Maintaining Arc Consistency through Constraint Retraction

B. Neveu, P. Berlandier

SECOIA Project, INRIA–CERMICS, B.P. 93, 06902 Sophia Antipolis, FRANCE

Abstract

To cope with a growing number of applications, the basic formalism of constraint satisfaction problems has to be augmented in various directions. One of these directions is the concept of dynamic constraint problems i.e. problems to which constraints can be added but also retracted at any time. To handle dynamic problems, it is important to adapt efficiently the solving procedures that are available on static ones. Of particular interest is the arc-consistency enforcement procedure; the goal of the paper is to present an adaptation of it to dynamic problems. Contrary to previous approaches, the one presented here does not rely on any reason maintenance system and consequently has the advantage of a low space complexity. The detailed algorithm of the procedure is given and an experimental evaluation of its performances is displayed. Some benefits gained by the approach regarding flexibility and extensibility are highlighted.

1 Introduction

Arc-consistency (AC) procedures are essential preprocessing tools for the resolution of constraint satisfaction problems (CSPs). Their use is well suited to constraint problems that grow monotonically: keeping the arc-consistent state of a problem while adding constraints is simply a matter of restarting AC from the added constraints. AC procedures are thus inherently incremental with respect to constraint addition.

Unfortunately, this is not true for constraint deletion. When a constraint is deleted from a problem, the filtering work that was caused by this constraint cannot be undone easily because no justification for that work was kept. The obvious, brute-force, solution is thus to restart the AC procedure from scratch, with the original domains, on the new problem. The waste of computational efforts of this strategy might not be acceptable for an application that deals with dynamic constraint satisfaction problems (DCSPs), i.e. problems where constraints are frequently added but also retracted.

As for many non-monotonic reasoning systems, one possibility to overcome the problem is to take advantage of past computations. Two previous works [Bessière 91, Prosser 92] have proposed some solutions that add some kind of reason maintenance system (RMS) to the AC procedure to record the reasons for the deletion of a value in a domain.

In this paper, we propose AC|DC (Arc Consistency for Dynamic Constraint problems), a procedure that copes with dynamic constraint problems but does not resort to any reason maintenance mechanism. The prime motivation for this work is to keep the nice simplicity of the CSP representation in the DCSP framework i.e. to leave the representation free from any additional structures used to reflect the past states of a problem. As long as it does not entail some significant performance degradations, we think that simplicity is a good feature that, in particular, favors extensibility. Another motivation is that the process of maintaining justifications for very large CSPs might not be an acceptable solution with respect to memory space.

The paper is organized as follows. We first study the algorithm of the AC|DC procedure and justify some implementation choices we made. Then, with a short experimental evaluation, we compare its efficiency with the DnAC_4 procedure [Bessière 91]. Finally, before concluding, we highlight some of the good properties of our RMS-free approach regarding flexibility, extensibility and also space complexity.

As usual in the constraint literature, we will concentrate in this paper on binary CSPs for the sake of clarity, but the results can easily be extended to CSPs with constraints of higher arity.

2 Presentation of the Procedure

As we have mentioned above, any plain arc-consistency procedure can be used on dynamic constraint problems. However, in the case of constraint retraction, it may be less costly to consider the restoration of the values that have been suppressed by the constraint. This can be accomplished in three steps: the first one proposes a set of restorable values for the variables connected to the deleted constraint. Then, the consequences of these potential additions are propagated throughout the constraint network. Finally, arc consistency is applied starting from the variables which domain has been enlarged, working only on the restorable values to filter out the ones that are inconsistent with respect to the relaxed problem.

2.1 Algorithm

Let $\mathcal{P}$ be the set of binary constraints that makes up our constraint problem. A binary constraint is a pair of variables $\{i, j\}$ associated to a relation R_{ij} which defines the legal couples of values for i and j.

We will assume that $R_{ij} = {R_{ji}}^{-1}$ for all i and j meaning that, if a couple (a, b) is in R_{ij} then (b, a) is in R_{ji}. We will say that a value a for i is a *support* for (or *supports*) a value b for j iff $(a, b) \in R_{ij}$.

Let $\mathcal{V}$ be the set of variables constrained by $\mathcal{P}$. For every variable i in $\mathcal{V}$, we suppose that we have access to its initial domain, noted D_i^0, and its current domain, noted D_i. The former is the set of possible values as defined by the user. The latter is a subset of the initial domain and is assumed to be arc-consistent with respect to $\mathcal{P}$ (which means that every value in D_i has at least one support in the current domain of each variable connected to i in $\mathcal{P}$).

