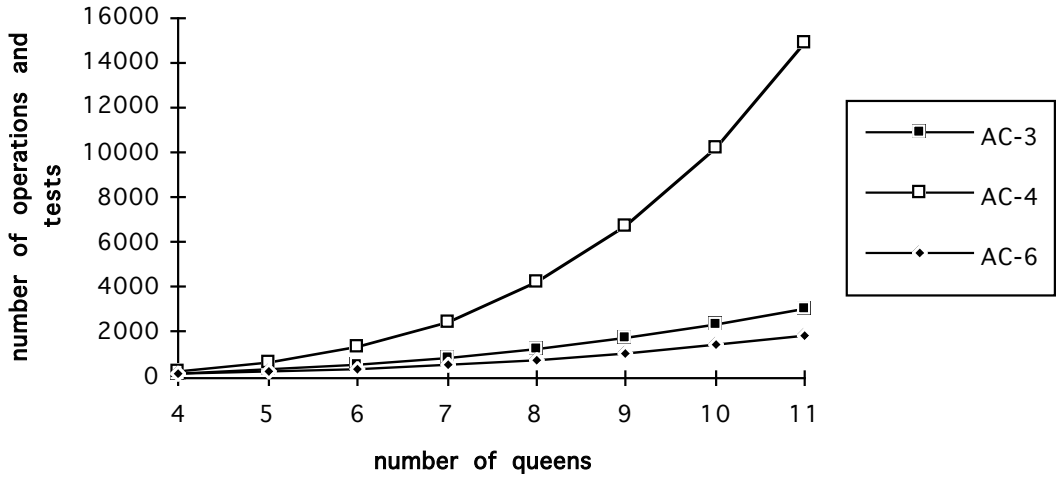

Figure 1. Comparison of AC-3, AC-4 and AC-6 on the *n*-queens problem

allowed positions for the two queens in columns i and j. This CN is very particular since it is extremely symmetrical and all the constraints are weak (note that arc-consistency does not discard any value in this CN). Results obtained here cannot be generalized for other kinds of CNs. However, this CN is interesting to illustrate the behavior of algorithms in particular cases which can locally arise in a part of a CN. In these cases, AC-4 has a bad behavior while AC-3 and AC-6 have an $O(ed)$ average time complexity.

Finally, we defined classes of randomly generated constraint networks and 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 $R_{ij}(a, b)$ is allowed. The result given for each class is the average of ten instances of problems in the class so as to be more representative of the class.

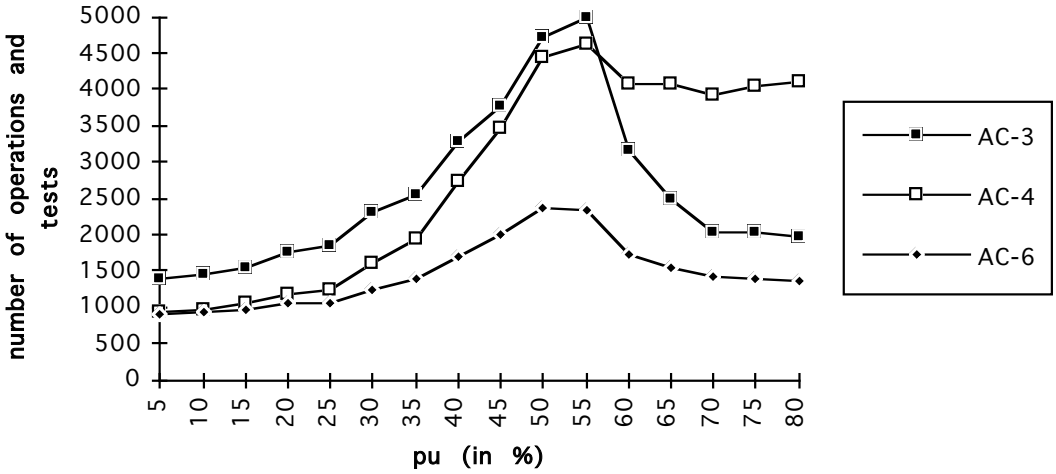

Figure 2. AC-3, AC-4 and AC-6 on randomly generated CNs with 20 variables having 5 possible values, where the probability *pc* to have a constraint between two variables is 30 %

