Arc-Consistency for Non-Binary Dynamic CSPs

Christian Bessiere

Arc-Consistency for Non-Binary Dynamic CSPs

Christian Bessière
LIRMM (UMR C 9928 CNRS / Université Montpellier II)
860, rue de Saint Priest 34090 Montpellier, France
Email: bessiere@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 recall what we name a dynamic CSP, and then, generalize the incremental algorithm achieving arc-consistency on binary dynamic CSPs to general dynamic CSPs. Like for the binary version of this algorithm, there is an advantage to use our specific algorithm for dynamic CSPs instead of the best static one, GAC4.

1. Introduction

Constraint satisfaction problems (CSPs) provide a simple and good framework to encode systems of constraints and are widely used to express static problems. Nevertheless, many problems in Artificial Intelligence involve reasoning in dynamic environments. For instance, 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 [1]). Every time a constraint has been added or removed we need to check if there still exist solutions in the CSP.

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. Almost 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. The only arc-consistency algorithm written for dynamic CSPs, DnAC-4 in [2], deals only with binary CSPs.

So, in this paper we recall that a Dynamic CSP (DCSP) is a sequence of static CSPs each resulting from the addition or retraction of a constraint in the preceding one ([3], [2]). We propose a generalized version of DnAC-4 to maintain arc-consistency in general 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 is a Dynamic CSP (2.2). Arc-consistency filtering method is introduced and the best algorithm achieving it (GAC4 in [4]) 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 GAC4 to reach better performances on DCSPs. Section 4 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) 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. A constraint $C_p(i_1,...,i_q)$ constraining the subset $\{i_1,...,i_q\}$ of X is a subset of the Cartesian product $dom(i_1)\times...\times dom(i_q)$, that specifies which values of the variables are compatible with each other. A constraint is usually represented by the set of all **tuples** which are not forbidden by it. 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. Consider for instance the CSP presented in fig. 1. Each node represents a variable whose values are explicitly indicated. Each constraint is explicitly given by the list of its admissible tuples (C_1 and C_5 are a strict lexicographic order along the arrows and C_2, C_3, C_4 are equality constraints).

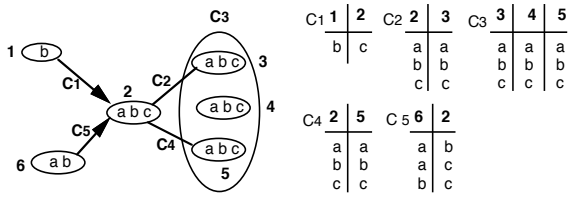

Figure 1: An example of CSP

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 subset of variables) or a **relaxation** (a constraint which 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 $p_{(\alpha)}=(X, dom, c_{(\alpha)})$, we will have $p_{(\alpha+1)}=(X, dom, c_{(\alpha+1)})$ where $c_{(\alpha+1)}=c_{(\alpha)} \pm C$, C being a constraint. $p_{(0)}=(X, dom, \varnothing)$.

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 ([5], [6], [7]). 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. The most widely used consistency algorithms are those achieving arc-consistency.

> **Definition.** A tuple u in a constraint C_p is viable iff for all variable i constrained by C_p, $u[i]$ is a viable value. A value a for a variable i is viable iff for all constraint C_p constraining i there exists a viable tuple u in C_p such that $u[i]=a$. A CSP is **arc-consistent** iff for all $i \in X$, for all $a \in dom(i)$, a is viable.

The complexity of the best algorithm building the largest arc-consistent domain in general CSPs (GAC4 in [4]) is $O(K)$ with K the sum of the lengths of all tuples for all constraints. It has been shown that it is an optimal algorithm.

