Arc-Consistency in Dynamic Constraint Satisfaction Problems

Christian Bessière

Laboratoire d'Informatique, de Robotique et de Microélectronique de Montpellier
860, rue de Saint Priest
34090 Montpellier FRANCE
Email: bessiere@crim.crim.fr

Abstract

Constraint satisfaction problems (CSPs) provide a model often used in Artificial Intelligence. Since the problem of the existence of a solution in a CSP is an NP-complete task, many filtering techniques have been developed for CSPs. The most used filtering techniques are those achieving arc-consistency. Nevertheless, many reasoning problems in AI need to be expressed in a dynamic environment and almost all the techniques already developed to solve CSPs deal only with static CSPs. So, in this paper, we first define what we call a dynamic CSP, and then, give an algorithm achieving arc-consistency in a dynamic CSP. The performances of the algorithm proposed here and of the best algorithm achieving arc-consistency in static CSPs are compared on randomly generated dynamic CSPs. The results show there is an advantage to use our specific algorithm for dynamic CSPs in almost all the cases tested.

1. Introduction

Constraint satisfaction problems (CSPs) provide a simple and good framework to encode systems of constraints and are widely used for expressing static problems. Nevertheless, many problems in Artificial Intelligence involve reasoning in dynamic environments. To give only one example, in a design process, the designer may add constraints to specify further the problem, or relax constraints when there are no more solutions (see the system to design peptide synthesis plans: SYNTHIA [Janssen et al 1989]). In those cases we need to check if there still exist solutions in the CSP every time a constraint has been added or removed.

Proving the existence of solutions or finding a solution in a CSP are NP-complete tasks. So a filtering step is often applied to CSPs before searching solutions. The most used filtering algorithms are those achieving arc-consistency. All arc-consistency algorithms are written for static CSPs. So, if we add or retract constraints in a CSP and achieve arc-consistency after each modification with one of these algorithms, we will probably do many times almost the same work.

So, in this paper we define a Dynamic CSP (DCSP) ([Dechter & Dechter 1988], [Janssen et al 1989]) as a sequence of static CSPs each resulting from the addition or retraction of a constraint in the preceding one. We propose an algorithm to maintain arc-consistency in DCSPs which outperforms those written for static CSPs.

The paper is organized as follows. Section 2 presents the CSP model (2.1) and defines what we call a Dynamic CSP (2.2). Arc-consistency filtering method is introduced and the best algorithm achieving it (AC-4 in [Mohr & Henderson 1986]) is described (2.3). Why this algorithm is not optimal in DCSPs is underlined in 2.4. Section 3 presents a new method which adapts the idea of AC-4 to reach better performances on DCSPs. In section 4, a comparison of the performances of the two algorithms on randomly generated DCSPs is given. Section 5 contains a summary and some final remarks.

2. Definitions and preliminaries

2.1. Constraint Satisfaction Problems

A **static** constraint satisfaction problem (CSP) (X, dom, c, R) involves a set of n **variables**, $X = \{i, j,...\}$, each taking **value** in its respective **domain**, $dom(i)$, $dom(j)$,..., elements of dom, and a set of **constraints** c. Each constraint C_p in c involves a subset $\{i_1,..., i_q\}$ of X and is labeled by a relation R_p of R, subset of the Cartesian product $dom(i_1)$ x...x $dom(i_q)$, that specifies which values of the variables are compatible with each other. A **binary** constraint satisfaction problem is one in which all the constraints are binary, i.e., involve two variables. A binary CSP can be associated with a constraint-graph in which

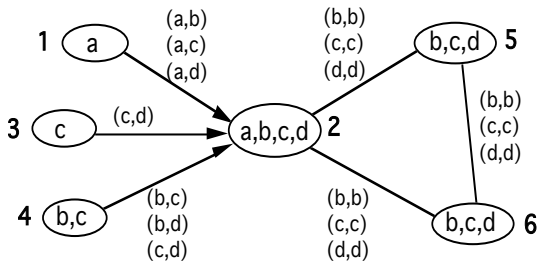

Figure 1: An example of CSP

nodes represent variables and edges connect those pairs of variables for which constraints are given. In that case, the relation associated with the edge $\{i, j\}$ can be denoted R_{ij}. Consider, for instance, the CSP presented in figure 1 (modified from [Mackworth 1977]). Each node represents a variable whose values are explicitly indicated, and each edge is labeled with the set of value-pairs permitted by the

constraint between the variables it connects (non-oriented edges are equality constraints and oriented ones are a strict lexicographic order along the arrows).

