Maintaining Alternative Values in Constraint-Based Configuration*

Caroline Becker
IRIT - University of Toulouse
France
becker@irit.fr

Hélène Fargier
IRIT - University of Toulouse
France
fargier@irit.fr

Abstract

Constraint programming techniques are widely used to model and solve interactive decision problems, and especially configuration problems. In this type of application, the configurable product is described by means of a set of constraints bearing on the configuration variables. The user interactively solves the CSP by assigning the variables according to her preferences. The system then has to keep the domains of the other variables consistent with these choices. Since maintaining the global inverse consistency of the domains is not tractable, the domains are instead filtered according to some level of local consistency, e.g. arc-consistency.

The present paper aims at offering a more convenient interaction by providing the user with possible alternative values for the already assigned variables, i.e. values that could replace the current ones without leading to a constraint violation. We thus present the new concept of *alternative domains* in a (possibly) partially assigned CSP. We propose a propagation algorithm that computes all the alternative domains in a single step. Its worst case complexity is comparable to the one of the naive algorithm that would run a full propagation for each variable, but its experimental efficiency is better.

1 Introduction

The Constraint Satisfaction Problem (CSP) formalism offers a powerful framework for representing a great variety of problems, e.g. routing problems, resource allocation, frequency assignment, configuration problems, etc. The main task addressed by the algorithms is the determination of the consistency of the CSP and/or the search for an (optimal) solution, and this is a difficult task: determining whether a CSP is consistent is an NP-complete request. In the CSP community, the main research stream thus addresses this question, either directly (looking for efficient complete algorithms) or getting around (studying the polynomial subclasses or proposing incomplete algorithms).

*This work is partially funded by the ANR project "Business Recommendation for Configurable Products" (BR4CP), ANR-11-BS02-008

But these algorithms do not help solving decision support problems that are interactive in essence. For such problems, the user herself is in charge of the choice of values for the variables and the role of the system is not to solve a CSP, but to help the user in this task. Constraint-based product configuration [Mittal and Frayman, 1989; Sabin and Weigel, 1998; Mailharro, 1998; Stumptner *et al.*, 1998; Junker, 2006] is a typical example of such problems: a configurable product is defined by a finite set of components, options, or more generally by a set of attributes, the values of which have to be chosen by the user. These values must satisfy a finite set of constraints that encode the feasibility of the product, the compatibility between components, their availability, etc.

Several extensions of the CSP paradigm have been proposed in order to handle the constraints-based definition of a catalog or a range of products, and more specifically the definition of configurable products. These extensions have been motivated by difficulties and characteristics that are specific to the modeling and the handling of catalogs of configurable products. Dynamic CSPs [Mittal and Falkenhainer, 1990], for instance suit problems where the existence of some optional variables depends on the values of some other variables. Other extensions proposed by the CSP community include composite CSPs [Sabin and Freuder, 1996], interactive CSPs [Gelle and Weigel, 1996], hypothesis CSPs [Amilhastre *et al.*, 2002], generative constraint satisfaction [Stumptner *et al.*, 1998; Fleischanderl *et al.*, 1998], etc.

In this paper, we do not deal with such representation problems: we assume that the product range is specified by a classical CSP. Instead, our work focuses on the human-computer interaction. When configuring a product, the user specifies her requirements by interactively giving values to variables. Each time a new choice is made, the domains of the further variables must be pruned so as to ensure that the values available in their domains can lead to a feasible product (i.e., a product satisfying all the initial configuration constraints): the aim of the system is to keep the domains of the other variables consistent with these choices. Since maintaining the global inverse consistency is generally not tractable, the domains are rather filtered according to some level of local consistency, e.g. arc-consistency. In the present paper, we propose to make this interaction more user-friendly by showing not only (locally) consistent domains, but also what we call the alternative domains of the assigned variables, i.e. the

values that could replace the ones of the assigned variable without leading to the violation of some constraint.

The structure of the present article is as follows: the problematics of alternative domains is described in the next Section. Section 3 then develops the basis of our algorithm. Our first experimental results are shown in Section 4. Proofs are omitted for the sake of brevity [1].

2 Background and Problematics

A CSP is classically defined by a triplet $(\mathcal{X}, \mathcal{D}, \mathcal{C})$ where $\mathcal{X} = \{x_1, \ldots, x_m\}$ is a finite set of m variables, each x_i taking its values in a finite domain $D(x_i)$, and a finite set of constraints $\mathcal{C}$. We note $\mathcal{D} = \prod_{j=1}^{n} D(x_j)$. An assignment t of a set of variable $\mathcal{Y} \subseteq \mathcal{X}$ is an element of the cartesian product of the domains of these variables; for any $x_j \in \mathcal{Y}$ we denote by $t[x_j]$ the value assigned to x_j in t.

A constraint C in $\mathcal{C}$ involves a set $vars(C) \subseteq \mathcal{X}$ and can be viewed as a function from the set of assignments of $vars(C)$ to $\{\top, \bot\}$: $C(t) = \top$ iff t satisfies the constraint; for any x_j in $vars(C)$ and any v in its domain, we say that an assignment t of $vars(C)$ is a support of this value (more precisely, of (x_j, v) on C) iff $t[x_j] = v$ and t satisfies C.

