Maintaining alternative values in constraint-based configuration

Caroline Becker and **Hélène Fargier** [1]

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

In the present work, we aim at offering a more convenient interaction by providing the user with possible alternative values for each of the already assigned variables - i.e. the values that could replace the current one without leading to the violation of some constraint. 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 with the one of the naive algorithm that would run a full propagation for each variable, but its experimental efficiency is much 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).

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 [14, 18, 12, 19, 20] 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 configuration 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 [13], for instance suit the problems where the existence of some optional variables depends on the value of another variable. Other extensions proposed by the CSP community include composite CSPs [17], interactive CSPs [10], hypothesis CSPs [1], generative constraint satisfaction [19, 7], etc.

In this article, 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 variables must be pruned so as to ensure that the values available for the further variables 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 the maintaining of 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 one 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 develop the basis of our algorithm. Our first experimental results are shown in Section 4. Proofs are gathered in Appendix.

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

[1] IRIT, University Toulouse III, France, email: {becker,fargier}@irit.fr

$\mathcal{C}$. 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 this variable. Following [1], the problem can be represented by an Assumption-based CSP (A-CSP).

Definition 1 (A-CSP) *An* A-CSP *is a 4-uple* $(\mathcal{X}, \mathcal{D}, \mathcal{C}, \mathcal{H})$ *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 set of current user choices, i.e. assignments of the variables: *we suppose in the sequel of the paper that all the restrictions in* $\mathcal{H}$ *bear with different variables and restrict their domain 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 original one (i.e. having the same set of solution) and that is arc consistent; this problem is called the closure by arc consistency of the original one.

Other, more powerful, levels of local consistency can be ensured, e.g. Path Inverse Consistency [4], Singleton Arc Consistency [5], k-inverse consistency [8, 9]. 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 l of local consistency - i.e. that computes the closure by l 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 l be a level of local consistency and $P = (\mathcal{X}, \mathcal{D}, \mathcal{C}, \mathcal{H})$ an A-CSP. The* current domain *according to l of a variable x_i is its domain in the closure by l-consistency of* $(\mathcal{X}, \mathcal{D}, \mathcal{C} \cup \mathcal{H})$.

[2] Actually, the definitions and results could be set in a more general framework and capture any type of restriction; the meaning of alternative value when the restrictions in $\mathcal{H}$ are not unary is nevertheless questionable, hence our assumption.

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 l 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 l is its domain in the closure by l-consistency of the CSP*$(\mathcal{X}, \mathcal{D}, \mathcal{C} \cup \mathcal{H} \setminus \{h_i\})$. *We write it* $D^l_{alt}(x_i)$.

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 another value than v and (ii) the single relaxation of this assignment would make v l-consistent. For instance, if x_i is the last assigned variable, all the values that were in the domain of x_i just before its assignment are alternative values.

Example 1 Consider the CSP $\mathcal{X} = \{x_1, x_2, x_3\}$, $\mathcal{D} = D_1 \times D_2 \times D_3 = \{1, 2, 3, 4\}^3$, $\mathcal{C} = \{Alldiff(x_1, x_2, x_3)\}$; initially, $\mathcal{H} = \emptyset$ and the current domains of the three variables are $D_C(x_1) = D_C(x_2) = D_C(x_3) = \{1, 2, 3, 4\}$. In this example, we suppose that that arc consistency is maintained.
Let theuser first assign value 1 to x_1. We get $\mathcal{H} = \{(x_1 = 1)\}$; then $D_C(x_1) = \{1\}$ and arc consistency removes value 1 from the current domains of x_2 and x_3: $D_C(x_2) = D_C(x_3) = \{2, 3, 4\}$. At this step, x_1 is the only assigned variable and has three alternative values, 2, 3 and 4.
Suppose that the user then assigns value 4 to x_2, i.e. $\mathcal{H} = \{(x_1 = 1), (x_2 = 4)\}$; arc consistency, removes 4 from the current domains of x_2 and x_3: $D_C(x_2) = D_C(x_3) = \{2, 3\}$ while $D_C(x_1) = \{1\}$ and $D_C(x_2) = \{4\}$. x_1 has only two alternative values left : 2 and 3; 4 is not alternative anymore since it does not belong to the closure by arc consistency of the CSP $< \mathcal{X} = \{x_1, x_2, x_3\}, \mathcal{D} = D_1 \times D_2 \times D_3 = \{1, 2, 3, 4\}^3, \mathcal{C} = \{Alldiff(x_1, x_2, x_3) \cup \{x_2 = 4\}\} >$. x_2 has also two alternative values, 2 and 3 (see Table 1).

