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 [13, 17, 11, 18, 19] 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 [12], 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 [16], interactive CSPs [10], hypothesis CSPs [1], generative constraint satisfaction [18, 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
We note $D = \prod_{i=1}^{n} D(x_i)$. An assignment $t$ of a set of variable $X \subseteq \mathcal{X}$ is an element of the cartesian product of the domains of these variables; for any $x_i \in X$ we denote by $t[x_i]$ the value assigned to $x_i$ in $t$.

A constraint $C$ in $\mathcal{C}$ involves a set $\text{vars}(C) \subseteq \mathcal{X}$ and can be viewed as a function from the set of assignments of $\text{vars}(C)$ to $\{\top, \bot\}$: $C(t) = \top$ iff $t$ satisfies the constraint; for any $x_i$ in $\text{vars}(C)$ and any $v$ in its domain, we say that an assignment $t$ of $\text{vars}(C)$ 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 $x_i \in \mathcal{X}$ and any value $v$ in its domain, for any constraint $C$ bearing on $x_i$, 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 $(\mathcal{X}, \mathcal{D}, \mathcal{C} \cup \mathcal{H})$.

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 $D_{alt}(x_i)$.

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 $\mathcal{X} = \{x_1, x_2, x_3\}$, $D = D_1 \times D_2 \times D_3 = \{1, 2, 3, 4\}^3$, $\mathcal{C} = \{\text{AllDiff}(x_1,x_2,x_3)\}$; 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 user first assign value 1 to $x_1$. We get $\mathcal{H} = \{(x_1 = 1)\}$; then $D_C(x_1) = \{1\}$ and arc consistency removes the 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 now two alternative values only: 2 and 3; 4 is not alternative anymore since it does not belong to the closure by arc consistency of the CSP $\mathcal{X} = \{x_1, x_2, x_3\}$, $D = D_1 \times D_2 \times D_3 = \{1, 2, 3, 4\}^3$, $\mathcal{C} = \{\text{AllDiff}(x_1, x_2, x_3)\}$, $\mathcal{H} = \{(x_1 \leftarrow 1), (x_2 \leftarrow 4)\}$.

<table>
<thead>
<tr>
<th>$x_1$</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
</tr>
</thead>
<tbody>
<tr>
<td>$x_2$</td>
<td>$\times$</td>
<td>$\times$</td>
<td>$\times$</td>
<td>$\times$</td>
</tr>
<tr>
<td>$x_3$</td>
<td>$\times$</td>
<td>$\times$</td>
<td>$\times$</td>
<td>$\times$</td>
</tr>
</tbody>
</table>

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

The notion of alternative domain is orthogonal to the notion of removal’s explanation, such as proposed in PaLM [15]: 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.

---

\(^2\) 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.
The notion of alternative domain can be compared to the concept of fault tolerant solution [20]. 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. under the name "super-solutions". The main difference fault tolerant solutions and alternative domains is that fault tolerant solutions deal with complete assignments while alternative domains suggest restoration values even for partial assignments. 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 P0 (where all the n variables are assigned), and n CSP Pi where each Pi has exactly the same assignments than P0, with the exception of the assignment of variable xi. Each Pi is filtered by l-consistency. The alternative domain of variable xi is obviously the domain of xi in the arc consistent closure of Pi. 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 hi ∈ H. The flag on hi must be true if and only if the single relaxation of the user’s choice hi will lead to have the value back in the domain of its variable. Let us formalize:

Definition 4 (Removal, invalid tuple)

Let P = (X, D, C, H) be an A-CSP and PΩ the closure of (X, D, C ∪ H) by some level of local consistency l.

A removal w.r.t. a level l of local consistency is a pair (xj, v), xj ∈ X, v ∈ D(xj) such that v does not belong to the domain of xj in PΩ.

We write RΩ the set of removals of P w.r.t. l.

Let C a constraint in C and t an assignment of vars(C) satisfying C. t is said to be invalid w.r.t. l iff there exists xj ∈ vars(C) such that t(xj) does not belong to domain of xj in PΩ; otherwise, it is said to be valid w.r.t. l.

To improve readability, a removal (xj, v) will often be written (xj, 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 = (X, D, C, H) be an A-CSP, l a level of local consistency, and RΩ the set of P’s removal according to l.

An user choice hi ∈ H is said to be an l-sufficient justification of a removal (xj, v) ∈ RΩ if and only if v belongs to the domain of xj in the l-consistent closure of (X, D, C ∪ H \ {hi}).

By extension, for any xj in X and any v in D(xj), hi ∈ H is said to be an l-sufficient justification of v for xj if and only if v belongs to the domain of xj in the l-consistent closure of (X, D, C ∪ H \ {hi}).