An assignment t of $\mathcal{X}$ is a solution of the CSP iff it satisfies all the constraints. If such a solution exists, the CSP is said to be consistent, otherwise it is inconsistent.

Formally, a configurable product is represented as a CSP $(\mathcal{X}, \mathcal{D}, \mathcal{C})$ and the current choices of the user by a set of couples (x_i, v) where x_i is a variable in $\mathcal{X}$ and v the value assigned to x_i. Following [Amilhastre *et al.*, 2002], the problem can be represented by an Assumption-based CSP (A-CSP).

Definition 1 (A-CSP) *An* A-CSP *is a 4-tuple* $(\mathcal{X}, \mathcal{D}, \mathcal{C}, \mathcal{H})$ *where* $(\mathcal{X}, \mathcal{D}, \mathcal{C})$ *is a CSP and* $\mathcal{H}$ *a finite set of constraints on variables of* $\mathcal{X}$.

In configuration, $\mathcal{H}$ represents the current user choices, i.e. assignments of the variables: *we suppose in the sequel that all the restrictions in* $\mathcal{H}$ *bear with different variables and restrict their domains to a unique value*[2]; we will denote by $h_i = (x_i \leftarrow v)$ the restriction from $\mathcal{H}$ on x_i, if it exists.

After each choice, the system filters the variables' domains, ideally leaving only the values compatible with current choices. Since such a computation is intractable in the general case, a weaker level of consistency is ensured in real applications, generally arc-consistency. Recall that a CSP is said to be arc consistent in the general sense (GAC) iff, for any variable $x_j \in \mathcal{X}$ and any value v in its domain, for any constraint C bearing on x_j, there exists an assignment t of the variables of C in their domains such that t is a support of (x_j, v). The role of an arc consistency algorithm is to remove from the domains the values that do not have any support so as to compute a CSP that is equivalent to the first one (i.e. having the same set of solutions) and arc consistent; this problem is called the closure by arc consistency of the first one.

Other, more powerful, levels of local consistency can be ensured, e.g. Path Inverse Consistency [Debruyne, 2000], Singleton Arc Consistency [Debruyne and Bessière, 1997], k-inverse consistency [Freuder, 1985; Freuder and Elfe, 1996]. In the following definitions, we do not make any assumption on the level of local consistency that is ensured. We simply consider that, after each choice, an algorithm is called that ensures some level a of local consistency - i.e. that computes the closure by a consistency of the original problem. We call the *current domain* of a variable its domain in this closure.

Definition 2 (Current domain of a variable) *Let* a *be a level of local consistency and* $P = (\mathcal{X}, \mathcal{D}, \mathcal{C}, \mathcal{H})$ *an A-CSP. The* current domain *according to* a*-consistency of a variable* x_i *is its domain in the closure by* a*-consistency of* $(\mathcal{X}, \mathcal{D}, \mathcal{C} \cup \mathcal{H})$.

We can now formally define the notion of alternative domain of an assigned variable as the current domain that it would have if the user would take this assignment back:

Definition 3 (Alternative domain) *Let* a *be a level of local consistency and* $P = (\mathcal{X}, \mathcal{D}, \mathcal{C}, \mathcal{H})$ *an A-CSP. The alternative domain of a variable* x_i *according to* a *is its domain in the closure by* a*-consistency of the CSP*$(\mathcal{X}, \mathcal{D}, \mathcal{C} \cup \mathcal{H} \setminus \{h_i\})$. *We write it* $D^a_{alt}(x_i)$.

A value v is thus an alternative value for x_i either if it belongs to the current domain of x_i (it is in particular the case when x_i is assigned to v), or if (i) x_i is assigned to another value than v and (ii) the single relaxation of this assignment would make v a-consistent. For instance, if x_i is the last assigned variable, all the values that were in the domain of x_i just before this assignment are alternative values. An example of alternative domain can be seen on Table 1. Notice that alternative values are generally not interchangeable in the full sense [Freuder, 1991] with assigned values (nor with each other): it is possible that only one of theses values belongs to a solution. Alternative values are alternatives *w.r.t the level of local consistency considered.* Notice also that the previous definition theoretically applies to any level of local consistency, but its application is actually restricted to local consistencies that remove values from the domains.

The notion of alternative domain is orthogonal to the notion of removal's explanation, such as proposed in PaLM [Jussien and Ouis, 2001; Jussien and Barichard, 2000]: explanations are a way to explain the pruning of the domains and aim at proposing a strategy of restoration of some values for an

	1	2	3	4
x_1	$\star$	$\diamond$	$\diamond$	$\times$
x_2	$\times$	$\diamond$	$\diamond$	$\star$
x_3	$\times$	$\diamond$	$\diamond$	$\times$

Table 1: Assigned ($\star$), forbidden ($\times$) and alternative ($\diamond$) values for the A-CSP $\mathcal{X} = \{x_1, x_2, x_3\}$, $\mathcal{D} = \{1, 2, 3, 4\}^3$, $\mathcal{C} = \{Alldiff(x_1, x_2, x_3)\}$, $\mathcal{H} = \{(x_1 \leftarrow 1), (x_2 \leftarrow 4)\}$.

