Maintenance de valeurs alternatives dans les CSP dynamiques : principes et exp erimentations en con figuration de produit

Caroline Becker, Hélène Fargier

▶ **To cite this version:**

Caroline Becker, Hélène Fargier. Maintenance de valeurs alternatives dans les CSP dynamiques : principes et exp erimentations en con figuration de produit. Journées Francophones de Programmation par Contraintes, May 2012, Toulouse, France. hal-00817956

HAL Id: hal-00817956
https://inria.hal.science/hal-00817956

Submitted on 25 Apr 2013

HAL is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L'archive ouverte pluridisciplinaire **HAL**, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d'enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.

Maintenance de valeurs alternatives dans les CSP dynamiques : principes et expérimentations en configuration de produit

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

[1] IRIT, UMR 5505
Université Paul Sabatier, CNRS
118 Route de Narbonne, Toulouse

{becker, fargier}@irit.fr

Résumé

En configuration à base de contraintes, le catalogue est représenté par un CSP dont les solutions sont les différents produits configurés réalisables. Lors d'une session de configuration en ligne, le client cherche à définir le produit qui lui convient le mieux par affectation ou relaxation des différentes variables qui composent le produit configurable. À chaque étape d'interaction sont présentées, pour les variables non affectées, les valeurs de leurs domaines respectifs cohérentes avec les choix d'affectation déjà effectués sur les autres variables. La cohérence globale étant difficile à obtenir, ce filtrage est généralement établi par la maintenance d'un niveau de cohérence locale donné.

Dans ce travail, nous cherchons à rendre cette interaction plus conviviale en présentant aussi, pour les variables déjà affectées, les possibilités de changement de valeur d'affection, ou *valeurs alternatives* qui seraient cohérentes avec les autres choix effectués. Nous présentons donc ici une nouveau concept, celui de domaine alternatif d'une variable, et proposons une méthode pour calculer en une passe les domaines alternatifs de toutes les variables déjà affectées. De complexité au pire cas identique à la méthode Âń naïve Âż qui consisterait à simplement relaxer les choix portant sur la variable dont on veut calculer le domaine alternatif, cette méthode s'avère bien plus efficace expérimentalement.

Abstract

Constraint programming techniques are widely used to model and solve decision problems, and especially configuration problems. In this type of application, the configurable product is described by means of a set of constraint bearing on some configuration variable.

The user (typically, the customer) 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 the maintaining of global consistency is generally not tractable, the domains are generally filtered according to some level of local consistency.

In the present work, we aim at offering a more convenient level of interaction by providing the user with the alternative possible values of each of the already assigned variable - i.e. the values that could replace the current one without leading to a wipe out. 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 possible relaxation, but its experimental efficiency is much better.

*Financé par le projet ANR BR4CP

1 Introduction

La configuration de produits est une méthode de modélisation résultant de la personnalisation de masse (en anglais, *mass customization*) qui est un paradigme économique héritier de la production de masse (*mass production*) : nous sommes passés d'un article hautement standardisé, unique et produit en série à une gamme complexe et modulaire, c'est-à-dire une déclinaison du même produit [5]. Tout le problème alors,

pour le client, est de converger vers le produit le plus intéressant pour lui parmi l'ensemble, souvent hautement combinatoire, des produits réalisables.

Les approches par configuration à base de contraintes [8, 12, 6, 13, 14] proposent de représenter cette gamme par un problème de satisfaction de contraintes (CSP) sur un ensemble de variables de configuration et d'utiliser les algorithmes développés dans ce domaine, ou des variantes de ces algorithmes, pour fournir à l'utilisateur des fonctionnalités lui permettant de définir le produit configurable réalisable le plus en adéquation possible avec ses préférences.

Plusieurs travaux ont proposé d'utiliser des extensions des CSP pour formaliser des problèmes de configuration ; ces travaux traitent généralement de difficultés de représentation spécifiques à ces types de problèmes, comme par exemple le fait que l'existence d'une variable puisse dépendre de la valeur affectée à une autre. L'étude des problèmes de configuration a ainsi poussé la communauté à étendre le modèle CSP de base, définissant ainsi par exemple les CSP dynamiques [7], les CSP composites [11], les CSP interactifs [3], ou les CSP à hypothèses [1], etc.