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 ̸= v. By extension, any hi is a sufficient justification of a value that does belong to the current domain of its variable.

Example 1 (cont’)

If we go back to example 1, once x2 and x3 are assigned, H contains two assumptions: h1 = (x1 = 1) and h2 = (x2 = 4). All the values deleted from the domain of x1 (resp. x2), have h1 (resp. h2) as a (sole) sufficient justification. Arc consistency has removed values 1 and 4 from the domains of x2 and x3, h1 is a sufficient justification for the removals (x2 ̸= 1) and (x3 ̸= 1), and h2 a sufficient justification of (x2 ̸= 4) and (x3 ̸= 4).

By convention, all the values that are still in the current domain of their variables receive both h1 and h2 as a sufficient justifications.

Example 2 A removal may have several justifications, as shown by the following example. Consider the CSP X = {x1, x2, x3, x4}, D = D1 × D2 × D3 × D4 = {1, 2, 3}4, C = {x1 ̸= x2, x3 ̸= x2, x4 ̸= x2}. Value 2 for x1 has two supports on x2 : 1 and 3. Suppose that the user has assigned value 1 to x3 (h3) and value 3 to x4 (h4). h3 forbids the first support of x1 = 2 and h4 forbids its second support ; value 2 is thus removed by arc consistency from the current domain of x1; DC(x1) = {1, 3} and this removal has two sufficient justifications: h3 and h4.

Of course, a value belongs v to the alternative domain of an assigned variable xi if hi is a sufficient justification of the removal (xj, v):

Proposition 6 Let P = (X, D, C, H) be an A-CSP, l a level of local consistency.

For any xj ∈ X, any v ∈ D(xj), v belongs to the alternative domain of xj, if either v belongs to the domain of xj, in the closure by l consistency of P = (X, D, C ∪ H) or (xj, v) ∈ RΩ and hi is a sufficient justification of (xj, v).

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

Definition 7 (Sufficient justification of a tuple)

Let P = (X, D, C, H) be an A-CSP, l a level of local consistency, C a constraint in C and t an assignment of vars(C) satisfying C.
A CSP is (1, k)-sufficient justification of (x ≠ v) ∈ R^{(1,k)} iff, for each set V of k variables there exists a support t of (x = v) on V such as h, is a (1,k)-sufficient justification of t.

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

### 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 [14]. Our algorithm propagates not only value removals, but also justifications: for each removal (x, v), we maintain a vector \( f(x,v) \) of n boolean flags, one for each choice in \( \mathcal{H} \), such that \( f(x\neq v)(v) = \text{True} \) if and only if \( h \) is a sufficient justification of (x, 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 \( h \), \( f_t(h_1) = \text{true} \) iff \( h_1 \) is a sufficient justification of \( t \). Intuitively (see Table 2 for an example), for the user choice \( h \), to be a sufficient justification for a removal (x ≠ v) provoked by constraint \( C \), it is needed that the relaxation of \( h \) 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 \( h \) as a sufficient justification (this is the meaning of Proposition 9). In other words, \( f_t \) is the intersection of the \( f(x\neq v) \) flags of all the removals (x ≠ w) in the conflict set of t. Formally:

**Proposition 12**

\[
\text{Proposition 12.} \quad f(x\neq v) = \bigwedge_{C|\in \text{vars}(C)} \big( \bigvee_{t \in \text{Support}(x,v,C)} \big( \bigwedge_{r \in \text{CS}(t)} f_r \big) \big) 
\]

where \( \text{Support}(x,v,C) \) is the set of assignments of \( \text{vars}(C) \) 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:

- \( (X, D, C) \) is the original CSP, that is supposed arc consistent;
- for any constraint \( C \in \mathcal{C} \), Table(c) is the set of assignments of \( \text{vars}(C) \) 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.
- \( D(x_i) \) is the current domain of \( x_i \)
- \( S_{x_i,v,C} \) is the set of supports of \( (x_i,v) \) on \( C \) and \( \text{Cpt}(x_i,v,C) \) 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, 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 \(H\)), 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, \(n\) 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 GAC4-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 \(x_i \in \mathcal{X}(C), v \in D(x_i)\) do
    \(Cp(x_i, v, C) := 0;\)
    \(S_{i,v,C} = \emptyset;\)
  endforeach
  foreach \(t \in Table(C)\) do
    \(f_t = True^\wedge;\)
    valid\(t\) = True\(^\wedge;\)
    \(CS(t) = False^\wedge;\)
    foreach \(x_i \in \mathcal{X}(C)\) do
      \(Cp(x_i, t[x_i], C) += +;\)
      Add \(t\) to \(S_{t,x_i,C}\)
    endforeach
  endforeach
end