A **solution** of a CSP is an assignment of values to all the variables such that all the constraints are satisfied. The task, in a CSP, is to find one, or all the solutions.

We now only consider binary CSPs for clarity, but the results presented here can easily be applied to general CSPs [Bessière 1991].

2.2. Dynamic Constraint Satisfaction Problems

A **dynamic** constraint satisfaction problem (DCSP) p is a sequence of static CSPs $p_{(0)},\ldots, p_{(\alpha)}, p_{(\alpha+1)},\ldots$, each resulting from a change in the preceding one imposed by "the outside world". This change can be a **restriction** (a new constraint is imposed on a pair of variables) or a **relaxation** (a constraint that was present in the CSP is removed because it is no longer interesting or because the current CSP has no solution).

So, if we have $p_{(\alpha)}=(X, dom, c_{(\alpha)}, R)$, we will have $p_{(\alpha+1)}=(X, dom, c_{(\alpha+1)}, R)$ where $c_{(\alpha+1)}=c_{(\alpha)} \pm C$, C being a constraint. $p_{(0)} =(X, dom, \varnothing, R)$.

2.3. Arc-consistency

The task of finding solutions in a CSP has been treated by several authors, and since the problem is NP-complete, some of them have suggested that a preprocessing or filtering step be applied before the search (or backtracking) procedures. Then, consistency algorithms were proposed ([Montanari 1974], [Mackworth 1977], [Freuder 1978], [Dechter & Pearl 1988]). These algorithms do not solve a CSP completely but they eliminate once and for all local inconsistencies that cannot participate in any solutions. These inconsistencies would otherwise have been repeatedly discovered by most backtracking procedures.

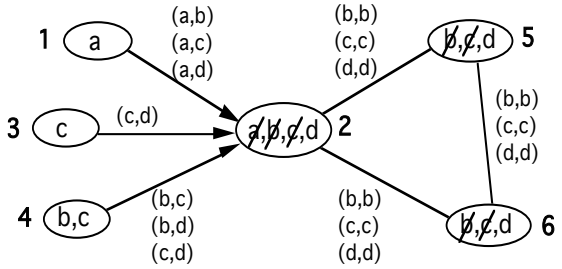

Figure 2: The CSP of fig.1 after application of an arc-consistency algorithm

A k-consistency algorithm removes all inconsistencies involving all subsets of size k of the *n* variables [Freuder 1978]. In fact, the most widely used consistency algorithms are those achieving 2-consistency (or arc-consistency). Arc-consistency checks the consistency of values for each couple of nodes linked by a constraint and removes the values that cannot satisfy this local condition (see figure 2). It is very simple to implement and has a good efficiency. The upper bound time complexity of the best algorithm achieving arc-consistency (AC-4 in [Mohr & Henderson 1986]) is $O(ed^2)$ with e the number of constraints and d the maximal number of values in the domain of a variable .

Arc-consistency can be seen as based on the notion of support. A value a for the variable i is viable if there exists at least one value that "supports" it at each variable j. The Mohr's and Henderson's algorithm, AC-4, makes this support evident by assigning a counter to each arc-value pair. Such pairs are denoted $[(i, j), a]$ and indicate the arc from i to j with value a at node i. The edge $\{i, j\}$ between i and j may be replaced by the two directed arcs (i, j) and (j, i) as they are treated separately by the algorithm (but we still have $R_{ij} = R_{ji}^{-1}$). The counters are designed by $counter[(i, j), a]$ and indicate the number of j values that support the value a for i in the constraint $\{i, j\}$. In addition, for each value b at node j, the set S_{jb} is constructed where $S_{jb} =\{(i, a) / b$ at node j supports a at node $i\}$, that is, if b is eliminated at node j, then counters at $[(i, j), a]$ must be decremented for each (i, a) supported by (j, b). This algorithm uses too, a table, M, to keep track of which values have been deleted from which nodes, and a list, $List$, to control the propagation of deletions along the constraints. $List$ is initialized with all values (i, a) having at least one counter equal to zero. These values are removed from M. During the propagation phase, the algorithm takes values (j, b) in $List$, removes one at each counter $counter[(i, j), a]$ for all (i, a) in S_{jb}, and when a $counter[(i, j), a]$ becomes equal to zero, it deletes (i, a) from M and puts it in $List$. The algorithm stops when $List$ is empty. That means all values in M have non empty supports on all the constraints. So, the CSP is arc-consistent. And it is the maximal arc-consistent domain.