	1	2	3	4
x_1	⋆	◇	◇	×
x_2	×	◇	◇	⋆
x_3	×	◇	◇	×

Table 1. Assigned (⋆), forbidden (×) and alternative (◇) values for the A-CSP $\mathcal{X} = \{x_1, x_2, x_3\}$, $\mathcal{D} = D_1 \times D_2 \times D_3 = \{1, 2, 3, 4\}^3$, $\mathcal{C} = \{Alldiff(x_1, x_2, x_3)\}$, $\mathcal{H} = \{(x_1 \leftarrow 1), (x_2 \leftarrow 4)\}$).

The notion of alternative domain is orthogonal to the notion of removal's explanation, such as proposed in PaLM [16]: explanations are a way to explain the pruning of the domains and aim at proposing a strategy of restoration of some value for an 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 the other user choices.

The notion of alternative domain can be compared to the concept of fault tolerant solution [21]. A fault tolerant solution is actually a solution such as all the variables have a non-empty alternative domain: if one of the current value 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. The notion has been generalized by Hebrard et al. [11] under the name "super-solutions". The main difference between the notion of fault tolerant solutions and the notion of alternative domains is that fault tolerant solutions deal with *complete* assignments while alternative domains suggests restoration values for *partial* assignments also. It should also be noticed that the two notions target different practical goals: when refereing to a super-solution, the one in looking for *some*, but not *all*, robust (and complete) solutions - there is indeed a potentially exponential number of fault tolerant 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 the assignment of variable x_i. Each P_i is filtered by l-consistency. The alternative domain of variable x_i is 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.

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^l the closure of $(\mathcal{X}, \mathcal{D}, \mathcal{C} \cup \mathcal{H})$ by some level of local consistency l.*

A removal w.r.t. a level l of local consistency 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^l
We write $\mathcal{R}^l$ the set of removals of P w.r.t. l.

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

To improve readability, a removal (x_j, v) will often be written $(x_j \neq v)$, and we will omit to mention level l 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, l a level of local consistency, and $\mathcal{R}^l$ the set of P's removal according to l.*

An user choice $h_i \in \mathcal{H}$ is said to be an l-sufficient justification of a removal $(x_j \neq v) \in \mathcal{R}^l$ if and only if v belongs to the domain of x_j in the l-consistent closure of $(\mathcal{X}, \mathcal{D}, \mathcal{C} \cup \mathcal{H} \setminus \{h_i\})$.

By extension, for any x_j in $\mathcal{X}$ and any v in $D(x_j)$, $h_i \in \mathcal{H}$ is said to be an l-sufficient justification of v for x_j if and only if v belongs to the domain of x_j in the l-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 belongs to the current domain of its variable.

Example 1 (cont') If we go back to example 1, once x_1 and x_2 are assigned, $\mathcal{H}$ contains two assumptions: $h_1 = (x_1 = 1)$, and $h_2 = (x_2 = 4)$.
All the values deleted from the domain of x_1 (resp. x_2), have h_1 (resp. h_2) as a (sole) sufficient justification.
Arc consistency has removed values 1 and 4 from the domains of x_2 and x_3. h_1 is a sufficent justification for the removals $(x_2 \neq 1)$ and $(x_3 \neq 1)$, and h_2 a sufficient justification of $(x_2 \neq 4)$ and $(x_3 \neq 4)$.
By convention, all the values that are still in the current domains of their variables receive both h_1 and $h2$ as a sufficient justifications.