The efficiency of GAC4 comes from the data structure it handles which is suitable to the arc-consistency property. 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 tuple which "supports" it at each constraint. GAC4 makes this support evident by assigning a set of supports to each constraint-value pair. The sets designed by S_{iap} contain the tuples in C_p that support the value a for the variable i. The algorithm is founded on a recursive value pruning, deleting a value when it has no support on a constraint, and then removing at their turn all tuples including this value. And so on until all remaining values have support 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 Masini's algorithm, GAC4, 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 support sets is empty after addition of constraints. But, when we remove a constraint (making a relaxation), GAC4 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. The incremental method

3.1. Introduction

As we have seen above, GAC4 does not have good properties (incrementality) for processing relaxations. So, in this section, we propose DnGAC4, a new arc-consistency algorithm for DCSPs. In DnGAC4 we extend GAC4 by recording some informations during restrictions while keeping its good properties. Then, DnGAC4 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. For every tuple deleted, we keep track of the value origin of its deletion as the "killer" of the tuple deleted. The killer is the first value retracted in the tuple. During a relaxation, with the help of justifications and killers, we can incrementally add to the current domain values which 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 set of justifications and killers must remain **well-founded** to ensure that the new domain is the maximal arc-consistent domain.

> **Definition.** Justifications and killers are well-founded iff in any set E of retracted values there exists a value (i, a) such that, its justification being the constraint C_p, every tuple of C_p including (i, a) has its killer out of E.

In other words, well-foundness means that every removed value is justified by a non-cyclic chain of justifications and killers. The value is not recorded as one of the reasons of its self deletion.

3.2. The algorithm DnGAC4

The algorithm we propose works with the following data structures:

• a table D of booleans keeps track of which values are in the current domain or not (the current domain being named D we confuse $(i, a) \in D$ and $D(i, a)$=**true**)

• a set of supports for each value on each constraint: S_{iap}={$u \in C_p$ /$(i, a) \in u$ **and** u is viable}

• for each value, the constraint origin of its deletion is stored in $justif(i, a)$.

if (i, a) is viable, $justif(i, a)$=**nil**

else $justif(i, a)$=the number of the first constraint on which (i, a) had got an empty support.

• for each tuple, we store in $killer(u)$ the first value deleted in it.

if u is viable, $killer(u)$=**nil**

else $killer(u)$=the first value in u retracted from D.

• a set of killed tuples for each value on each constraint: T_{iap}={$u \in C_p$ /$killer(u)=(i, a)$}

• the lists SL and RL respectively control the propagation of deletions and additions of values along the constraints; the list UL records for reconsideration the tuples put back to ensure well-foundness but that still have a killer.

Before giving the algorithms, we define the function *find-killer* which will be used in the following.

```
function find-killer(u: tuple): value;
    return (j, b) ∈ u /D(j, b)=false
    else return nil
```

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

```
for all (i, a) ∈ dom do
    D(i, a) , true; justif(i, a) , nil ;
```

Add a constraint C_p to the DCSP is done by calling the procedure Add.

```
procedure Add (Cp(i1,...,iq): constraint);
    Put Cp(i1,...,iq) in c;
    SL , ∅ ;
    Init-Add (Cp(i1,...,iq), SL );
    Propag-Suppress (SL );
```

• the procedure Init-Add creates lists S_{iap} and T_{iap} for all (i, a) such that i is constrained by C_p. It updates $killer$ for all tuple u in C_p, adding u in all S_{jbp} such that $(j, b) \in u$ if u is viable or in $T_{killer(u)p}$ if not. All (i, a, p) without support (S_{iap}=∅) are added to SL.

```
procedure Init-Add (in Cp(i1,...,iq): constraint;
                    in out SL: list);
1   for all i ∈ {i1,...,iq } do
2     for all a ∈ dom(i ) do Siap , ∅ ; Tiap , ∅
3   for all u ∈ Cp do
4     killer(u ) , find-killer(u );
5     if killer(u ) = nil then
6       for all (j, b ) ∈ u do Append(Sjbp, u );
7     else Append(Tkiller(u)p, u )
8   for all i ∈ {i1,...,iq } do
9     for all a ∈ dom(i ) do
10      if D(i, a ) and Siap = ∅ then Append(SL, (i, a, p ));
```