2.4. Arc-consistency in DCSPs

Mohr's and Henderson's algorithm, AC-4, can be used in DCSPs. It keeps all its goods properties when we do a restriction, starting filtering from the current arc-consistent domain and pruning a new value when one of its counters has become zero (i.e. the value has no support on a constraint) after addition of constraints. But, when we remove a constraint (making a relaxation), AC-4 cannot find which value must be put back and which one must not: as it has "forgotten" the reason why a value has been removed, it cannot make the opposite propagation it has done during restrictions. So, we have to start filtering from the initial domain.

3. A new method

3.1. Introduction

As we have seen above, AC-4 does not have good properties (incrementality) for processing relaxations. So, in this section, we propose DnAC-4, a new arc-consistency algorithm for DCSPs. In DnAC-4 we extend AC-4 by recording some informations during restrictions while

keeping its good properties. Then, DnAC-4 remains incremental for relaxations.

More precisely, during a restriction, for every value deleted, we keep track of the constraint origin of the deletion as the "justification" of the value deleted. The justification is the first constraint on which the value is without support. During a relaxation, with the help of justifications we can incrementaly add to the current domain values that belong to the new maximal arc-consistent domain. But we need to be careful because after the relaxation, the system must be in the same state as if the algorithm had started with the initial CSP $p_{(0)}$, and had done only restrictions with all the new set of constraints: the new domain must be the maximal arc-consistent domain and the set of justifications of removed values must remain **well-founded**. Well-founded means that every value removed is justified by a non-cyclic chain of justifications (see figure 2: (2, c) deletion justified by the constraint {2, 6}, (6, c) by {6, 5} and (5, c) by {5, 2} would not be a well-founded set of justifications).

This process of storing a justification for every value deleted is based on the same idea as the system of justifications of deductions in truth maintenance systems (TMSs) [Doyle 1979], [McAllester 1980].

3.2. The algorithm DnAC-4

The algorithm we propose works with nearly the same data structures than AC-4. Each arc-value pair $[(i, j), a]$ has a counter of the number of supports, denoted $counter[(i, j), a]$. A table D of booleans keeps track of which values are in the current domain or not. The first difference is that a set of supported values S_{jib} is constructed for each arc-value pair $[(j, i), b]$: S_{jib} ={a / b at node j supports a at node i} (we have S_{jb} (of AC-4) equal to $\approx S_{jib}$ for $\{j, i\} \in$ c). So, when a constraint $\{i, j\}$ is retracted, we delete S_{ija} and S_{jib} for all arc-value pairs $[(i, j), a]$ and $[(j, i), b]$ instead of removing values (i, a) in S_{jb} and values (j, b) in S_{ia}. In the data structure we added a table *justif* to record the justifications of the values deleted: $justif(i, a)=j$ iff (i, a) has been removed from D because $counter[(i, j), a]$ was equal to zero (i.e. $\{i, j\}$ is the origin of (i, a) deletion). Then, for all (i, a) in D, $justif(i, a)$=nil. The lists SL and RL respectively control the propagation of deletions and additions of values along the constraints.

When the algorithm starts with $p_{(0)}$, the tables are initialized:

```
for each (i, a) ∈ dom do
   begin D(i, a) , true; justif(i, a) , nil end;
```

Adding a constraint $\{i, j\}$ is done by calling:

```
procedure Add ({i, j});
   begin
      Put {i, j} in the set of constraints c;
      SL , ∅;
      Beg-Add ((i, j), SL); Beg-Add ((j, i), SL);
      Propag-Suppress (SL);
   end;
```

• the procedure Beg-Add (see fig.3) builds $counter[(i, j), a]$, $counter[(j, i), b]$, S_{ija}, S_{jib}, for each values (i, a) and (j, b). It puts in the suppression list SL arc-value pairs without support on $\{i, j\}$ (i.e. with counter equal to zero).

```
procedure Beg-Add ((i, j ) ; var SL );
   begin
1  for each b ∈ dom(j ) do S_jib , ∅;
2  for each a ∈ dom(i )do
      begin
3     Total , 0;
4     for each b ∈ dom(j ) do
5        if ((i, a ), (j, b )) ∈ R_ij then
            begin
6           if D(j, b )=true then Total , Total + 1;
7           Append(S_jib , (a ));
            end;
8     counter[(i, j ), a ] , Total  ;
9     if counter[(i, j ), a ]=0 then Append(SL, [(i, j ), a ]);
      end;
   end;
```