[1] A full-proof version of the paper is available at `ftp://ftp.irit.fr/IRIT/ADRIA/ijcai13BF.pdf`

[2] Actually, the definitions and results could be set in a more general framework and capture any type of restriction, and in particular restrictions of the form "$x_i \in A$" or "$x_i \neq v$"; the meaning of alternative values when the restrictions in $\mathcal{H}$ bear on more than one variable is nevertheless questionable, hence our assumption.

unassigned variable by the relaxation of a (minimal) subset of user's choices. On the contrary, the alternative domain of a variable provides a way to change the value of an assigned variable *without* any modification of her other choices.

The notion of alternative domain can be compared to the concept of $(1,0)$-super-solution proposed by Hebrard et al. [Hebrard *et al.*, 2004]. An (a,b)-super-solution is a solution of the CSP such that, if a variables lose their values, it can be repaired by assigning these variables with new values and modifying the values of at most b other variables. It is a generalization of the notion of fault tolerant solution [Weigel and Bliek, 1998], a fault tolerant solution being a $(1,0)$-super-solution. A fault tolerant solution is actually a solution such as all the variables have a non-empty alternative domain: if one of the current values in the assignment is made unavailable for any reason, a solution can still be found by choosing a value from its alternative domain - this value is by definition compatible with the other choices. Formally,

Proposition 1 *Let d be a solution of a CSP $(\mathcal{X},\mathcal{D},\mathcal{C})$ and let $\mathcal{H} = \{h_i = (x_i \leftarrow d[x_i]), i = 1,\ldots,card(\mathcal{X})\}$.*
d is a $(1,0)$-super-solution of $(\mathcal{X},\mathcal{D},\mathcal{C})$ if and only if, $\forall x_i \in \mathcal{X}$, the alternative domain of x_i according to arc-consistency is a strict superset of $\{d[x_i]\}$.

The main difference between $(1,0)$ super-solutions and alternative domains is that super-solutions deal with complete assignments while alternative domains suggest restoration values even for partial assignments. Technically, this implies that algorithms computing super-solutions that are based on variable duplication do not work for the computation of alternative domains, as shown in Figure 1. In this example, x'_i denotes the duplicates of the original variables x_i; consider that the user has chosen $x_1 \leftarrow 1$ and $x_2 \leftarrow 2$, the arc-consistency filtering reduces the domain of x_3 to $\{3\}$; the domain of x'_1, which were supposed to hold the alternative values of x_1, thus becomes empty, while 3 still belongs to the alternative domain of x_1.

We can finally notice that the two notions target different practical goals: when refereing to a super-solution, we are looking for *some*, but not all, robust (and *complete*) solutions - there is indeed a potentially exponential number of super-solutions. When computing alternative domains, we are looking for *all* the alternative values, and this even during the search, when the assignments are *partial*.

3 Computing alternative domains

When n variables are assigned, a naive way of computing the alternative domains of these variables is to make $n+1$ copies of the CSP: a reference CSP P_0 (where all the n variables are assigned), and n CSP P_i where each P_i has exactly the same assignments than P_0, with the exception of variable x_i which remains unassigned in P_i. Each P_i is filtered by a-consistency. The alternative domain of variable x_i in obviously the domain of x_i in the arc consistent closure of P_i. This method does not require much space but does a lot of redundant computations. It will be the reference point from our method, which follows the opposite philosophy: memorizing information in order to avoid a duplicate work.

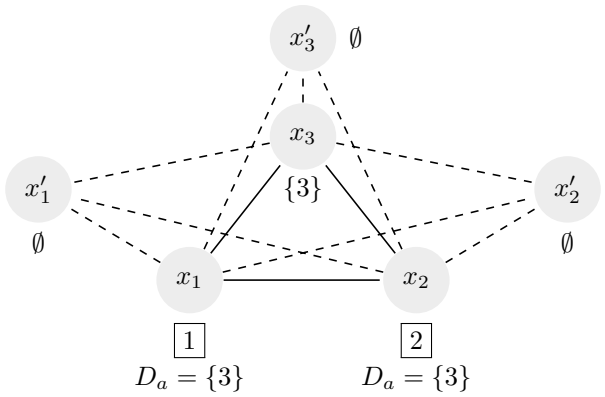

Figure 1: An attempt of computation by variable duplication of the arc consistent alternative domains of the A-CSP $X = \{x_1,x_2,x_3\}, \mathcal{D} = \{1,2,3\}^3, \mathcal{C} = \{x_1 \neq x_2, x_2 \neq x_3, x_1 \neq x_3,\}, \mathcal{H} = \{(x_1 \leftarrow 1),(x_2 \leftarrow 2)\}$).

