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).

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 [Amilhatre 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

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

## 2 Background and Problematics

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

A constraint \(C\) in \(C\) involves a set \(\text{vars}(C) \subseteq X\) 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 \(x_j\) 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, \((x_j, v)\) on \(C\)) iff \(t(x_j) = v\) and \(t\) satisfies \(C\).

An assignment \(t\) of \(X\) 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 \((X, D, C)\) and the current choices of the user by a set of couples \((x_i, v)\) where \(x_i\) is a variable in \(X\) 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 \((X, D, C, H)\) where \((X, D, C)\) is a CSP and \(H\) a finite set of constraints on variables of \(X\).

In configuration, \(H\) represents the current user choices, i.e. all assignments of the variables: we suppose in the sequel that all the restrictions in \(H\) bear with different variables and restrict their domains to a unique value; we will denote by \(h_i = (x_i \leftarrow v)\) the restriction from \(H\) 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 \(x_j \in X\) and any value \(v\) in its domain, for any constraint \(C\) bearing on \(x_j\), there exists an assignment \(t\) of the variables \(C\) 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 Elffie, 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 \(P = (X, D, C, H)\) 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 \((X, D, C \cup H)\).

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 \(P = (X, D, C, H)\) 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\((X, D, C \cup H \setminus \{h_i\})\). 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 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 these 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 [Jussieu and Ouis, 2001; Jussieu 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

---

1A full-proof version of the paper is available at ftp://ftp.irit.fr/IRIT/ADRIA/ijcai13BF.pdf
2Actually, 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 \(H\) bear on more than one variable is nevertheless questionable, hence our assumption.

Table 1: Assigned (\(*\)), forbidden (\(\times\)) and alternative (\(\circ\)) values for the A-CSP \(X = \{x_1, x_2, x_3\}, D = \{1, 2, 3, 4\}^{3}, C = \{\text{Alldiff}(f(x_1, x_2, x_3))\}, 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>(\circ)</td>
<td>(\circ)</td>
<td>(\times)</td>
<td></td>
</tr>
<tr>
<td>x_3(\times)</td>
<td>(\circ)</td>
<td>(\circ)</td>
<td>(\circ)</td>
<td></td>
</tr>
</tbody>
</table>


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 Blik, 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 \((X, D, C)\) and let \(H = \{h_i = (x_i \leftarrow d[x_i]) \mid i = 1, \ldots, \text{card}(X)\}\). \(d\) is a \((1, 0)\)-super-solution of \((X, D, C)\) if and only if, \(\forall x_i \in X\), 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, \(x'_1\) 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 referring 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.

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

#### 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 \(h_i \in H\). 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 \(P = (X, D, C, H)\) be an A-CSP and \(P^a\) the closure of \((X, D, C \cup H)\) by some level of local consistency \(a\).

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

Let \(C\) be a constraint in \(C\) and \(t\) an assignment of \(\text{vars}(C)\) satisfying \(C\). \(t\) is said to be invalid w.r.t. \(a\) if there exists \(x_j \in \text{vars}(C)\) 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 \(P = (X, D, C, H)\) be an A-CSP, \(a\) a level of local consistency, and \(R^a\) the \(P\)’s removals according to \(a\).

For any \(x_j\) in \(X\) and any \(v\) in \(D(x_j)\), \(h_i \in H\) 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 \((X, D, C \cup H \setminus \{h_i\})\).

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

For any \( x_i \in X \), 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 \( P = (X, D, C \cup \mathcal{H}) \) or \( (x_i \neq v) \in R^n \) 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 \( P = (X, D, C, \mathcal{H}) \) be an A-CSP, a a level of local consistency, \( C \) a constraint in \( C \) and \( t \) an assignment of \( \text{vars}(C) \) satisfying \( C \).

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

Our algorithm is based on the fact that an assignment \( h_i \) is an \( \alpha \)-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. \( a \) is the subset of \( R^n \) defined by: \( \text{CS}(t) = \{ (x_i \neq v) \in R^n \mid t[x_i] = v \} \).

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

Proposition 3 \( h_i \) is an \( \alpha \)-sufficient justification of a tuple \( t \) if and only if it is an \( \alpha \)-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.

Proposition 5 \( h_i \in \mathcal{H} \) is a \((1, k)\)-sufficient justification of \((x \neq v) \in R^{(1,k)} \) if and only if, for each assignment \( t \) of \( k \) variables there exists a support \( t \) of \( x = v \) on \( C \) such that \( h_i \) 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 \( f_{(x_i \neq v)}(h_i) \) is true if and only if \( h_i \) is a sufficient justification of \((x_i \neq v)\). According to Proposition 4, \( f_{(x_i \neq v)} \) depends on the justifications of the tuples that support \((x_i \neq v)\). 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 \( f_{(x_j \neq v)} \) flags of all the removals \((x_j \neq v)\) in the conflict set of \( t \). Formally:

Proposition 6 Let \( \text{Support}(x, v, C) \) is the set of assignments of \( \text{vars}(C) \) that support \((x, v)\); it holds that

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

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 to be arc consistent;
- \(\text{Table}(C)\) is the assignments of \(\text{vars}(c)\) that satisfy \(C\). Moreover the tuples involved in the tables are valid (i.e. \(\text{Table}(c)\) is a subset of the cartesian product of the domains of the variables its bears on);
- \(D_t(x_i)\) 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 \(C\);
- \(f_t\) 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 change. Since the vectors of justification are 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 \(Q\),
We have tested this algorithm on three industrial problems. Experimental results

Small

Medium

Big

Variables: 139, domain sizes from 2 to 16, 147 constraints of arity 2 to 6 (68Ko, total of 3,041 tuples).

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,
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.

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 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].

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