Example 2 A removal may have several sufficient justifications, as shown by the following example. Consider the CSP $\mathcal{X} = \{x_1, x_2, x_3, x_4\}$, $\mathcal{D} = D_1 \times D_2 \times D_3 \times D_4 = \{1, 2, 3\}^4$, $\mathcal{C} = \{x_1 \neq x_2, x_3 \neq x_2, x_4 \neq x_2)\}$. Value 2 for x_1 has two supports on x_2 : 1 and 3. Suppose that the user has assigned value 1 to x_3 (h_3) and value 3 to x_4 (h_4); in other terms, $\mathcal{H} = \{(x_3 = 1), (x_4 = 3)\}$. h_3 forbidds the first support of $x_1 = 2$ and h_4 forbids its second support ; value 2 is thus removed by arc consistency from the current domain of x_1: $D_C(x_1) = \{1, 3\}$ and this removal has two sufficient justifications: h_3 and h_4.

Of course, a value belongs v to the alternative domain of an assigned variable x_i iff h_i is a sufficient justification of the removal $(x_i \neq v)$:

Proposition 6 *Let $P = (\mathcal{X}, \mathcal{D}, \mathcal{C}, \mathcal{H})$ be an A-CSP, l 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 l consistency of $P = (\mathcal{X}, \mathcal{D}, \mathcal{C} \cup \mathcal{H})$ or $(x_i \neq v) \in \mathcal{R}^l$ and h_i is a sufficient justification of $(x_i \neq v)$.

The notion of sufficient justification is extended to tuples as follows:

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

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

Definition 7 (Sufficient justification of a tuple)
Let $P = (\mathcal{X}, \mathcal{D}, \mathcal{C}, \mathcal{H})$ be an A-CSP, l a level of local consistency, C a constraint in $\mathcal{C}$ and t an assignment of $vars(C)$ satisfying C.

An user choice $h_i \in \mathcal{H}$ is said to be an l- 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 l consistency of the CSP $(\mathcal{X}, \mathcal{D}, \mathcal{C} \cup \mathcal{H} \setminus \{h_i\})$.

Example 1 (cont') If we go back to example 1, once x_1 and x_2 have been assigned, tupple $(3, 2, 4)$ is not valid anymore and has one sufficient justification, h_1 (it is enough to relax $x_1 = 1$ to make this tupple valid again); remark that tupple $(4, 2, 1)$, that is also invalid, has no sufficient justification (the relaxation of the two choices is necessary to make it valid again).

Our algorithm is based on the fact that an assignment h_i is an l-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 8 (Conflict set)
The conflict set of a tuple t w.r.t. some level of l consistency is the subset of $\mathcal{R}^l$ defined by: $CS(t) = \{(x_i \neq v) \in \mathcal{R}^l$ s. t. $t[x_i] = v\}$.

Of course, a tuple is invalid if and only if it has a non-empty conflict set.

Proposition 9 *h_i is an l-sufficient justification of a tuple t if and only it is an l-sufficient justification of each of the removals in its conflict set w.r.t. l.*

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

Proposition 10 *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 GAC-sufficient justification of t.*

Similar properties can be established for other levels of local consistency based on the notion of support, typically for k inverse consistency [8][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 11

$h_i \in \mathcal{H}$ 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 of maintenance of the alternative domains w.r.t. Arc Consistency

In our application, interactive configuration, the constraint to be taken into account are mostly table constraints and the level of consistency referred to is Generalized Arc Consistency. We thus propose to maintain the alternative domain upon the assignment of a variable using an extension of GAC4 [15]. 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 10, $f_{(x \neq v)}$ 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 9). 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 12

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

where $Support(x, v, C)$ is the set of assignments of $vars(C)$ that support (x, v).