3.1 Removals and sufficient justifications

The main idea of our approach is to maintain, for each value removed by the filtering algorithm, a vector of boolean flags, one flag for each $h_i \in \mathcal{H}$. The flag on h_i must be true if and only if the single relaxation of the user's choice h_i will lead to have the value back in the domain of its variable. Let us formalize:

Definition 4 (Removal, invalid tuple) *Let $P = (\mathcal{X},\mathcal{D},\mathcal{C},\mathcal{H})$ be an A-CSP and P^a the closure of $(\mathcal{X},\mathcal{D},\mathcal{C}\cup\mathcal{H})$ by some level of local consistency a.*

A removal w.r.t. a is a pair (x_j,v), $x_j \in \mathcal{X}, v \in D(x_j)$ such that v does not belong to the domain of x_j in P^a
We write $\mathcal{R}^a$ the set of removals of P w.r.t. a.

Let C be a constraint in $\mathcal{C}$ and t an assignment of $vars(C)$ satisfying C. t is said to be invalid w.r.t. a iff there exists $x_j \in vars(C)$ such that $t[x_j]$ does not belong to domain of x_j in P^a; otherwise, it is said to be valid w.r.t. a.

To improve readability, a removal (x_j,v) will often be written $(x_j \neq v)$, and we will omit to mention level a of local consistency to which the removal refers when not ambiguous.

Definition 5 (Sufficient Justification of a removal) *Let $P = (\mathcal{X},\mathcal{D},\mathcal{C},\mathcal{H})$ be an A-CSP, a a level of local consistency, and $\mathcal{R}^a$ the $P's$ removals according to a.*

For any x_j in $\mathcal{X}$ and any v in $D(x_j)$, $h_i \in \mathcal{H}$ is said to be an a-sufficient justification of v for x_j if and only if v belongs to the domain of x_j in the a-consistent closure of $(\mathcal{X},\mathcal{D},\mathcal{C}\cup\mathcal{H}\setminus\{h_i\})$.

For instance, if the propagation of the last assignment leads to the removal of the value v in the domain of x, this assignment is a sufficient justification of $x \neq v$. By extension, any h_i is a sufficient justification of a value that does belong to the current domain of its variable. Of course:

		supports of $(x = v)$			
removals	justifications	t_1	t_2	t_3	t_4
$x_1 \neq v_1$	$\{h_1, h_2\}$	$\star$	$\star$		
$x_2 \neq v_2$	$\{h_1, h_3\}$	$\star$			$\star$
$x_3 \neq v_3$	$\{h_2, h_4\}$		$\star$	$\star$	$\star$
		$\{h_1\}$	$\{h_2\}$	$\{h_2, h_4\}$	$\emptyset$
$x \neq v$	$\{h_1, h_2, h_4\}$				

Table 2: Computing the justifications of the removal $(x \neq v)$ on a constraint; the t_i are the supports of $x = v$. A $\star$ in some cell $(t_i, x_j \neq v_j)$ means that t_i is invalid when $x_j \neq v_j$.

Proposition 2 *Let $P = (\mathcal{X}, \mathcal{D}, \mathcal{C}, \mathcal{H})$ be an A-CSP, a a level of local consistency.*

For any $x_i \in \mathcal{X}$, any $v \in D(x_i)$, v belongs to the alternative domain of x_i iff either v belongs to the domain of x_i in the closure by a consistency of $P = (\mathcal{X}, \mathcal{D}, \mathcal{C} \cup \mathcal{H})$ or $(x_i \neq v) \in \mathcal{R}^a$ and h_i is a sufficient justification of $(x_i \neq v)$.

The notion of sufficient justification extends to tuples:

Definition 6 (Sufficient justification of a tuple) *Let $P = (\mathcal{X}, \mathcal{D}, \mathcal{C}, \mathcal{H})$ be an A-CSP, a a level of local consistency, C a constraint in $\mathcal{C}$ and t an assignment of $vars(C)$ satisfying C.*

A user choice $h_i \in \mathcal{H}$ is said to be an a- sufficient justification for t if and only if, for each $x_j \in vars(t)$, $t[x_j]$ belongs to the domain of x_j in the closure by a consistency of the CSP $(\mathcal{X}, \mathcal{D}, \mathcal{C} \cup \mathcal{H} \setminus \{h_i\})$.

Our algorithm is based on the fact that an assignment h_i is an a-sufficient justification for the tuple t if and only if, for each x_j involved by the tuple, either $t[x_j]$ is in the current domain of x_j or h_i is a sufficient justification of the removal $(x_j \neq t[x_j])$. Formally, let us call the *conflict set* of t the set of removals that make it invalid:

Definition 7 (Conflict set)
The conflict set of a tuple t w.r.t. some level of a consistency is the subset of $\mathcal{R}^a$ defined by: $CS(t) = \{(x_i \neq v) \in \mathcal{R}^a \text{ s. t. } t[x_i] = v\}$.

Of course, a tuple is invalid iff it has a non-empty conflict set.

Proposition 3 *h_i is an a-sufficient justification of a tuple t if and only it is an a-sufficient justification of each of the removals in its conflict set w.r.t. a.*

Finally, it can easily be shown that, when the level of local consistency to be maintained is generalized arc consistency:

Proposition 4 *h_i is a sufficient justification w.r.t. Arc consistency (GAC) for a removal $(x \neq v)$ iff, for each constraint C bearing on x, there exists a tuple t support of $(x = v)$ on C such that h_i is a GAC-sufficient justification of t.*

Similar properties can be established for other levels of local consistency based on the notion of support, e.g. for k inverse consistency [Freuder, 1985][3].