AC|DC is composed of two sub-procedures. The first one is the classic AC_3 procedure [Mackworth 77] to handle constraint addition. The second one, which is of interest here, is the *retract-constraint* procedure (presented on figure 1) to handle constraint retraction. It takes in argument a constraint $\{k, m\}$ to be retracted from $\mathcal{P}$. After its execution, the current domains of the variables will have been made arc-consistent with respect to the problem $\mathcal{P} \setminus \{\{k, m\}\}$. Retraction occurs in three steps, described below.

1. Propose the initial restorations. The starting point of the value restoration process is the pair of variables $\{k, m\}$ connected to the constraint that has to be deleted. For each of these two variables, we want to approximate (overestimate by a superset) the set of values that have been ruled out by the influence of the deleted constraint alone.

For variable k (resp. m), a rough (but inexpensive) guess for this set is the difference between its initial and its current domain. However, a better guess can be achieved by only retaining the values of this difference that do not have any support in the current domain of m (resp. k). The resulting set of values is stored in the *propagable* table. It will serve as the basis for the forthcoming propagation step.

This preliminary work is achieved by the *propose* procedure. Its benefit is particularly evident in some pathological cases where the retraction of a constraint does not entail any value restoration (this is often the case for "easy" constraints such as inequality).

```
retract-constraint({k, m}):
initialize propagable to ∅;
delete {k, m} from P;
propose(k, m, propagable);
propose(m, k, propagable);
initialize restorable to ∅;
propagate({k, m}, propagable, restorable);
filter(restorable);
for each i in V do
    add restorable[i] to D_i;
end.

propose(i, j, var propagable):
for each a in D_i^0 \ D_i do
    initialize supported to false;
    for each b in D_j do
        if (a, b) ∈ R_ij then
           set supported to true;
           break;
    if supported = false then
       add a to propagable[i];
end.

propagate({k, m}, var propagable, var restorable):
set L to {k, m};
while L ≠ ∅ do
  choose and delete i from L;
  for each j such that {i, j} ∈ P do
      set S to ∅;
      for each b in
          D_j^0 \ (D_j ∪ restorable[j] ∪ propagable[j]) do
          for each a in propagable[i] do
              if (a, b) ∈ R_ij then
                 add b to S;
                 break;
      if S ≠ ∅ then
         add j to L;
         append S to propagable[j];
  append propagable[i] to restorable[i];
  set propagable[i] to ∅;
end.
```

Figure 1: the *retract-constraint* procedure

If empty propagable sets are detected right from the start, the two subsequent steps of costly propagations are avoided.

2. Propagate the restorations. The role of the *propagate* procedure is to compute, for every variable of the problem, a superset of the values that will eventually be restored. As its name reveals, it works by propagating throughout the problem the consequences of the addition of the propagable values.

The process is quite simple: while the *propagable* table holds some non-empty sets of values, these sets are propagated. Propagating a set of values S_i for a given variable i means finding, for each variable j connected to i, the set of values S_j in the initial domain D_j^0 that are newcomers to j (i.e. that are neither in its current domain nor already stored as restorable or propagable values) and that have at least one support in S_i. Once determined, S_j is added to the propagable values of j. Finally, when the values in S_i have been propagated, they are stored as restorable values so that they will never be considered again for propagation. This ensures that the propagation process will terminate.

3. Filter out the inconsistent restorations. The role of the *filter* procedure is to eliminate from the *restorable* table the values that are not arc-consistent with respect to the relaxed problem. This procedure can thus be any classical AC procedure provided that only the restorable values are filtered and that the supporting values are sought either in the current domains or in the restorable values.

To conclude the retraction process, the restorable values that have been found arc-consistent are added to the current domains of the variables (lines 7 and 8 of *retract-constraint*). So eventually, all the restorable values are restored (ensured by step 1 and 2) and none of the restored values are inconsistent (ensured by step 3).

2.2 Alternative Propagation Strategy

We devised a more cautious version of the propagation process (step 2) which checks that every propagable value a supports at least one value for all its connected variables. When it is not the case, a is not stored as a restorable but as an *excluded* value and is never considered again. This additional test avoids the restoration of inconsistent values and their ill-fated propagation. But on the down side, the fact that we have to look for a support in all the possible values of a variable makes the process costly and possibly redundant with the subsequent filtering step.