Algorithm 1: Initialization

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

Procedure Propagate( \(x_h, w\): assumption; \(\mathcal{X}, \mathcal{D}, \mathcal{C}\): the initial CSP; \(H\): the past assumptions; \(D_i\): the current domains)
Add \((x_h, w)\) to \(H\);
\(Q := \emptyset;\)
/* The removal of the other values in the current domain of \(x_h\) is due to \(h_k\) */
foreach \(u \neq w \in D(x_h)\) do
  \(f(x_h \neq u) \leftarrow True\)
  \(f(x_h \neq u)[h_k] \leftarrow False\)
end
Add \((x_h \neq u)\) to \(Q\);
while \(Q \neq \emptyset\) do
  Choose and remove a \((x, \neq v)\) from \(Q\);
  if \(v \in D_i(x_i)\) then
    Remove \(v\) from \(D_i(x_i)\);
  end
  foreach \(C\) s.t. \(x_i \in \mathcal{X}(C)\) 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 \(x_j \in \mathcal{X}(t)\) s.t. \(j \neq i\) do
        if \(Cp(x_j, t[x_j], C) == 0\) then
          \(f(x_j \neq t[x_j]) \leftarrow False^\wedge;\)
          \(CS(t) \leftarrow False^\wedge;\)
        else
          \(f(x_j \neq t[x_j]) \leftarrow True^\wedge;\)
        end
      endforeach
    end
  endforeach
  valid\(t\) = False;
if \(Mem! = f_t\) /* A justif. of \(t\) is not sufficient anymore */ then
  foreach \(x_j \in \mathcal{X}(t)\) s.t. \(j \neq i\) do
    \(mem! = f(x_j \neq t[x_j]);\)
    \(f(x_j \neq t[x_j]) = f(x_j \neq t[x_j]) \wedge f_t;\)
    \(f(x_j \neq t[x_j]) = f(x_j \neq t[x_j]) \wedge f_t;\)
    if \(mem! \neq f(x_j \neq t[x_j])\) then
      Add \((x_j \neq t[x_j])\) to \(Q\);
    end
  endforeach
end
endif

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

The protocol simulates 1000 sessions of configurations as follows. First, a sample of 1000 consistent complete assignments is randomly fired. For each of them, 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 algorithm that works on as may copies of the original CSP as the number of user choices in $H$, as described in introduction of Section 3); for the sake 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 needed 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).](image)

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

Let $h_i$ be an $l$-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 $P^g_i$ the $l$-consistent closure of $(X, D, C \cup H \setminus \{h_i\})$. Since $h_i$ is an $l$-sufficient justification of $t$, by definition, $t$ is valid in $P^g_i$. Since $h_i$ is also not a sufficient justification of $(x \neq v)$, $v$ is not in the domain of $x$ in $P^g_i$; $t$ is thus invalid in $P^g_i$, which is a contradiction.

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 $P^g_i$. Thus, $t$ is a valid tuple in $P^g_i$ by definition of the notion of justification, $h_i$ is thus an $l$-sufficient justification of $t$. $\square$

[Proof of Proposition 10]

Let $h_i$ be a GAC sufficient justification of a removal $(x = v)$. 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 $(X, D, C \cup H \setminus \{h_i\})$ (denoted $P^g_{i\text{GAC}}$). Thus $v$ has no support on $C$ in $P^g_{i\text{GAC}}$; it does not belong to the domain of $x$ in $P^g_{i\text{GAC}}$; $h_i$ is thus not a sufficient justification of $(x \neq v)$.

Reciprocally, consider an assumption $h_i$ and suppose that $\forall C$ bearing on $x$, $\exists$ 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 $t = v$ valid in $P^g_{i\text{GAC}}$; $v$ thus belongs to the domain of $x$ in $P^g_{i\text{GAC}}$; by definition, this meant that $h_i$ is a GAC-sufficient justification of $(x \neq v)$. $\square$

[Proof of Proposition 11]

According to Proposition 10, when GAC is ensured $f(x \neq v)[h_i] = \bigwedge_{C,x \in \text{vars}(C) \in \text{Support}(x,v,C)} f_i[h_i]$ for any $h_i$, any $x \in X$ and any $v \in D_x$. I.e.:

$$f(x \neq v) = \bigwedge_{C,x \in \text{vars}(C) \in \text{Support}(x,v,C)} f_i$$

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: $f_i = \bigwedge_{v \in C \subseteq \{t\}} f_r$. Hence the result. $\square$