[3] A CSP is $(1, k)$ consistent iff, for each variable x and each value v in $D(x)$, for each set $\mathcal{V}$ of k additional variables, $x = v$ has a support on $\mathcal{V}$, i.e. there exists an assignment t of $\{x\} \cup \mathcal{V}$ such that for any $C \in \mathcal{C}$ with $vars(C) \subseteq \{x\} \cup \mathcal{V}$, t satisfies C.

Proposition 5 *$h_i \in \mathcal{H}$ is a $(1, k)$-sufficient justification of $(x \neq v) \in \mathcal{R}^{(1,k)}$ iff, for each set $\mathcal{V}$ of k variables there exists a support t of $x = v$ on $\mathcal{V}$ such as h_i is a $(1, k)$-sufficient justification of t.*

3.2 An algorithm for maintaining the alternative domains w.r.t. Arc Consistency

In our application, interactive configuration, the constraints to be taken into account are mostly table constraints and the level of consistency referred to is Generalized Arc Consistency. We propose to maintain the alternative domain upon the assignment of a variable using an extension of GAC4 [Mohr and Masini, 1988]. Our algorithm propagates not only value removals, but also justifications: for each removal $(x_i \neq v)$, we maintain a vector $f_{(x_i \neq v)}$ of n boolean flags, one for each choice in $\mathcal{H}$, such that $f_{(x \neq v)}(h_i) =$ True if and only if h_i is a sufficient justification of $(x_i \neq v)$. According to Proposition 4, $f_{(x \neq v)}$ depends on the justifications of the tuples that support (x, v). Hence, we keep, for each tuple t, a bit vector f_t such as, for each h_i, $f_t[h_i]$ is true iff h_i is a sufficient justification of t. Intuitively (see Table 2 for an example), for the user choice h_i to be a sufficient justification for a removal $(x \neq v)$ provoked by constraint C, it is needed that the relaxation of h_i makes at least one support t of $(x = v)$ on C valid again, i.e. that all the elements in the conflict set of t have h_i as a sufficient justification (this is the meaning of Proposition 3). In other words, f_t is the intersection of the $f_{(x_j \neq w)}$ flags of all the removals $(x_j \neq w)$ in the conflict set of t. Formally:

Proposition 6 *Let $Support(x, v, C)$ is the set of assignments of $vars(C)$ that support (x, v); it holds that*

$$f_{(x \neq v)} = \bigwedge_{C | x \in vars(C)} \left(\bigvee_{t \in Support(x,v,C)} \left(\bigwedge_{r \in CS(t)} f_r \right) \right)$$

We propose here a GAC4 like algorithm, the initialization and main propagation of which are depicted by Algorithms 1 and 2. We use the following notations:

- $(\mathcal{X}, \mathcal{D}, \mathcal{C})$ is the original CSP, that is supposed to be arc consistent;
- $Table(C)$ is the assignments of $vars(c)$ that satisfy C. Moreover the tuples involved in the tables are valid (i.e. $Table(c)$ is a subset of the cartesian product of the domains of the variables its bears on).
- $D_c(x_i)$ is the current domain of x_i
- $S_{x_i,v,C}$ is the set of supports of (x_i, v) on C and $Cpt(x_i, v, C)$ is the number of supports of (x_i, v) on C.
- f_t is the vector of justifications of tuple t; $f_{(x_i \neq v)}$ is the vector of justifications of removal $(x_i \neq v)$; for any removal $(x_i \neq v)$ and any constraint C bearing on x_i, $f_{(x_i \neq v, C)}$ is the vector of justification of $(x_i \neq v)$ on C.

The difference with GAC4 is that a removal $(x \neq v)$ must be propagated not only when it is created, but every time its vector of justifications changes. Since updating the vectors of justification is monotonic (an h_i might go from being sufficient to not, but not the other way around), the algorithm terminates. More precisely, instead of entering just once in Q,

Algorithm 1: Initialization

Procedure Initialize($(\mathcal{X}, \mathcal{D}, \mathcal{C})$:CSP; n: integer)
/* $(\mathcal{X}, \mathcal{D}, \mathcal{C})$ is the original CSP assumed to be arc consistent */
/* All the tuples are supposed to be valid */
/* n is the maximal number of assumptions to be considered */
begin
  foreach $C \in \mathcal{C}$ **do**
    foreach $x_i \in vars(C)$, $v \in D(x_i)$ **do**
      $Cpt(x_i, v, C) \leftarrow 0$;
      $S_{i,v,C} \leftarrow \emptyset$
    foreach $t \in Table(C)$ **do**
      $f_t \leftarrow \text{True}^n$;
      $valid(t) \leftarrow$ True;
      $CS(t) \leftarrow \text{False}^n$;
      foreach $x_i \in vars(C)$ **do**
        $Cpt(x_i, t[x_i], C) + +$;
        Add t to $S_{i,t[x_i],C}$

each removal can enter in the queue n times at most (n being the number of h_i in $\mathcal{H}$), i.e.as many times as the number of possible changes in a vector of justifications. The worst case complexity is thus bounded by $O(ned^k)$ with e the number of constraints, m the number of variables, n the maximal number of assumptions (typically, $n = m$), d the maximum size of the domains and k the maximum arity of constraints. It is thus the same complexity as the GAC-4 based naive method: $n.O(e.d^k)$ - with the important difference that in the naive method, GAC-4 is called exactly n times while n is a worst case bound for justification-based algorithm.