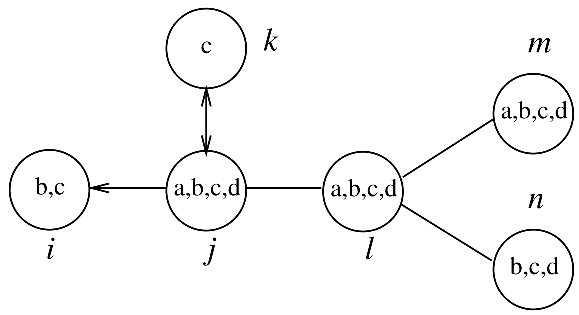
An experimental comparison of the two versions revealed that the straightforward propagation strategy is usually more efficient than the cautious one.

It is interesting to mention the average repartition of the total number of constraint evaluations among the three steps:

- for the straightforward strategy, 8% of the evaluations are spent in step 1, 12% in step 2 and 80% in step 3,
- for the cautious strategy, 6% are spent in step 1, 54% in step 2 and 40% in step 3.

2.3 Example

We work here a small example of constraint retraction to illustrate the role of the different steps. The following figure shows a constraint problem with the initial domains of the variables. The constraint are $i < j$, $j \neq k$, $j = l$, $l = m$ and $l = n$.



In the table below, the D-line shows the domains after arc-consistency was applied. Now, if we decide to delete constraint $\{i, j\}$, the following actions will take place: Step 1 will look for propagable values for i and j. The current domain of i is the same as its initial domain, so there is no value to propagate here. For variable j, we will find that a and b are propagable but c is not as it is currently supported by b in D_i. Then, step 2 will blindly propagate a and b from j to l and m and b to n. Finaly, step 3 will filter out value a for l and m, leaving only b as a restored value.

	i	j	k	l	m	n
D	$\{b, c\}$	$\{d\}$	$\{c\}$	$\{d\}$	$\{d\}$	$\{d\}$
propagable	-	$\{a, b\}$	-	-	-	-
restorable	-	$\{a, b\}$	-	$\{a, b\}$	$\{a, b\}$	$\{b\}$
restored	-	$\{b\}$	-	$\{b\}$	$\{b\}$	$\{b\}$

2.4 Complexity

The worst case complexity of the *propose* procedure is trivially $O(d^2)$ where d is the size of the domains. The maximal cost of lines 3 and 4 of *retract-constraint* is thus in $O(d^2)$.

In the *propagate* procedure, the maximal number of values that can be made restorable for each variable is d. For each of these values, the propagation loop considers all the constraints connected to the variable and for each constraint, at most d constraint checks are performed to find a newly supported value. The maximal number of constraint checks for all the variables is then $2ed^2$ where e is the number of constraints of the problem (the sum of the number of constraints on all the variables is $2e$). The maximal cost of line 6 of *retract-constraint* is thus in $O(ed^2)$.

For the *filter* procedure, let us suppose that d_a is the maximal number of restorable values for a variable. Filtering one arc costs at most $d_a d$ constraint checks and each arc is enqueued at most d_a times for each variable. The maximal number of constraint checks is then edd_a^2. An upper bound for d_a is d and then the maximal cost of line 7 of *retract-constraint* is in $O(ed^3)$.

The total complexity of the *retract-constraint* procedure is then $O(d^2+ed^2+ed^3) = O(ed^3)$ which makes it theoretically comparable to the non-dynamic AC3 procedure. Fortunately, in the average case, the sets of *propagable* and *restorable* values have a size much smaller that d, preserving a fair efficiency, at least w.r.t. the use of non-dynamic AC3.

The space complexity of AC|DC is discussed in section 4.1, where it is compared to the one of RMS-based procedures.

3 Experimental Evaluation

In order to figure out how acceptable the performances of AC|DC are, we have implemented DnAC4 as a reference. To produce a fair comparison, the two algorithms have been implemented in the same language, using the same types of data structures. Also, the benchmarks have been run on the same problem implementations and last, the execution times have been measured on the same machine.

The basis for our tests is the well known "Zebra problem" [Smith 92]. This problem is composed of 25 variables and our implementation shows 62 constraints of different types (equalities with and without offset, inequalities and disjunctive equalities).

3.1 Retraction from a Consistent State

Our first set of tests consists in the retraction of some constraints from the arc-consistent state of the Zebra problem.