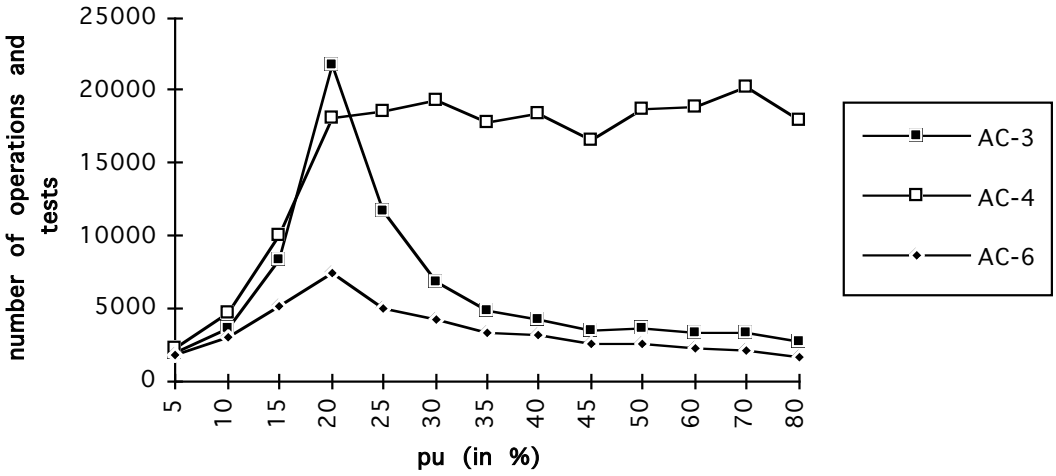

Figure 3. AC-3, AC-4 and AC-6 on randomly generated CNs with 12 variables having 16 possible values, where the probability *pc* to have a constraint between two variables is 50 %

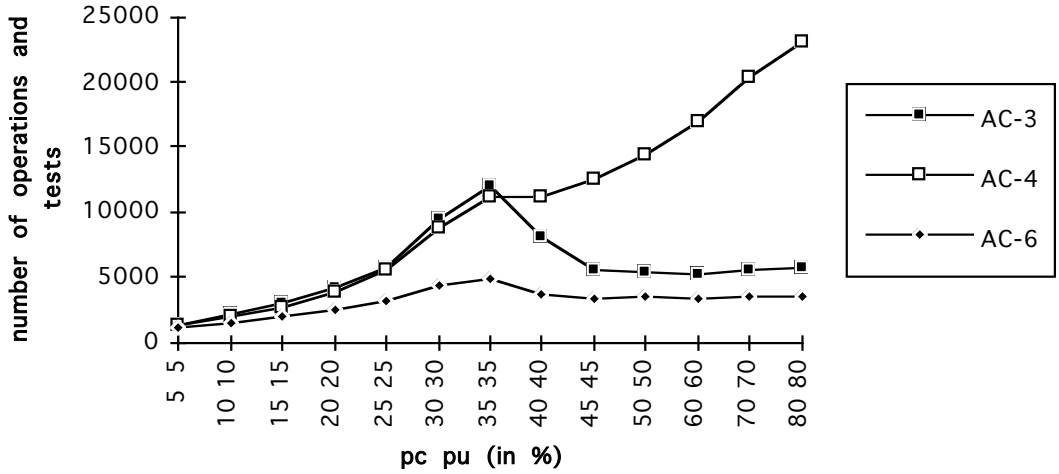

Figure 4. AC-3, AC-4 and AC-6 on randomly generated CNs with 18 variables having 9 possible values

Let's roughly summarize these 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 these figures, AC-4 does not seem interesting. However, 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 [5] 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. When propagation of deletions is long, in "middle" CNs (i.e. not too much constrained and not under-constrained), AC-3 becomes less efficient. However, CNs treated in practice are often "middle" CNs since under-constrained CNs and too-constrained CNs are easy to solve, with solutions or contradictions found quickly.

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

5. Discussion on the limitations of AC-6

In the above sections, we only considered constraint networks with arbitrary constraints. We said that AC-6 is optimal in time on these constraint networks. All experimental results presented in Section 4 use basic versions of the three algorithms AC-3, AC-4, and AC-6. But there are improved versions of AC-4 [10] dealing with arithmetic relations ($<$, $\geq$, $\neq$, $=$, …) in $O(ed)$. The generic algorithm AC-5 can be instanciated to produce efficient algorithms for a number of important classes of constraints: functional, anti-functional, monotonic…[15]. For all of these classes of constraints, the basic version of AC-6 is not optimal in time since the improved versions of AC-4 or AC-5 have a worst-case time complexity of $O(ed)$. However, in many applications, functional constraints, arithmetic relations and other particular constraints are mixed with arbitrary constraints in the constraint network. In these cases, AC-6 provides a simple, general, and efficient algorithm, which deals well with every kind of constraint, avoiding ad hoc implementations or heavy data structures.