Figure 3

• in the procedure Propag-Suppress (see fig.4), values without support (in SL) are deleted and the consequences of these deletions are recursively propagated. For all value removed from D, Propag-Suppress updates the table *justif*.

```
procedure Propag-Suppress (var SL );
   begin
1  while SL ≠ ∅ do
      begin
2     choose [(i, m ), a ] from SL and remove it from SL;
3     if D(i, a )=true and counter[(i, m ), a ]=0 then
         begin
4        justif(i, a ) , m ;
5        D(i, a ) , false;
6        for each j / {i, j } ∈ c do
7           for each b ∈ S_ija do
               begin
8              counter[(j, i ), b ] , counter[(j, i ), b ] - 1;
9              if counter[(j, i ), b ]=0 then
                  Append(SL, [(j, i ), b ] ) ;
               end;
         end;
      end
   end;
```

Figure 4

Removing a constraint $\{k, m\}$ is done by calling:

```
procedure Relax ({k, m } );
   begin
      SL , ∅ ;
      Init-Propag-Relax ({k, m }, SL );
      Propag-Suppress (SL );
   end;
```

The well-foundness property must be kept after the relaxation of a constraint. So, there are two parts in the relaxation process:

• part1: the procedure Init-Propag-Relax (see fig.5) in step 1 puts in the relaxation list RL values (k, a) and (m, b) for which removing was directly due to $\{k, m\}$ (i.e. $justif(k, a)=m$ or $justif(m, b)=k$), and deletes counters and sets of supported values for all arc-value pairs $[(k, m), a]$ and $[(m, k), b]$. In step 2 it adds in D values in RL and these adding of values are recursively propagated to each value that has a support restored on the constraint marked as its justification. Init-Propag-Relax finishes when every value with a support on the constraint marked as its justification of deletion is added to D. During this phase of putting back values, when an added value (i, a) is still without support on a constraint $\{i, j\}$ (i.e. $counter[(i, j), a]=0$), Init-Propag-Relax puts in SL the arc-value pair $[(i, j), a]$.

```
procedure Init-Propag-Relax ({k, m }; var SL );
   begin
   { Step 1: values whose justification was {k, m } are
      put in RL }
1  RL , ∅ ;
2  for each a ∈ dom(k ) do
      if justif(k, a )=m then
         begin
         Append(RL, (k, a )); justif(k, a ) , nil ;
         end;
3  for each b ∈ dom(m ) do
      if justif(m, b )=k then
         begin
         Append(RL, (m, b )); justif(m, b ) , nil ;
         end;
4  Delete {k, m } from the set of constraints c and remove
      its counters and sets of supported values;

   { Step 2: values in RL are added to D and
      consequences are propagated }
5  while RL ≠ ∅ do
      begin
6     choose (i, a ) from RL and remove it from RL;
7     D(i, a ), true ;
8     for each j / {i, j } ∈ c do
         begin
9        for each b ∈ S_ija do
            begin
10          counter[(j, i ), b ] , counter[(j, i ), b ] + 1;
11          if justif(j, b )=i then
               begin
               Append(RL, (j, b ));  justif(j, b ) , nil ;
               end;
            end;
12       if counter[(i, j ), a ]=0 then
            Append(SL, [(i, j ), a ] );
         end;
      end;
   end;
```

Figure 5

• part2: Propag-Suppress retracts again values in SL, marking as the new justification the constraint on which

the value is still without support (or one of the constraints if there are more than one). These suppressions are propagated: the classic arc-consistency process restarts.

We develop here DnAC-4 on the DCSP of figures 1, 2 and 6, to show the mechanism of justifications:

Changes:	Consequences:	Justifications stored:
Add {1, 2}	deletion of (2, a)	justification(2, a)={1, 2}
Add {2, 3}	deletion of (2, b)	justification(2, b)={2, 3}
	deletion of (2, c)	justification(2, c)={2, 3}
Add {2, 4}		
Add {2, 5}	deletion of (5, b)	justification(5, b)={2, 5}
	deletion of (5, c)	justification(5, c)={2, 5}
Add {5, 6}	deletion of (6, b)	justification(6, b)={5, 6}
	deletion of (6, c)	justification(6, c)={5, 6}
Add {6, 2}		

Relax {2, 3}: • step1: (2, b) and (2, c) are added because justification(2, b)={2, 3} and justification(2, c)={2, 3}. So, (5, b) and (5, c) are added and so (6, b) and (6, c) too.