• the procedure Propag-Suppress takes constraint-value pairs (i, a, m) in SL, removes (i, a) from D, updates $justif(i, a)$ and also $killer(u)$ for all u viable such that $(i, a) \in u$. These tuples u are added to T_{iap} and removed from all S_{jbp} which they were belonging. If a S_{jbp} becomes empty, (j, b, p) is added in SL. And so on.

```
procedure Propag-Suppress (in out SL: list);
1   while SL ≠ ∅ do
2     keep(SL, (i, a, m ));
3     if D(i, a ) and Siam = ∅ then
4       justif(i, a ) , m ;
5       D(i, a ) , false;
6       for all Cp in c constraining i do
7         for all u ∈ Siap do
8           killer(u ) , (i, a ); Append(Tiap, u );
9           for all (j, b ) ∈ u do
10            Suppress(Sjbp, u );
11            if D(j, b ) and Sjbp = ∅ then
                Append(SL, (j, b, p ));
```

To retract the constraint $C_m(i_1,\ldots,i_q)$ from the DCSP we call the procedure Relax.

```
procedure Relax (Cm(i1,...,iq): constraint);
    SL , ∅ ;
    Init-Propag-Relax (Cm(i1,...,iq), SL );
    Propag-Suppress (SL )
```

Two parts are needed in the relaxation process to maintain the well-founded property.

• **part 1.** first, the procedure Init-Propag-Relax adds to D and puts in RL all values (i, a) for which deletion was justified by C_m (i.e. $justif(i, a)=m$). In a second step, the consequences of the addition of these values (i, a) to D are propagated. Tuples u deleted by them ($killer(u)=(i, a)$) are put back in the supports lists S_{jbp} of values (j, b) that they support (lines 13-15), except if there is another value of u for which we are sure that retraction is independant of u's retraction (lines 10-12). At each tuple put back, we add in D and in RL every (j, b) which has the supports set S_{jbp} that becomes non empty on its justification constraint ($p=justif(j, b)$) (line 16). Additions of values are recursively propagated: this second step processes while RL is non empty. In this second step, if a tuple is added when it

includes values not in D because we are not sure of the independance of the retraction of these values, we add it in a list UL which records these "suspended" tuples (line 17). In the third step, we check if these suspended tuples are really saved (their killers were not independent, they had not a well-founded justification) or if their killers are still present. In this latter case, we kill again the tuple and remove it from the supports sets of the values it supports.

During all the procedure Init-Propag-Relax, each time a support set S_{jbp} of an added value (j, b) is found empty, we put the constraint-value pair (j, b, p) in SL.

procedure Init-Propag-Relax (**in** $C_m(i_1,\ldots,i_q)$: constraint;
in out SL: list);
{Step 1: values with justification equal to C_m are added to D and put in RL for propagation}
1 $RL \leftarrow \emptyset$;
2 **for all** $i \in \{i_1,\ldots,i_q\}$ **do**
3 **for all** $a \in dom(i)$ **do**
4 **if** $justif(i, a)=m$ **then**
Append(RL, (i, a));
$justif(i, a) \leftarrow$ **nil**; $D(i, a) \leftarrow$ **true**
5 Remove C_m from c;
{Step 2: consequences of the addition to D of the values in RL are propagated }
6 **while** $RL \neq \emptyset$ **do**
7 keep (RL, (i, a));
8 **for all** C_p in c constraining i **do**
9 **for all** $u \in T_{iap}$ **do**
10 **if** $u \in D$ **and** $(\forall (j, b) \in u, justif(j, b) \neq p)$ **then**
11 $killer(u) \leftarrow$ find-killer(u);
12 Append($T_{killer(u)p}$, u)
else
13 $killer(u) \leftarrow$ **nil**;
14 **for all** $(j, b) \in u$ **do**
15 Append(S_{jbp}, u);
16 **if** $justif(j, b)=p$ **then**
Append(RL, (j, b));
$justif(j, b) \leftarrow$ **nil**; $D(j, b) \leftarrow$ **true**
17 **if** $u \in D$ **then** Append(UL, u);
18 $T_{iap} \leftarrow \emptyset$;
19 **if** $S_{iap} = \emptyset$ **then** Append(SL, (i, a, p));
{**endwhile**}
{Step 3: "suspended" tuples are checked }
20 **while** $UL \neq \emptyset$ **do**
21 keep(UL, u);
{ let C_p being the constraint including u }
22 $killer(u) \leftarrow$ find-killer(u);
23 **if** $killer(u) \neq$ **nil then**
24 Append($T_{killer(u)p}$, u);
25 **for all** $(j, b) \in u$ **do**
26 Suppress(S_{jbp}, u);
27 **if** $D(j, b)$ **and** $S_{jbp} = \emptyset$ **then**
Append(SL, (j, b, p))