1. Enforcing arc-consistency. The determination of the arc-consistent state of the problem costs an average (on 10 different runs) of 9550 ms using AC|DC. With DnAC4, it costs an average of 18300 ms, which is almost twice more. This is not surprising and agrees with the results exposed in [Wallace 93], showing that AC3 performs almost always better than AC4 [Mohr 86]

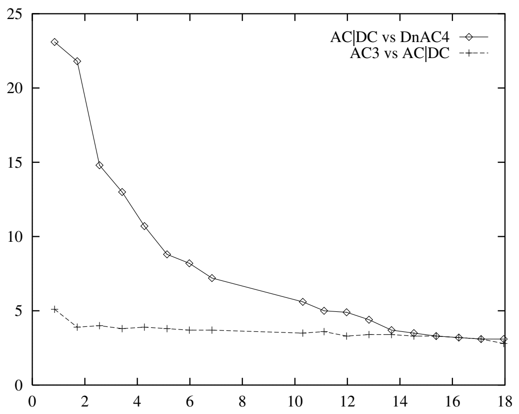


Figure 2: retraction from the consistant state

(from which DnAC4 is derived) when used as a preprocessing tool.

2. Retracting constraints. From the consistent state of the problem, we retract some constraints that are responsible for a certain number of value deletions with three methods: AC|DC, DnAC4 and the good old AC3. Arc-consistency rules out a total of 31 values on the whole problem. For different percentages of restored values (w.r.t. the total number of values of the problem), figure 2 shows the ratios of the execution times of the different methods.

For only one restored value, we can see that the ratio between AC|DC and DnAC4 is highly in favor of the latter. The work of DnAC4 is almost instantaneous. Then, as the number of restored values augments, the ratio lowers. Execution time for AC|DC does not increase very much (from 2100 ms for 1% to 2350 ms for 18%) while the time for DnAC4 increases constantly (from 90 ms to 750 ms). The cost of DnAC4 is directly proportional to the workload when the cost of AC|DC is rather a fix toll for any workload plus a small proportional fee.

Figure 2 also shows that the use of plain AC3 is 5 times more expensive than AC|DC for a small number of restored values. This overhead decreases with the augmentation of the number of values as it corresponds for AC3 to a diminution of the number of values to eliminate. Still, AC|DC performs 2.8 times better than AC3 when restoring 18% of the values.

3.2 Retraction from an Inconsistent State

For our second set of tests, we have added one constraint to the Zebra problem so that the resulting problem is inconsistent.

1. Enforcing arc-consistency. To wipe out the domains, AC|DC takes an average of 7300 ms (the minimum being 3900 ms and the maximum 9200 ms) while

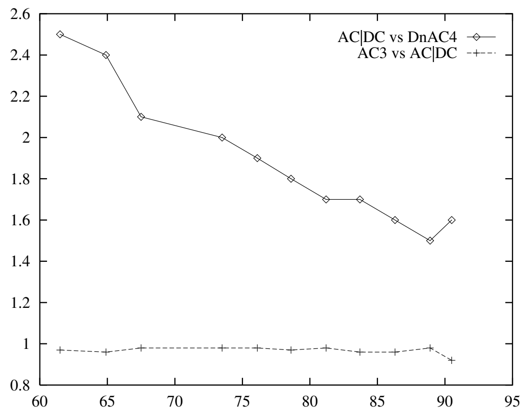


Figure 3: retraction from an inconsistant state

DnAC$_4$ takes an average of 20300 ms. Here, AC|DC has got an important edge on DnAC$_4$ because the propagation process of the former stops as soon as the first empty domain is detected.

2. Retracting constraints. A total of 117 values are deleted when domains are wiped-out. We report on figure 3 the ratios of the execution times for different percentages of these values. The results here are the prolongation of the ones displayed on figure 2: when a big bunch of values are restored, the edge of DnAC$_4$ over acdc reduces (up to a ratio of less than 1.6). Also, it is important to note that AC|DC becomes less efficient than AC$_3$. This is easily understandable as AC|DC has to first restore almost all the values before filtering them. This restoration process, which is costly here, is avoided by AC$_3$.

4 Final Remarks

Before we conclude, we present some benefits gained by our RMS-free approach regarding space complexity, flexibility and extensibility.

4.1 Space Complexity

We know of two previous approaches to dynamic AC. In [Prosser 92], Prosser adds a simple justification-based truth maintenance system to the AC$_3$ procedure. Each time a constraint C_{ij} is used to filter the domain of j from the domain of i, a justification for the new domain of j is recorded. This justification takes the form of a triple $\langle d_i, C_{ij}, d_j \rangle$ with the previous justifications for d_i and d_j as antecedents. Recording this justification graph induces a worst case space complexity of $O(ed^2)$.

In [Bessière 91] and [Bessière 92], Bessière adds to the AC$_4$ procedure a data structure that justifies each value suppression by the first constraint which is responsible for that suppression. This structure does not change the (already high) upper bound space complexity of AC$_4$ which is $O(ed^2)$. In these two approaches, our propagation of restorable values in the constraint graph is replaced by a propagation step in the justification graph.