• step2: (2, b) has no support on {2, 4} so (2, b) is deleted and justification(2, b)={2, 4}. So, (5, b) and (6, b) are removed too by propagation and their justifications are recorded: justification(5, b)={2, 5} or {5, 6} and justification(6, b)={5, 6} or {6, 2} (it depends of the order of the propagation of suppressions).

Remarks: - (2, a) is not added in step 1 of the relaxation process because its empty support on {1, 2} (its justification) is not affected by the {2, 3} retraction.

- when step 1 starts, (2, c) has no support on {6, 2}, but since its justification is not {6, 2}, it is added and the propagation shows that (2, c) deleted cannot be supported by a well-founded set of justifications. (2, c) is in the new arc-consistent domain.

- at the end of step 1 (2, b) is still without support on the constraint {2, 4}, so the classic arc-consistency process restarts, deleting (2, b) and propagating.

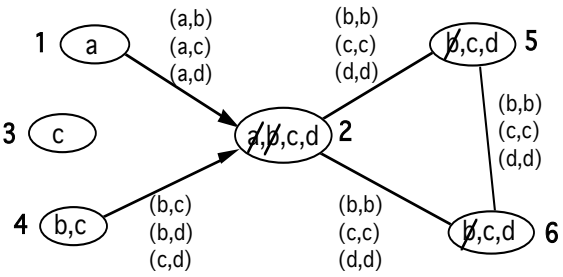

Figure 6: The CSP of fig.2 after the relaxation of the constraint {2, 3}

3.3. Correctness of DnAC-4

We outline here the key steps for a complete proof given in [Bessière 1991].

Notations:

$AC_\alpha(dom)$=the maximal arc-consistent domain of the CSP $p_{(\alpha)}$.

$dom = \{(i, a) / i \in X, a \in dom(i)\}$

$(i, a) \in D \Leftrightarrow D(i, a)$=**true**

TWS = $\{[(i, j), a] / counter[(i, j), a]=0$ **and** $(i, a) \in D\}$
(* TWS : true without support *)

Let E$/dom$: p(E) ≡ $\exists$ $(i, a) \in$ E, $justif(i, a)=j$ **and** S_{ija} (E = $\varnothing$

Basic properties:

(1) $S_{jib} = \{a / ((i, a), (j, b)) \in R_{ij}\}$

(2) $counter[(i, j), a]$ = number of supports of (i, a) on $\{i, j\}$ (*=$\wedge S_{ija}$ ($D\wedge$ because R_{ij} is symetric *)

Justification properties:

(3) $justif(i, a)=j \Rightarrow (i, a) \notin D$ **and** $counter[(i, j), a]=0$

(4) $(i, a) \notin D \Rightarrow justif(i, a) \neq$ **nil**

(5) $\forall$E$/dom\backslash D$: p(E)=**true** (* well-foundness property *)

Property 1: The assertion (TWS$/SL$) denoted (P1) is true after each procedure of DnAC-4.

Proof: (P1) was true for $p_{(0)}$ with $D=dom$ because c = $\varnothing$ $\Rightarrow$ TWS = $\varnothing$. We can easily verify this remains true by looking lines 8-9 of Beg-Add, lines 8-9 of Propag-Suppress and lines 7 and 12 of Init-Propag-Relax. ❑

Corollary 1: D is an arc-consistent domain at the end of Propag-Suppress.

Proof: We know that P1 is true after Propag-Suppress.

Line 1 $\Rightarrow SL = \varnothing$ at the end of Propag-Suppress

$\overset{(P1)}{\Rightarrow} \forall (i, a) \in D, \forall j / \{i, j\} \in c : counter[(i, j), a] > 0$

$\overset{(2)}{\Rightarrow} \forall (i, a) \in D, \forall j / \{i, j\} \in c$: (i, a) has a support (j, b) on $\{i, j\}$ in D

$\Rightarrow D$ is arc-consistent. ❑

Corollary 2: At the end of Add, D is arc-consistent.

Corollary 3: At the end of Relax, D is arc-consistent.

Lemma 1: The assertion ($AC_\alpha(dom) / D$) is not affected by Propag-Suppress.