Concerning space complexity, GAC4 memorizes the support $S_{i,v,C}$ for each x_i , each value v in its domain and each constraint C bearing on x_i; Let say that this structure is in $O(T)$ (T is actually proportional to the space taken by valid tuples in constraint tables). Our algorithm also maintains, for each tuple t, a vector of n flags, meaning a $O(T.n)$ space. For each removal and each constraint bearing on the variable of the removal, we also keep a vector of n boolean flags. Since the number of removals is bounded by the number of variable/value pairs (x_i, v) in the problem, the algorithm involves in the worst case as many boolean vectors as the number of $S_{i,v,C}$ sets used by GAC4; Hence a global a spatial consumption bounded by $O(n.T)$.

4 Experimental results

We have tested this algorithm on three industrial problems of car configuration [4]. The names and values have been banalized for confidentiality reasons - we call these problems `Small`, `Medium` and `Big`.

- `Small` : 139 variables, domain sizes from 2 to 16, 147 constraints of arity 2 to 6 (68Ko, a total of 3.041 tuples).

[4]These instances have been built in collaboration with the car manufacturer Renault; see [Astesana *et al.*, 2013] for a more detailled description.

Algorithm 2: Propagation of decision $h_k = (x_k \leftarrow w)$

Procedure Propagate((x_k, w): assumption; $(\mathcal{X}, \mathcal{D}, \mathcal{C})$: the initial CSP; $\mathcal{H}$: the past assumptions; D_c: the current domains);
Add (x_k, w) to $\mathcal{H}$;
$Q \leftarrow \emptyset$;
/* The removal of the other values in the current domain of x_k is due to h_k */
foreach $u \neq w \in D_c(x_k)$ **do**
  $f_{(x_k \neq u)} \leftarrow \text{False}^m$;
  $f_{(x_k \neq u)}[h_k] \leftarrow$ True;
Add $(x_k \neq u)$ to Q;
while $Q \neq \emptyset$ **do**
  Choose and remove an $(x_i \neq v)$ from Q;
  if $v \in D_c(x_i)$ **then**
    Remove v from $D_c(x_i)$;
  foreach C *s.t.* $x_i \in vars(C)$ **and each** *tuple* t *in* $S_{i,v,C}$ **do**
    $Mem \leftarrow f_t$;
    $f_t \leftarrow f_t \wedge f_{(x_i \neq v)}$;
    if *valid(t)* **then**
      foreach $x_j \in vars(t)$ *s.t.* $j \neq i$ **do**
        $Cpt(x_j, t[x_j], C)^{--}$;
        if $Cpt(x_j, t[x_j], C) == 0$ **then**
          $f_{(x_j \neq t[x_j]),C} \leftarrow \text{False}^m$ /* init; will be computed later */;
          Add $(x_j \neq t[x_j])$ to Q;
          if $t[x_j] \in D_c(x_j)$ **then**
            $f_{(x_j \neq t[x_j])} \leftarrow \text{True}^m$ /* init */;
      $valid(t) \leftarrow false$;
    if $Mem! = f_t$ /* A justif. of t is not sufficient anymore */ **then**
      foreach $x_j \in vars(t)$ *s.t.* $j \neq i$ **do**
        $mem' \leftarrow f_{(x_j \neq t[x_j])}$;
        $f_{(x_j \neq t[x_j]),C} \leftarrow f_{(x_j \neq t[x_j]),C} \vee f_t$;
        $f_{x_j \neq t[x_j]} \leftarrow f_{x_j \neq t[x_j]} \wedge f_{x_j \neq t[x_j],C}$;
        if $mem' \neq f_{(x_j \neq t[x_j])}$ **then**
          Add $(x_j \neq t[x_j])$ to Q;
foreach $h_i \in \mathcal{H}$ **do**
  $D^{alt}(x_i) \leftarrow \emptyset$; **foreach** $v \in D_{x_i}$ **do**
    if $f_(x_i \neq v)[h_i]$ **then**
      Add v to $D^{alt}(x_i)$

- `Medium` : 148 variables, domain sizes from 2 to 20, 174 constraints of arity 2 to 10 (185Ko, 9.531 tuples).
- `Big` : 268 variables, domain sizes from 2 to 324, 332 constraints of arity 2 to 12 (3.57M, 225.982 tuples).