• **part 2.** Propag-Suppress takes constraint-value pairs (i, a, m) in SL and removes again from D values (i, a) which still are without support on the constraint C_m ($S_{iam}=\emptyset$). These retractions are propagated on the tuples and on the other values: the classic arc-consistency process restarts.

We develop here DnGAC4 on the DCSP of fig.1, to show the mechanism of justifications and killers (notation: u_{pq} is the qth tuple of the constraint C_p).

Add C_1. $(2, a)$ and $(2, b)$ deleted: $justif(2, a),1$ and $justif(2, b),1$

Add C_2. $killer(u_{21}),(2, a)$ and $killer(u_{22}),(2, b)$.
$(3, a)$ and $(3, b)$ deleted: $justif(3, a),2$; $justif(3, b),2$

Add C_3. $killer(u_{31}),(3, a)$ and $killer(u_{32}),(3, b)$.
$(4, a)$, $(4, b)$, $(5, a)$, $(5, b)$ deleted: $justif(4, a),3$; $justif(4, b),3$; $justif(5, a),3$; $justif(5, b),3$

Add C_4. $killer(u_{41}),(2, a)$ and $killer(u_{42}),(2, b)$.

Add C_5. $killer(u_{51}),(2, b)$.

Relax C_1. $(2, a)$ and $(2, b)$ are added because $justif(2, a)=1$ and $justif(2, b)=1$. All tuples killed by $(2, a)$ and $(2, b)$ are examined. u_{21}, u_{22}, u_{51} are restored but not u_{41} and u_{42} because we can find another killer for them without losing well-foundness. So, new killers are recorded: $killer(u_{41}),(5, a)$ and $killer(u_{42}),(5, b)$. $(2, a)$ and $(2, b)$ are without support on C_4 so we add them to SL. The consequence of u_{21} and u_{22} rehabilitation is the addition of $(3, a)$ and $(3, b)$ to D. So, u_{31} and u_{32} are restored and $(4, a)$, $(4, b)$, $(5, a)$ and $(5, b)$ are added to D. Then, u_{41} and u_{42} are restored. The first part of the relaxation process is finished. The second part is the classical arc-consistency process, examining values in SL and propagating deletions. $(2, b)$ is in SL but has now a support on C_4 so nothing is done. $(2, a)$ is without support on C_5 so it is discarded and $justif(2, a),5$. So, $killer(u_{21}),(2, a)$ and $killer(u_{41}),(2, a)$. Then, $(3, a)$ and $(5, a)$ are discarded ($justif(3, a),2$, $justif(5, a),4$). Then, $killer(u_{31}),(3, a)$ and $(4, a)$ is discarded ($justif(4, a),3$). We have the new maximal arc-consistent domain.

3.3. Correctness of DnGAC4

We give here the properties true after each procedure of DnGAC4 which are needed to prove the correctness of the algorithm. For a complete proof we will see [8].

Basic properties.

(1) $S_{iap}=\{u \in C_p \,/\, (i, a) \in u \text{ and } u \in D\}$
(2) $T_{iap}=\{u \in C_p \,/\, (i, a)=killer(u)\}$

Justifications and killers properties.