Proof: Suppose ($AC_\alpha(dom) / D$) is true when Propag-Suppress starts. A value is removed from D if one of its counters is equal to zero. So, it has no supporting value in D on one constraint. Since ($AC_\alpha(dom) / D$) was true before its deletion, the value has no supporting value in $AC_\alpha(dom)$ on this constraint and couldnot be in $AC_\alpha(dom)$. So, ($AC_\alpha(dom) / D$) remains true after the deletion of the value. By induction it remains true during all Propag-Suppress. ❑

Corollary 4: ($AC_\alpha(dom) / D$) is true at the end of Add.

Proof: In the procedure Add, before the call of Propag-Suppress, $AC_\alpha(dom) / AC_{\alpha-1}(dom)=D$ since D is not affected by Beg-Add. From lemma 1 we deduce that ($AC_\alpha(dom) / D$) is true at the end of Add. ❑

Theorem 1: At the end of Add we have $AC_\alpha(dom)=D$.

Proof: From corollary 2 and 4. ❑

Lemma 2: ($AC_\alpha(dom) / D$) is true at the end of Init-Propag-Relax.

Proof: $AC_\alpha(dom)\backslash D \neq \varnothing$ implies (from (5)) that:

$\exists (i, a) \in AC_\alpha(dom)\backslash D, justif(i, a)=j$ **and** S_{ija} ($(AC_\alpha(dom)\backslash D) = \varnothing$

Now *justif*$(i, a)=j$ $\overset{(3)(2)}{\Rightarrow} S_{ija} \cap D = \varnothing$
so: $S_{ija} \cap \mathrm{AC}_\alpha(dom) = \varnothing$. It is a contradiction because $(i, a) \in \mathrm{AC}_\alpha(dom)$ ❑

Theorem 2: At the end of Relax we have $\mathrm{AC}_\alpha(dom)=D$.
Proof: From lemma 1, 2 and corollary 3. ❑

3.4. Complexity of DnAC-4

DnAC-4 uses a maximum of $2ed$ counters (with e the number of edges and d the maximal number of values in the domain of a variable). The tables D and *justif* have a size of nd. There are $2ed$ sets S_{ija}, each having a maximal size of d, so they need a total space of $O(ed^2)$ which is the space complexity of the algorithm.

The upper bound time complexity for DnAC-4 is trivially $O(ed^2)$, like for AC-4. More precisely, on one hand, during a restriction, DnAC-4 checks the consistency of more pairs of values than AC-4. The reason is that during a restriction, DnAC-4 builds S_{jib} and *counter*$[(i, j), a]$ even after the deletion of the value (i, a) from D, because it needs these informations for an hypothetic future relaxation. AC-4 stops this work, as soon as (i, a) is out of D. On the other hand, during a relaxation, DnAC-4 only checks values needed to verify the well-founded property of justifications. AC-4 handles all the new CSP.

3.5. Cases where DnAC-4 is performant

DnAC-4 is efficient when the phase of adding values is short.

This is the case when the constraint graph is not connected. Then, the propagation stay in one connected-component.

Actually, what seems the most important for DnAC-4 efficiency is chronology. Indeed, constraints are added step by step, and their coming date is important since the justification of a removed value is the first constraint for which the value had no support. This means that when a "young" constraint is retracted, the algorithm probably propagates a few, as the opposite of the retraction of an "old" constraint that can modify many justifications, and so can make the propagation larger.

4. Performance comparison

We compared AC-4 to DnAC-4 on randomly generated DCSPs. Two probabilistic parameters were used in the generation of CSPs. *pc* determines the probability that any two variables are directly connected, and *pu* the probability that any two values in an existing constraint are permitted. Two other parameters are n the number of variables and d the number of values for each variable.

For each randomly generated CSP, we counted the total number of consistency checks done in AC-4 and in DnAC-4 to achieve arc-consistency when we add successively all the constraints generated. Then, we choosed randomly a constraint which is responsible of at least one value deletion (if there exists any such constraint), and removed this constraint. We counted the number of consistency checks done in AC-4 and in DnAC-4 to achieve again arc-consistency in the CSP. We summed for each algorithm the number of consistency checks done during restrictions and relaxation. The comparison of the results indicates if DnAC-4 is better than AC-4 after only one relaxation (i.e. the number of consistency checks avoided during relaxation is more important than the number done in excess during restrictions).