The protocol simulates 500 sessions of configurations as follows. First, a sample of 500 complete and consistent assignments is randomly built. For each of them, the corresponding session is simulated by assigning the variables following a random (uniform) order. After each assignment,

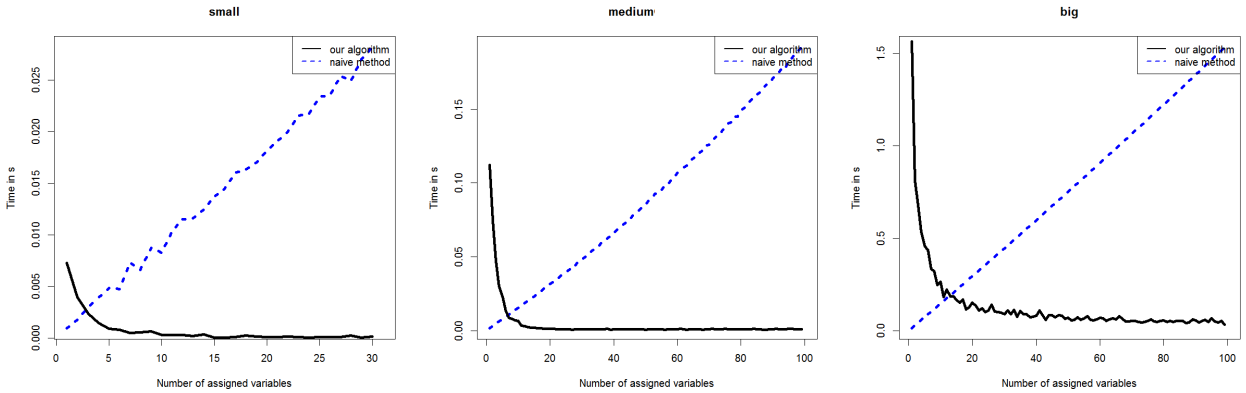

Figure 2: Computation time required by both the naive method and the justification-based one.

we measure the cpu time needed to make the current problem arc-consistent and to compute the alternative domains of all the assigned variables. The whole protocol is applied by both algorithms, which play on the same assignments with the same assignment orders. Figure 2 presents the result of these experiments on a HP Z600 workstation (Intel Xeon X5650 2.66Ghz processor, 16Gb Mb of RAM). On the x-axis is the number of the assignment in the sequence; the y-axis the mean cpu time needed for the naive method (dotted blue line) and for the justification-based algorithm dotted line (plain black line). We also measured the cpu used for a simple run of GAC4 (without maintaining any alternative domain) on the same protocol. For the first assignment, GAC4 -without the initialization step - takes a max of 0.003s (average 0.001s); for the next assignments, a max of 0.001s (average about 0.0005; the higher the number of past assignments, the lower the number of propagations).

The results are quite good: our algorithm is faster after the first 3 (resp. 5, resp. 15) variables have been assigned, i.e.when more than 3 (resp. 5, resp. 15) alternative domains are to be computed. Unsurprisingly, the first assignments are more costly, because of the initialisation of the structure - anyway, except for the first variable in the `Big` instance, the cpu is always lower than one second ; recall that `Big` is really big: its table constraints involve a total of 225.982 tuples. As expected, the time required by the naive algorithm grows linearly with the number of assigned variables, while our algorithm has a decreasing computation time.

5 Conclusion

In this work, we have coined the new concept of alternative domain of a variable and proposed an extension of GAC4 as a way to compute the alternative domains when maintaining General Arc Consistency on CSP with table constraints. Contrarily to the naive method that calls the propagation algorithm as many times as the number of alternative domains to be computed, our approach keeps limited justifications of the removals. Tested on industrial benchmarks, this method quickly outperforms the naive method.

The main limitation of our method is obviously its space consumption; the extra space consumption depends directly of the number of variables for which we want to compute the alternative domain. This being said, it should be kept in mind that for practical purposes the system is not asked to display all the alternative domains; the human user has a limited cognitive load and it is not obvious that she can or even wants to see a lot of alternative domains at a glance. In a configuration application for instance, the user looks at only a small number of variable simultaneously, typically the ones involved in the subcomponent currently being configured.

The concepts we have coined are close to the notion of value restoration. In the current work, we focused on the computation of alternative domains; an alternative value is a forbidden value that can be restored by the sole relaxation of the assignment of its variable. But more generally any value having at least one sufficient justification can be restored by the relaxation of only one assignment (if the variable is not assigned, by the relaxation of another one variable assignment). Hence the potential use of the algorithm proposed by this paper to provide the user with alternative values in a wider sense, and more generally to support the task of interactive relaxation by providing easily restorable values.

This work has a great potential for developments and perspectives. Firstly, our algorithm obviously needs to be improved, for instance with a lazy implementation, and our experiments must be completed. Secondly, we should think about the extension of the maintenance of alternative domains in CSP with general constraints, and not just in table constraints (actually, the GAC4-based approach does apply for intentional constraints, but would be too heavy: in the worst case, it would lead to generate the tables corresponding to the constraints). Finally, we should be able to consider the whole interaction; in this paper, we have considered only the assignment of a value to a variable: we need to study also the relaxation of these choices. This adaptation might mean an hybridizing with some algorithms of propagation/depropagation in dynamic CSP, e.g. the ones proposed by [Bessière, 1992; Debruyne, 1996; Debruyne *et al.*, 2003].