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 arc consistent;
- for any constraint $C \in \mathcal{C}$, $Table(c)$ is the set of assignments of $vars(c)$ that satisfy it. We 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.
- for any tuple t, f_t is its vector of justifications; for any removal $(x_i \neq v)$ $f_{(x_i \neq v)}$ is its vector of justifications; 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 non only when it is created, but for each change in its vector of justifications. Since the updating of the vectors of justification is monotonic (a 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, each removal can enter in the queue n times at most (n being the number of h_i in $\mathcal{H}$), i.e.as much 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 $O(n.T)$.

```
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
         $Cpt(x_i, v, C) := 0$;
         $S_{i,v,C} = \emptyset$
      end
      foreach $t \in Table(C)$ do
         $f_t = \text{True}^n$;
         $valid(t) = \text{True}$;
         $CS(t) = \text{False}^n$;
         foreach $x_i \in vars(C)$ do
            $Cpt(x_i, t[x_i], C) + +$;
            Add $t$ to $S_{i,t[x_i],C}$
         end
      end
   end
end
```
Algorithm 1: Initialization

```
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 $\mathcal{H}$;
$Q := \emptyset$;
/* 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
   $f_{(x_k \neq u)} \leftarrow \text{False}^m$;
   $f_{(x_k \neq u)}[h_k] \leftarrow \text{True}$;
end
Add $(x_k \neq u)$ to $Q$;
while $Q \neq \emptyset$ do
   Choose and remove a $(x_i \neq v)$ from $Q$;
   if $v \in D_c(x_i)$ then
      Remove $v$ from $D_c(x_i)$;
   end
   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 */ ;
               end
            end
         end
         $valid(t) = false$;
      end
      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
            $mem' = f_{(x_j \neq t[x_j])}$;
            $f_{(x_j \neq t[x_j]),C} = f_{(x_j \neq t[x_j]),C} \vee f_t$;
            $f_{x_j \neq t[x_j]} = f_{x_j \neq t[x_j]} \wedge f_{x_j \neq t[x_j],C}$;
            if $mem' \neq f_{(x_j \neq t[x_j])}$ then
               Add $(x_j \neq t[x_j])$ to $Q$;
            end
         end
      end
   end
end
foreach $h_i \in \mathcal{H}$ do
   $D^{alt}(x_i) = \emptyset$; foreach $v \in D_{x_i}$ do
      if $f_(x_i \neq v)[h_i]$ then
         Add $v$ to $D^{alt}(x_i)$
      end
   end
end
```
Algorithm 2: Propagation of decision $h_k = (x_k \leftarrow w)$

4 First experimental results

We have tested this algorithm on an industrial problem of configuration. It involves 32 variables of domain of size 2 to 10, 35 binary constraints. The product to configure is a blowing machine, which blows bottles for different matters. The benchark can be found at url `ftp://ftp.irit.fr/pub/IRIT/ADRIA/PapersFargier/Config/souffleuse.xml`.

The protocol simulates 1000 sessions of configurations as follows. First, a sample of 1000 consistent complete assignments is randomly fired. For each of then, the corresponding session is simulated by assigning the variables following a random (uniform) order. After each assignment, we measure the cpu time needed to make the current problem arc-consistent and to compute the alternative domains of all the already assigned variables are computed. The whole protocol is applied by both the justification-based algorithm described in the previous Section and the naive method (that works on as may copies of the original CSP as the number of user choices in $\mathcal{H}$, as decribed in introduction of Section 3) ; for the shake of rigor, the two algorithms play on the same assignments and the same assignment orders.

Figure 1 presents the result of these experiments. On the x-axis is the number of the assignment in the sequence; the y-axis is logarithmic and indicates the mean cpu time need for the naive method (plain line, with rounds) and for the justification-based algorithm dotted line, with squares.

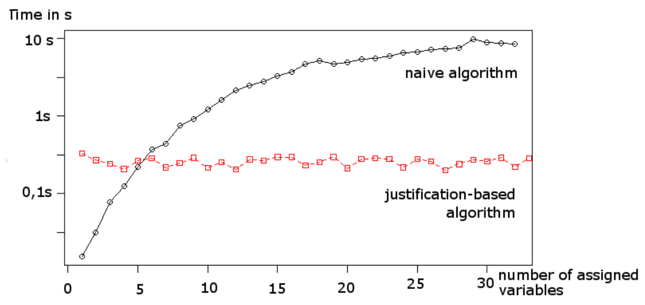

Figure 1. Computation time required by both the naive method and the justification-based algorithm (logarithmic scale).

The results are quite good: our algorithm is faster as soon as more than 5 variables are assigned, i.e.when more than 5 alternative domains are to be computed. As expected, the time required by the naive algorithm grows linearly with the number of variables, while our algorithm has stable computation time. These first results have obviously to be confirmed by more experiments on bigger configuration problems.