We tested the algorithms on random DCSPs with 8, 12 and 16 variables, having respectively 16, 12 and 8 values. We tried three values for (*pc*, *pu*): (35, 65), (50, 50) and (65, 35). For each of the nine classes of CSPs defined, we made the test on 10 different instances of DCSPs to have a result representative of the class.

The results reported in the table below are the averages of the ten tests for each class.

	n=16 d=8			n=12 d=12			n=8 d=16		
	pc=35 pu=65	pc=50 pu=50	pc=65 pu=35	pc=35 pu=65	pc=50 pu=50	pc=65 pu=35	pc=35 pu=65	pc=50 pu=50	pc=65 pu=35
restrictions AC-4	3611	3866	4498	4140	4683	4223	3128	3654	3298
restrictions DAC-4	3622	4010	6968	4140	4698	4449	3128	3654	3337
relaxation AC-4	3536	3818	4526	3946	4541	4136	2793	3401	3142
relaxation DAC-4	4	28	266	0	11	33	0	0	12
total AC-4	7147	7684	9025	8086	9224	8359	5921	7055	6440
total DAC-4	3626	4038	7234	4140	4709	4482	3128	3654	3349
DAC-4 gain	49%	47%	20%	48%	48%	46%	47%	48%	47%

Figure 7: Results of comparison tests between AC-4 and DnAC-4

We can see that on all the classes of problems tested, after one relaxation of constraint DnAC-4 has recovered the time losed during restrictions. We found only three instances, in class 3, where AC-4 remains better than DnAC-4 after one relaxation. But in that class, CSPs are too restricted and much more than one relaxation is needed before the CSP accepts solutions. So, we can say that DnAC-4 can easily recover its extra-time-consuming.

The very good results after one relaxation in classes 1, 4, 7 and 8 are not really significant because CSPs in that classes are underconstrained, and doing a relaxation in that case is unlikely.

The last remark we can add is that randomly generated CSPs are not the best way to test efficiency of an algorithm. Constraints that are created are meaningless and propagations during relaxations always found very short in our tests could be larger in real applications, and so the algorithm DnAC-4 be less advantageous. But the gain during a relaxation is so important here in all DCSPs tested that we can hope DnAC-4 remains good on real applications.

DnAC-4 is currently under implementation on the SYNTHIA system [Janssen et al 1989].

5. Conclusion

We have defined what we call Dynamic CSPs and have provided an efficient algorithm (DnAC-4) achieving arc-consistency in DCSPs. We have compared the performances of DnAC-4 and AC-4 (the fastest arc-consistency algorithm on static CSPs) on many different randomly generated CSPs. If DnAC-4 uses a little more time than AC-4 to build an arc-consistent domain after a restriction, it is more efficient for a relaxation because it has learned informations about the reasons of the deletions of values.

DnAC-4 can be useful for many systems that work in a dynamic environment. It can easily be extended to non-binary CSPs (see [Bessière 1991]).

The data structure created for the algorithm DnAC-4 can be used too to answer requests of the system (or the expert), like: "why this value has been deleted ?". The explanation given is then the set of constraints currently justifying the deletion of the value. It is a TMS-like use.

Acknowledgments

I would like to thank particularly Marie-Catherine Vilarem who gives me advice and invaluable help in preparing this paper, and also Philippe Janssen and Philippe Jégou for their useful comments.

References

Bessière, C. 1991. *Using CSPs to encode TMSs*. Technical Report, LIRMM, Montpellier II, France

Dechter, R., and Dechter, A. 1988. *Belief Maintenance in Dynamic Constraint Networks*. in Proceedings AAAI-88, St Paul MN, 37-42

Dechter, R., and Pearl, J. 1988. *Network-Based Heuristics for Constraint-Satisfaction Problems*. Artificial Intelligence 34, 1-38

Doyle, J. 1979. *A Truth Maintenance System*. Artificial Intelligence 12, 231-272

Freuder, E.C. 1978. *Synthesizing Constraint Expressions*. Communications of the ACM Vol.21 No.11, 958-966

Janssen, P.; Jégou, P.; Nouguier, B.; and Vilarem, M.C. 1989. *Problèmes de Conception : une Approche basée sur la Satisfaction de Contraintes*. 9èmes Journées Internationales d'Avignon: Les Systèmes Experts et leurs Applications, 71-84

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

McAllester, D.A. 1980. *An Outlook on Truth Maintenance*. Technical Report AI Memo No.551, MIT, Boston MA

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