Regarding space complexity, the *retract-constraint* procedure uses two tables of domains indexed by the variables of the problem thus having a maximal size of nd. The *filter* procedure uses a queue of at most e elements. This gives an upper bound space complexity of $O(e+nd)$ for AC|DC which compares very favorably to the above mentioned approaches.

4.2 Flexibility

In DnAC$_4$, the deleted values are justified by the constraint that caused their deletion. Therefore, the least relaxation step that we can take is to retract one whole constraint. AC|DC offers a much finer relaxation grain. Indeed, the *propagable* table (which defines the values that must be propagated and might be restored) can be initialized from a set of tuples or directly with a set of values.

For example, if we decide to weaken the problem by accepting a set of extra tuples T_{km} for the constraint between k and m, we just have to initialize *propagable* with the projection of these tuples on k and m. To do so, lines 1 to 4 of the *retract-constraint* procedure can be replaced by the following ones:

add-tuples($\{k,m\}$, T_{km}):
1 initialize *propagable* to $\emptyset$;
2 append T_{km} to R_{km} and T_{km}^{-1} to R_{mk};
3 set *propagable*[k] to $\{a \mid (a,b) \in T_{km}\}$;
4 set *propagable*[m] to $\{b \mid (a,b) \in T_{km}\}$; ...

Now, if we decide to weaken the problem by directly adding a set of extra values S_k to the initial domain of a variable k, we have to initialize *propagable* with this set. In this case, lines 1 to 4 of *retract-constraint* can be replaced by:

add-values(k, S_k):
1 initialize *propagable* to $\emptyset$;
2 append S_k to D_k^0;
3 set *propagable*[k] to S_k; ...

4.3 Extensibility

In this paper, we have presented a version of AC|DC that relies on AC$_3$ for both constraint addition and the third step of constraint retraction. However, for constraint addition, any AC procedure can be used instead, as long as it is not based on some global data structures. For constraint retraction, any efficient AC procedure can be used for the third step (e.g. AC_5 [Van Hentenryck 92] or AC_6 [Bessière 93]) and thus improve the performances of AC|DC.

Finally, it is worth stressing the fact that AC|DC does not introduce any complex data structure that might turn out to be cumbersome when adapting the consistency procedure to new CSP frameworks (e.g. constraint problems with preferences, uncertainty or hierarchical domains).

5 Conclusion

In this paper, we have explored the use of a RMS-free procedure to maintain arc-consistency through constraint retraction. We have weighted this solution against two other possibilities: the use of a RMS-based procedure (e.g. DnAC_4) and the use of a plain AC procedure (e.g. AC_3). This experiment has shown the following points:

- AC|DC is definitely more flexible and less greedy in memory than RMS-based procedures. The latter point makes it a good candidate to handle large problems.
- Compared to DnAC_4, we found that the efficiency of AC|DC is poor for a small proportion of restorations but gets better as this proportion is growing.
- On the contrary, compared to AC_3, we found that the efficiency of AC|DC is poor for a high proportion of restorations but gets better as this proportion is decreasing.

In our sets of tests, AC|DC performed at most 5 times better than AC_3. We can ask whether this factor is attractive enough provided the very good performances of AC_5-like AC procedures on real constraint problems (e.g. numeric constraints on integer range domains) ? On the other hand, we claim that we can not loose much by always using AC|DC as the worst ratio to AC_3 was close to 1 for a very high proportion (90%) of restored values.

References

[Bessière 91] C. Bessière. Arc–consistency in dynamic constraint satisfaction problems. In *Proc. AAAI*, 1991.

[Bessière 92] C. Bessière. Arc-consistency for non-binary dynamic CSPs. In *Proc. ECAI*, 1992.

[Bessière 93] C. Bessière, M-O. Cordier. Arc–consistency and arc–consistency again. In *Proc. AAAI*, 1993.

[Mackworth 77] A. Mackworth. Consistency in networks of relations. *Artificial Intelligence*, 8:99–118, 1977.

[Mohr 86] R. Mohr, T. Henderson. Arc and path consistency revisited. *Artificial Intelligence*, 28:225–233, 1986.

[Prosser 92] P. Prosser, C. Conway, C. Muller. A distributed constraint maintenance system. In *Proc. Les Systèmes Experts et leurs Applications*, Avignon, France, 1992.

[Smith 92] B. Smith. How to solve the zebra problem, or path consistency the easy way. In *Proc. ECAI*. J. Wiley & sons, 1992.

[Van Hentenryck 92] P. Van Hentenryck, Y. Deville, C-M. Teng. A generic arc-consistency algorithm and its specializations. *Artificial Intelligence*, 57:291–321, 1992.

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