However, the main limitation of AC-6 is its theorical complexity when used in a backtracking search procedure. The *really full look-ahead* search procedure [12] is a search procedure which computes arc-consistency at each step of the tree search (i.e. every time a value is chosen for a variable). Any arc-consistency algorithm can be taken to implement this procedure. When AC-4 is used, a heavy initialization phase is made at the root of the search tree, computing all sets of supports and counters for every value. But once this data structure is constructed, we can avoid a new expensive initialization phase at each step of the search, duplicating the data structure of the step before. We can still process in $O(1)$ the direct consequence of the deletion of a support for a value (i.e. decrementing a counter). Using AC-6, the direct consequence of the deletion of a support for a value (i.e. checking the constraint to find the next support) will be processed in $O(d)$. Hence, we think that we can find examples of constraint networks where a *really full look-ahead* procedure written with AC-6 has a worse behavior than one written with AC-4. In practice, on the RESYN application [16] where AC-4 has been replaced by AC-6 in the search procedure, the results are quite favourable to AC-6. But complete experimentation must be done on a large spectrum of problems. It should involve *really full look-ahead* implemented with AC-4 and AC-6, and other look-ahead search procedures like forward-checking [6], which is often considered in the literature as being the best search procedure for constraint networks.

6. 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 as 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. However, its superiority on AC-4 when used in a search procedure requires further experimental proof.

Acknowledgments

I would particularly like to thank Marie-Odile Cordier, without our discussions this paper would not have been written. I also thank Amit Bellicha and Jean-Charles Régin for their useful comments on earlier drafts, and the anonymous AI Journal reviewers for their suggestions.

References

[1] C. Bessière, *Arc-Consistency in Dynamic Constraint Satisfaction Problems*, in: Proceedings 9th National Conference on Artificial Intelligence, Anaheim CA, (1991) 221-226

[2] C. Bessière, M.O. Cordier, *Arc-Consistency and Arc-Consistency Again*, in: Proceedings 11th National Conference on Artificial Intelligence, Washington D.C., (1993) 108-113

[3] R. Dechter, *Enhancement Schemes for Constraint Processing: Backjumping, Learning, and Cutset Decomposition*, Artificial Intelligence 41 (1989/90) 273-312

[4] Y. Deville, P. Van Hentenryck, *An Efficient Arc Consistency Algorithm for a Class of CSP Problems*, in: Proceedings 12th International Joint Conference on Artificial Intelligence, Sydney, Australia, (1991) 325-330

[5] P. Janssen, P. Jégou, B. Nouguier, M.C. Vilarem, B. Castro, *SYNTHIA: Assisted Design of Peptide Synthesis Plans*, New Journal of Chemistry 14-12 (1990) 969-976

[6] R.M. Haralick, G.L. Elliot, *Increasing Tree Seach Efficiency for Constraint Satisfaction Problems*, Artificial Intelligence 14 (1980) 263-313

[7] A.K. Mackworth, *Consistency in Networks of Relations*, Artificial Intelligence 8 (1977) 99-118

[8] A.K. Mackworth, E.C. Freuder, *The Complexity of Some Polynomial Network Consistency Algorithms for Constraint Satisfaction Problems*, Artificial Intelligence 25 (1985) 65-74

[9] R. Mohr, T.C. Henderson, *Arc and Path Consistency Revisited*, Artificial Intelligence 28 (1986) 225-233

[10] R. Mohr, G. Masini, *Running efficiently arc consistency*, in: G. Ferraté et al., eds., Syntactic and Structural Pattern Recognition (Springer-Verlag, Berlin, 1988) 217-231

[11] U. Montanari, *Networks of Constraints: Fundamental Properties and Applications to Picture Processing*, Information Science 7 (1974) 95-132

[12] B.A. Nadel, *Tree Search and Arc Consistency in Constraint Satisfaction Algorithms*, in: L. Kanal and V. Kumar, eds., Search in Artificial Intelligence (Springer-Verlag, Berlin, 1988) 287-342

[13] M. Perlin, *Arc-consistency for factorable relations*, Artificial Intelligence 53 (1992) 329-342

[14] P. Van Hentenryck, *Constraint Satisfaction in Logic Programming* (The MIT Press, Cambridge, MA, 1989)

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

[16] P. Vismara, J.C. Régin, J. Quinqueton, M. Py, C. Laurenço, L. Lapied, *RESYN: Un système d'aide à la conception de plans de synthèse en chimie organique*, in: Proceedings 12th International Conference Avignon'92, EC2 eds., Avignon, France, (1992) 305-318 (in French)

[17] R.J. Wallace, *Why AC-3 is Almost Always Better than AC-4 for Establishing Arc Consistency in CSPs*, in: Proceedings 13th International Joint Conference on Artificial Intelligence, Chambéry, France, (1993) 239-245

[18] R.J. Wallace, E.C. Freuder, *Ordering Heuristics for Arc Consistency Algorithms*, in: Proceedings 9th Canadian Conference on Artificial Intelligence, Vancouver, Canada, (1992) 163-169

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