5 Conclusion

In this work, we have coined the new concept of the alternative domain of a variable with respect to a local consistency level and proposed an extension of GAC4 algorithm as a way to compute the alternative domains when maintaining General Arc Consistency on problems involving table constraints.

Contrarily to the naive method that applies 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 two 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 with a limited mental capacity 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. For each value in the domain of an assigned variable, the user knows whether she can change her choice to this value without modifying the other choices - this is the notion alternative domain. But the user also knows something about the values that have been filtered from the domains of the *unassigned* variables: the justification vector of such a value provides her with the set of , previous choices (on *other* variables) she could relax in order to make the value available again. 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 huge 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 domain in CSP with general constraints, and not just in table constraints ; such an algorithm is not too difficult to conceive for CSPs involving binary constraints only, but the task seems much more tricky for general constraints. Finally, we should be able to consider the whole interaction; for the moment, we only considered the assignment of a value to a variable: we need to study the relaxation of choices also. This adaptation might mean an hybridizing with the maintenance algorithms of propagation/depropagation in dynamic CSP[2, 3, 6].

ACKNOWLEDGEMENTS

This work is partially funded by the ANR project BR4CP (ANR-11-BS02-008)

REFERENCES

[1] 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).

[2] Christian Bessière, 'Arc-consistency for non-binary dynamic csps', in *Proceedings of ECAI'92*, pp. 23–27, (1992).

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

[4] Romuald Debruyne, 'A property of path inverse consistency leading to an optimal pic algorithm', in *Proceedings of ECAI'2000*, pp. 88–92, (2000).
[5] Romuald Debruyne and Christian Bessière, 'Some practicable filtering techniques for the constraint satisfaction problem', in *Proceedings of IJCAI'97*, pp. 412–417, (1997).
[6] Romuald Debruyne, Gérard Ferrand, Narendra Jussien, Willy Lesaint, Samir Ouis, and Alexandre Tessier, 'Correctness of constraint retraction algorithms', in *Proceedings of FLAIRS'03*, pp. 172–176, (2003).
[7] 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).
[8] Eugene C. Freuder, 'A sufficient condition for backtrack-bounded search', *Journal of the ACM*, **32**(4), 755–761, (1985).
[9] Eugene C. Freuder and Charles D. Elfe, 'Neighborhood inverse consistency preprocessing', in *Proceedings of AAAI'96*, pp. 202–208, (1996).
[10] Ester Gelle and Rainer Weigel, 'Interactive configuration using constraint satisfaction techniques', in *Proceedings of PACT-96*, pp. 37–44, (1996).
[11] Emmanuel Hebrard, Brahim Hnich, and Toby Walsh, 'Super solutions in constraint programming', in *Proceedings of CPAIO'04*, pp. 157–172, (2004).
[12] Daniel Mailharro, 'A classification and constraint-based framework for configuration', *Artificial Intelligence for Engineering Design, Analysis and Manufacturing*, **12**, 383–397, (September 1998).
[13] Sanjay Mittal and B. Falkenhainer, 'Dynamic constraint satisfaction problems', in *Proceedings of AAAI'90*, pp. 25–32, (1990).
[14] Sanjay Mittal and Felix Frayman, 'Towards a generic model of configuraton tasks', in *Proceedings of the IJCAI'89*, pp. 1395–1401, (1989).
[15] Roger Mohr and Gérald Masini, 'Good old discrete relaxation', in *Proceedings of the ECAI'88*, pp. 651–656, (1988).
[16] Samir Ouis, Narendra Jussien, and Olivier Lhomme, 'Explications conviviales pour la programmation par contraintes', in *Actes de JFPLC*, pp. 105–118, (2002).
[17] Daniel Sabin and Eugene C. Freuder, 'Configuration as composite constraint satisfaction', in *AI and Manufacturing Research Planning Workshop*, pp. 153–161, (1996).
[18] Daniel Sabin and Rainer Weigel, 'Product configuration frameworks — a survey', *IEEE Intelligent Systems*, **13**(4), 42–49, (1998).
[19] Markus Stumptner, Gerhard E. Friedrich, and Alois Haselböck, 'Generative constraint-based configuration of large technical systems', *AI EDAM*, **12**(04), 307–320, (1998).
[20] Junker Ulrich, 'Configuration', in *Handbook of Constraint Programming*, 837–874, Elsevier Science, (2006).
[21] Rainer Weigel and Christian Bliek, 'On reformulation of constraint satisfaction problems', in *Proceedings of ECAI'98*, pp. 254–258, (1998).