Dans cet article, nous négligerons la prise en compte de ces difficultés de représentation pour nous tourner exclusivement vers l'interaction entre la machine et un client cherchant à définir un produit. Le principe de cette interaction est que l'utilisateur définit le produit qu'il désire en choisissant interactivement des valeurs pour les variables ; il peut également revenir en arrière en relâchant un ou plusieurs de ses choix. Après chaque action, les domaines de valeurs possibles fournis à l'utilisateur doivent être modifiés de manière à contenir toutes les valeurs compatibles avec les choix courants, et seulement les valeurs compatibles avec ces choix.

Nous proposons de rendre cette interaction plus pertinente en présentant non pas uniquement les domaines cohérents des variables non instanciées mais aussi les possibles changements de choix d'affectation, c'est-à-dire en présentant ce que nous appelons les domaines alternatifs des variables affectées.

Le présent article est structuré comme suit. La problématique des domaines alternatifs, et ses liens avec la notion de solution robuste proposée par Hebrard et al. [4] sont explicités dans la section 2. La section 3 présente notre approche algorithmique du calcul des domaines alternatifs. Nos premiers résultats expérimentaux sont donnés en section 4.

2 Problématique

Formellement, le produit configurable est donc représenté par un CSP $(\mathcal{X}, \mathcal{D}, \mathcal{C})$ et l'ensemble des choix courants par un ensemble de décision utilisateur, c'est-à-dire de couples $(x_i \leftarrow v)$ où x_i est une variable de $\mathcal{X}$ et $v \in D(x_i)$ est la valeur que l'utilisateur lui a affecté. Comme proposé par [1], le problème peut être représenté par un CSP à hypothèses :

Définition 1 (CSP à hypothèses) *Un CSP à hypothèses est un quadruplet $(\mathcal{X}, \mathcal{D}, \mathcal{C}, \mathcal{H})$ où :*

- *$(\mathcal{X}, \mathcal{D}, \mathcal{C})$ est un CSP ou $\mathcal{X} = \{x_1, \ldots, x_n\}$ est l'ensemble des variables, $\mathcal{D} = \prod_{j=1}^{N} D(x_j)$ le produit cartésien de leurs domaines initiaux et $\mathcal{C} = \{C_1, \ldots, C_m\}$ l'ensemble des contraintes. Une contrainte $C_i = (var(C_i), \mathcal{R}_i)$ est définie par l'ensemble $var(C_i) = \{x_{i_1}, \ldots, x_{i_k}\}$ des variables qu'elle impacte et par $\mathcal{R}_i \subset \prod_{j \in vars(C_i)} D(x_j)$ qui représente l'ensemble des valeurs que les variables contraintes peuvent prendre simultanément.*
- *$\mathcal{H}$ est un ensemble fini de contraintes unaires portant sur les variables de $\mathcal{X}$.*

Une décision utilisateur est un couple (x_i, v) où x_i étant une variable de $\mathcal{X}$ et $v \in D(x_i)$ la valeur que l'utilisateur lui a affecté. En configuration à base de contraintes, $\mathcal{H}$ représente l'ensemble des choix utilisateurs courants, avec une contrainte unaire par choix utilisateur ; les restrictions de $\mathcal{H}$) portent donc toutes sur des variables différentes. En pratique, on se limitera à des affectations et on notera $h_i = (x_i \leftarrow v)$ la restriction de $\mathcal{H}$ portant sur x_i, si elle existe ; les principes et définitions proposées ici sont valides également quand $\mathcal{H}$ contient des restrictions quelconques de domaines.

Après chaque choix, le système filtre le domaine des variables, ne laissant idéalement dans les domaines que des valeurs compatibles avec les choix courants. En pratique, un niveau de cohérence plus faible est assuré, généralement l'arc cohérence, et on appelle *CSP courant* la fermeture, selon ce niveau de cohérence locale, du problème initial.