References

[Amilhastre *et al.*, 2002] Jérôme Amilhastre, Hélène Fargier, and Pierre Marquis. Consistency restoration and explanations in dynamic csps application to configuration. *Artificial Intelligence*, 135(1-2):199–234, 2002.

[Astesana *et al.*, 2013] Jean-Marc Astesana, Laurent Cosserat, and Hélène Fargier. Business recommendation for configurable products (BR4CP) project: the case study. `http://www.irit.fr/~Helene.Fargier/BR4CPBenchs.html`, March 2013.

[Bessière, 1992] Christian Bessière. Arc-consistency for non-binary dynamic csps. In *Proceedings of ECAI'92*, pages 23–27, 1992.

[Debruyne and Bessière, 1997] Romuald Debruyne and Christian Bessière. Some practicable filtering techniques for the constraint satisfaction problem. In *Proceedings of IJCAI'97*, pages 412–417, 1997.

[Debruyne *et al.*, 2003] Romuald Debruyne, Gérard Ferrand, Narendra Jussien, Willy Lesaint, Samir Ouis, and Alexandre Tessier. Correctness of constraint retraction algorithms. In *Proceedings of FLAIRS'03*, pages 172–176, 2003.

[Debruyne, 1996] Romuald Debruyne. Arc-consistency in dynamic csps is no more prohibitive. In *Proceedings of ICTAI'96*, pages 299–307, 1996.

[Debruyne, 2000] Romuald Debruyne. A property of path inverse consistency leading to an optimal pic algorithm. In *Proceedings of ECAI'2000*, pages 88–92, 2000.

[Fleischanderl *et al.*, 1998] Gerhard Fleischanderl, Gerhard Friedrich, Alois Haselböck, Herwig Schreiner, and Markus Stumptner. Configuring large systems using generative constraint satisfaction. *IEEE Intelligent Systems*, 13(4):59–68, 1998.

[Freuder and Elfe, 1996] Eugene C. Freuder and Charles D. Elfe. Neighborhood inverse consistency preprocessing. In *Proceedings of AAAI'96*, pages 202–208, 1996.

[Freuder, 1985] Eugene C. Freuder. A sufficient condition for backtrack-bounded search. *Journal of the ACM*, 32(4):755–761, 1985.

[Freuder, 1991] Eugene C. Freuder. Eliminating interchangeable values in constraint satisfaction problems. In *Proceedings of AAAI'91*, pages 227–233, 1991.

[Gelle and Weigel, 1996] Ester Gelle and Rainer Weigel. Interactive configuration using constraint satisfaction techniques. In *Proceedings of PACT-96*, pages 37–44, 1996.

[Hebrard *et al.*, 2004] Emmanuel Hebrard, Brahim Hnich, and Toby Walsh. Super solutions in constraint programming. In *Proceedings of CPAIO'04*, pages 157–172, 2004.

[Junker, 2006] Ulrich Junker. Configuration. In *Handbook of Constraint Programming*, pages 837–874. Elsevier Science, 2006.

[Jussien and Barichard, 2000] Narendra Jussien and Vincent Barichard. The palm system: explanation-based constraint programming. In *Proceedings of TRICS: Techniques foR Implementing Constraint programming Systems, a post-conference workshop of CP 2000*, pages 118–133, 2000.

[Jussien and Ouis, 2001] Narendra Jussien and Samir Ouis. User-friendly explanations for constraint programming. In *ICLP'01 Workshop on Logic Programming Environments*, 2001.

[Mailharro, 1998] Daniel Mailharro. A classification and constraint-based framework for configuration. *Artificial Intelligence for Engineering Design, Analysis and Manufacturing*, 12:383–397, September 1998.

[Mittal and Falkenhainer, 1990] Sanjay Mittal and B. Falkenhainer. Dynamic constraint satisfaction problems. In *Proceedings of AAAI'90*, pages 25–32, 1990.

[Mittal and Frayman, 1989] Sanjay Mittal and Felix Frayman. Towards a generic model of configuraton tasks. In *Proceedings of the IJCAI'89*, pages 1395–1401, 1989.

[Mohr and Masini, 1988] Roger Mohr and Gérald Masini. Good old discrete relaxation. In *Proceedings of the ECAI'88*, pages 651–656, 1988.

[Sabin and Freuder, 1996] Daniel Sabin and Eugene C. Freuder. Configuration as composite constraint satisfaction. In *AI and Manufacturing Research Planning Workshop*, pages 153–161, 1996.

[Sabin and Weigel, 1998] Daniel Sabin and Rainer Weigel. Product configuration frameworks — a survey. *IEEE Intelligent Systems*, 13(4):42–49, 1998.

[Stumptner *et al.*, 1998] Markus Stumptner, Gerhard E. Friedrich, and Alois Haselböck. Generative constraint-based configuration of large technical systems. *AI EDAM*, 12(04):307–320, 1998.

[Weigel and Bliek, 1998] Rainer Weigel and Christian Bliek. On reformulation of constraint satisfaction problems. In *Proceedings of ECAI'98*, pages 254–258, 1998.