A Proofs

[Proof of Proposition 9]
Of course, the proposition holds when t is valid (it has an empty conflict set). Let us examine the case of an invalid tuple.

$\boxed{\Rightarrow}$ Let h_i be l-sufficient justification of an invalid tuple t and suppose that there exists a removal $(x \neq v)$ in the conflict set of t such that h_i is not a sufficient justification of $(x \neq v)$.
We write P_i^l the l-consistent closure of $(\mathcal{X}, \mathcal{D}, \mathcal{C} \cup \mathcal{H} \setminus \{h_i\})$. Since h_i is an l-sufficient justification of t, by definition, t is valid in P_i^l. Since h_i is also not a sufficient justification of $(x \neq v)$, v is not in the domain of x in P_i^l; t is thus invalid in P_i^l, which is a contradiction.

$\boxed{\Leftarrow}$ Reciprocally, let h_i be an l-sufficient justification of all the removals in the conflict set of t. For each of these $(x_j \neq v_j)$, v_j is by definition in the domain of x_j in P_i^l. Thus, t is a valid tuple in P_i^l - by definition of the notion of justification, h_i is thus an l-sufficient justification of t. □

[Proof of Proposition 10]
$\boxed{\Rightarrow}$ Let h_i be a GAC sufficient justification of a removal $(x \neq v)$. Suppose that there exits a constraint C bearing on x such that none of the supports of $x = v$ on C admits h_i as a sufficient justification. This means that these tuples are not valid in the arc consistent closure of $(\mathcal{X}, \mathcal{D}, \mathcal{C} \cup \mathcal{H} \setminus \{h_i\})$ (denoted P_i^{GAC}). Thus v has no support on C in P_i^{GAC}: it does not belongs to the domain of x in P_i^{GAC}; h_i is thus not a sufficient justification of $(x \neq v)$.

$\boxed{\Leftarrow}$ Reciprocally, consider an assumption h_i and suppose that $\forall C$ bearing x, $\exists t$ support of $x = v$ such that h_i is a GAC sufficient justification of t . This means that, for any constraint bearing on x there exists a support t of $x = v$ valid in P_i^{GAC}; v thus belongs to the domain of x in P_i^{GAC} - by definition, this meant that h_i is a GAC-sufficient justification of $(x \neq v)$. □

[Proof of Proposition 11]
$\forall(x_1, ..., x_k), \exists(v_1, ..., v_k)$ a support of $x = v$ such that h_i is a $(1, k)$-sufficient justification of $(v_1, ..., v_k)$

$\Leftrightarrow \forall(x_1, ..., x_k), \exists(v_1, ..., v_k)$ support of $x = v$ such that any of the v_j belongs to the domain of its variable in the closure by $(1, k)$-consistency of $(\mathcal{X}, \mathcal{D}, \mathcal{C} \cup \mathcal{H} \setminus \{h_i\})$

$\Leftrightarrow$ v belongs to the domain of x the closure by $(1, k)$-consistency of $(\mathcal{X}, \mathcal{D}, \mathcal{C} \cup \mathcal{H} \setminus \{h_i\})$ (definition of the $1, k$ consistency)
$\Leftrightarrow$ h_i is a justification $(1, k)$-sufficient of $x \neq v$. □

[Proof of Proposition 12]
According to Proposition 10, when GAC is ensured

$$f_{(x \neq v)}[h_i] = \bigwedge_{C, x \in vars(C)} \bigvee_{t \in Support(x,v,C)} f_t[h_i]$$

For any h_i, any $x \in \mathcal{X}$ and any $v \in D_x$. I.e.:

$$f_{(x \neq v)} = \bigwedge_{C, x = vars(C)} \bigvee_{t \in Support(x,v,C)} f_t$$

Yet, according to Proposition 9, the GAC-sufficient justifications set of a tuple is the intersection of GAC-sufficient justifications of the removals from its conflict set: $f_t = \bigwedge_{r \in CS(t)} f_r$. Hence the result. □