Définition 2 (Domaine courant)
Soit a un niveau de cohérence locale et $P = (\mathcal{X}, \mathcal{D}, \mathcal{C}, \mathcal{H})$ un CSP à hypothèses.
On appelle CSP courant *la fermeture par a-cohérence de $(\mathcal{X}, \mathcal{D}, \mathcal{C} \cup \mathcal{H})$. On le notera $(\mathcal{X}, \mathcal{D}_c, \mathcal{C}_c$*
Le domaine courant *d'une variable x_i dans P est son domaine dans le CSP courant.*

Le domaine alternatif d'une variable est défini comme le domaine courant qu'aurait une variable instanciée si l'utilisateur remettait en cause cette affectation. C'est l'ensemble des valeurs que l'utilisateur peut donner à cette variable sans faire apparaître d'incohérence dans le CSP courant.

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

TABLE 1 – Valeurs affectées ($\star$), interdites ($\times$) et alternatives ($\diamond$) pour le CSP $X = \{x_1, x_2, x_3\}, \mathcal{D} = D_1 \times D_2 \times D_3 = \{1, 2, 3, 4\}^3, \mathcal{C} = \{$ Alldiff$(x_1, x_2, x_3)\}, \mathcal{H} = \{(x_1 \leftarrow 0), (x_2 \leftarrow 4)\})$.

Définition 3 (Domaine alternatif)
Soit a un niveau de cohérence locale et $P = (\mathcal{X}, \mathcal{D}, \mathcal{C}, \mathcal{H})$ un CSP à hypothèses.
Le domaine alternatif d'une variable x_i modulo a, est son domaine selon la fermeture par a-cohérence du CSP $(\mathcal{X}, \mathcal{D}, \mathcal{C} \cup \mathcal{H} \setminus \{h_i\})$.

Dans la suite, on utilisera la notation P_i^a pour désigner la fermeture par a-cohérence du CSP $(\mathcal{X}, \mathcal{D}, \mathcal{C} \cup \mathcal{H} \setminus \{h_i\})$.

Une valeur v est donc une valeur alternative pour x_i soit si elle est dans le domaine courant de x_i (c'est en particulier le cas lorsque x_i est affectée à v), soit si x_i est affectée à une autre valeur que v et que le seul retrait de cette affectation suffit à réautoriser v. Par exemple, si x_i est la dernière variable à avoir été affectée, toutes les valeurs de son domaine avant l'affectation sont dans son domaine alternatif. Un exemple de domaine alternatif peut être visualisé sur la figure 1.

La notion de domaine alternatif peut être rapprochée de la notion de $(1, 0)$ super-solution proposée par Hebrard et al. [4]. Rappelons qu'une (i, j) super solution d'un CSP est une solution complète d du CSP, c'est-à-dire une affectation de toutes les variables, telle que quel que soit l'ensemble $Y \subseteq X$ de i variables que l'on considère, il existe une solution d' du CSP telle que, (i) les valeurs données aux variables de Y diffèrent dans d et d' et (ii) d et d' diffèrent sur au plus $i + j$ variables au total. Cette notion généralise la notion de *fault tolerant solution* proposée par [15] - une *fault tolerant solution* étant une $(1, 0)$ super solution, c'est-à-dire une solution du CSP pour laquelle chaque variable affectée possède une valeur alternative : si le choix utilisateur devient impossible pour une raison ou pour une autre, on peut le Âń réparer Âż sans changer les valeurs des autres variables. En d'autres termes, une fois l'affectation complète d réalisée, chaque variable x_i doit posséder un domaine alternatif contenant au moins une valeur en plus de $d[x_i]$. Plus formellement, on peut montrer que :