(3) $justif(i, a)=p \Rightarrow (i, a) \notin D$ **and** $S_{iap}=\emptyset$
(4) $\{(i, a) \notin D \,/\, justif(i, a)=$**nil**$\}=\emptyset$

(5) $\{(i, a, p) / S_{iap}=\varnothing$ **and** $(i, a) \in D\} / SL$

(6) $killer(u)=(i, a) \Rightarrow (i, a) \in u$ **and** $(i, a) \notin D$

(7) u ! $D \Rightarrow killer(u)\neq$**nil**

(8) $\forall E/dom\backslash D$: $\exists (i, a) \in E$: $justif(i, a)=p$ **and** $\forall u \in C_p$ $/(i, a) \in u$: $killer(u) \notin E$ {well-founded property}

From (5) we can deduce that D is arc-consistent at the end of Propag-Suppress, so at the end of Add or Relax too. From Propag-Suppress construction we can see that if D includes the maximal arc-consistent domain at its begining, this remains true at its end. Now, we need only to prove that when Propag-Suppress begins, we always have D including the maximal arc-consistent domain. It is evident in Add, and for Relax, we can prove this using properties (6), (7), (8) and looking Init-Propag-Relax construction.

3.4. Complexity of DnGAC4

In DnGAC4, the bigger data structures are the S_{iap} and T_{iap} lists of tuples. They take the same space as all the constraints C_p in the CSP. This space is linear in the sum of the lengths of all the tuples of all the constraints (it is linear in the size of the problem).

The upper bound time complexity of DnGAC4, like in GAC4, is linear in the size of the problem since the atomic operations are first Append(S_{iap}, u), and then Suppress(S_{iap}, u) (the total size of the S_{iap} sets being bounded by the size of the problem). But these operations must be processed in constant time. A suitable data structure is shown in [4].

If we want to specify the complexity of DnGAC4, we can say that, on one hand, during a restriction, DnGAC4 does a little more processing than GAC4, building a more complex data structure (for each tuple a killer is stored, and for each value we record a justification). On the other hand, during a relaxation, DnGAC4 only checks values and tuples needed to verify the well-founded property of justifications and killers. GAC4 handles all the new CSP.

3.5. Performance comparison

There are too many parameters in general DCSPs to provide significant tests on randomly generated DCSPs but it seems that DnGAC4 on general DCSPs works at least as better as DnAC-4 on binary DCSPs. If DnGAC4 is applied on CSPs containing only ternary constraints, we can see that it generally outperforms GAC4 after the first relaxation of constraint, reaching a gain of 35% (average of the classes of CSPs tested) on the sum of the running time of all restrictions plus the relaxation. More, if we modify the DCSP several times, adding and removing constraints until we find a "good" consistent CSP (what is done in all design problems) we remark that the gain of DnGAC4 fastly grows and overtops 50% after two or three relaxations. More changes will be done, more the use of DnGAC4 will be interesting.

DnGAC4 is now under implementation on SYNTHIA [1].

4. Conclusion

We have defined what we call Dynamic CSPs and have underlined that there does not exist algorithms incrementally maintaining arc-consistency in general DCSPs (there exists one only for binary DCSPs). We have noticed that arc-consistency is the most used filtering technique in CSPs, and so, it is important to have an incremental algorithm for all DCSPs. Then, we have provided an efficient algorithm (DnGAC4) incrementally achieving arc-consistency in general DCSPs. DnGAC4 can be useful for many systems which work in a dynamic environment.

Acknowledgements

I would like to thank particularly Marie-Catherine Vilarem for the implementation of my work on SYNTHIA and Philippe David who gives me constructive suggestions for the final draft.

References

[1] 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

[2] C. Bessière: "*Arc-Consistency in Dynamic Constraint Satisfaction Problems*"; Proceedings AAAI-91, Anaheim CA, 221-226

[3] R. Dechter, A. Dechter: "*Belief Maintenance in Dynamic Constraint Networks*"; Proceedings AAAI-88, St Paul MN, 37-42

[4] R. Mohr, G. Masini: "*Good Old Discrete Relaxation*"; Proceedings ECAI-88, München, FRG, 651-656

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

[6] A.K. Mackworth: "*Consistency in Networks of Relations*"; Artificial Intelligence 8 (1977) 99-118

[7] E.C. Freuder: "*Synthesizing Constraint Expressions*"; Communications of the ACM Nov. 1978 Vol.21 No.11, 958-966

[8] C. Bessière: "*Algorithmes d'arc-consistance pour les CSP dynamiques*"; Tech.Rep. 91-086, LIRMM Montpellier II, France, August 1991 (in French)