Propriété 1

Soit d une affectation de toutes les variables d'un $(\mathcal{X}, \mathcal{D}, \mathcal{C})$ et $(\mathcal{X}, \mathcal{D}, \mathcal{C}, \mathcal{H})$ le CSP à hypothèses où $\mathcal{H} = \{h_i = (x_i \leftarrow d[x_i]), i = 1, card(\mathcal{X})\}$.
d est $(1, 0)$-super solution de $(\mathcal{X}, \mathcal{D}, \mathcal{C})$ si et seulement si pour tout $x_i \in \mathcal{X}$, le domaine alternatif de x_i par arc cohérenune ce est un sur ensemble strict de $\{d[i]\}$

Preuve : Soit d une affectation complète de X ; on note $d_{i \leftarrow v}$ l'affectation d' telle que $d[x_i] = v$ et que, pour tout $j \neq i$, $d'[x_j] = d[x_j]$. Considérons le CSP à hypothèses $(\mathcal{X}, \mathcal{D}, \mathcal{C}, \mathcal{H})$ correspondant : puisque toutes les variables sont affectée, $\forall i = 1, card(\mathcal{X}), h_i = (x_i \leftarrow d[x_i])$.

Supposons que pour tout x_i le domaine alternatif de x_i par arc cohérence est un sur ensemble strict de $\{d[i]\}$, i.e. que $D_{alt}^{arc}(x_i) \supsetneq \{d[x_i]\}$, et considérons une variable x_i quelconque ; par définition de la notion de domaine alternatif, le domaine de x_i dans P_i^{arc}, la fermeture par arc cohérence de $(\mathcal{X}, \mathcal{D}, \mathcal{C}, \mathcal{H} \setminus \{h_i\})$ contient au moins une valeur v différente de $d[x_i]$. Puisque $\mathcal{H}$ affecte toutes les variables, chacune des variables $x_j \neq x_i$ de P_i^{arc} est affectée à la valeur $d[x_j]$. Par définition de l'arc cohérence, et puisque que seule x_i est non affectée le domaine alternatif de x_i contient donc toutes les valeurs globalement cohérentes avec cette affectation ; $d_{i \leftarrow v}$ est donc une solution de $(\mathcal{X}, \mathcal{D}, \mathcal{C})$. Ce raisonnement s'appliquant à chaque, x_i, d est donc une $(1, 0)$ super solution de $(\mathcal{X}, \mathcal{D}, \mathcal{C})$.

Réciproquement, supposons que d est une $(1, 0)$ super solution de $(\mathcal{X}, \mathcal{D}, \mathcal{C})$. Par définition de cette notion, pour chaque x_i, il existe une valeur $v \neq d[x_i]$ telle que $d_{i \leftarrow v}$ soit une solution du CSP. Comme l'arc cohérence ne filtre que des valeurs ne conduisant à aucune solution, v appartient au domaine de x_i dans P_i^{arc} : le domaine alternatif de x_i contient donc, en plus de $d[x_i]$, la valeur v : c'est donc sur ensemble strict de $\{d[i]\}$.

□

La différence fondamentale entre les notions de $(1, 0)$ super-solution et de domaine alternatif est que la première notion est limitée aux affectations complètes, alors que la seconde propose des valeurs de réparation y compris pour des configurations partielles. Les algorithmes de calcul de super-solutions fondés sur la duplication de variable ne fonctionnent pas pour la maintenance de domaines alternatifs, comme le montre l'exemple de la figure 2 : lorsque l'utilisateur a choisi $x_1 \leftarrow 1$ et $x_2 \leftarrow 2$, la propagation réduit le domaine de x_3 à $\{3\}$; et celui de x'_1, qui devrait porter les valeurs alternatives de x_1, est alors vidé, alors que 3 fait bien parti du domaine alternatif de x_1.

Notons enfin que l'objectif pratique est différent : dans le premier cas, on cherche quelques solutions robustes (mais pas toutes : elles sont potentiellement

en nombre exponentiel), et seulement des solutions robustes alors que dans le second, on désire calculer *toutes* les valeurs alternatives, si il y a.

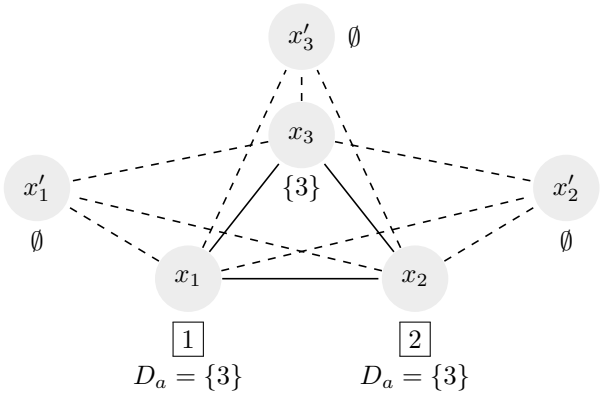

FIGURE 1 – Représentation de $x_1 \neq x_2$, $x_2 \neq x_3$, $x_1 \neq x_3, \forall i, D(x_i) = \{1, 2, 3\}$ lorsque l'utilisateur a choisi $x_1 \leftarrow 1$ et $x_2 \leftarrow 2$.

C'est pourquoi certaines techniques de calcul des super-solutions fondées sur la duplication des variables ne fonctionnement pas pour calculer le domaine alternatif. Intuitivement, c'est parce que, une fois x instanciée, le domaine de y en est restreint et que ces restrictions vont ensuite se répercuter sur le double de x. En revanche, une fois y instanciée, toute valeur retirée du domaine de y l'est à cause de ce choix et donc indépendant du choix sur x. Cela peut être visualisé sur la figure 2, où le domaine de x'_1 est vide car le domaine de x_3 est réduit à $\{3\}$ alors que 3 fait bien parti du domaine alternatif de x_1.

Le domaine alternatif est une notion orthogonale à la notion d'explications de retraits, telles que celle proposée par PaLM [10]. En effet, les explications servent à justifier de la restriction de domaines et à proposer une stratégie de restauration de ces valeurs par relâchement des choix déjà effectués, alors que la présentation du domaine alternatif permet justement d'effectuer des restaurations de valeur sans avoir besoin de modifier les autres variables. Disons qu'une valeur est alternative si son retrait n'a d'autre explication que son l'affectation de sa variable : la notion est moins lourde, et son calcul ne nécessite pas la mémorisation d'explications.

3 Calcul des domaines alternatifs

Lorsque n variables sont affectées, une méthode naïve pour calculer leurs domaines alternatifs respectifs consiste en la génération de $n+1$ copies du CSP : un CSP de référence P_0 où toutes les variables sont instanciées, et n CSP P_i, chaque P_i ayant les mêmes affectations que P_0, sauf pour la variable x_i. Chacun des P_i est alors filtré par a-cohérence. Cette méthode, qui a l'avantage de demander peu d'espace mémoire supplémentaire, servira de point de référence par rapport auquel comparer l'algorithme que nous présentons, qui prend le parti exactement inverse, à savoir stocker les informations utiles afin d'éviter des calculs redondants.

Si, lors de la présentation de la notion de domaine alternatif, on ne faisait aucune supposition sur les domaines des variables qui pouvaient être continus, discrets, etc ; nous supposons par la suite que le domaine de chaque variable est fini.

3.1 Retraits et justification suffisantes

L'idée est de maintenir, pour chaque retrait de valeur d'un domaine, une liste de flags : un par affectation $h_i \in \mathcal{H}$. Ce flag est positionné à vrai si et seulement si la remise en cause de l'affectation provoque le retour de la valeur dans le domaine de sa variable.

Définition 4 (Retrait)
Soit $P = (\mathcal{X}, \mathcal{D}, \mathcal{C}, \mathcal{H})$ un CSP à hypothèses et $(\mathcal{X}, \mathcal{D}_c, \mathcal{C}_c$ la fermeture par $a-$cohérence de $(\mathcal{X}, \mathcal{D}, \mathcal{C} \cup \mathcal{H})$ (le CSP courant).
Un retrait est un couple $r = (x_j, v)$ tel que $v \in D(x_j)$ et $v \notin D_c(x_j)$
On note $\mathcal{R}$ est l'ensemble des retraits de P.

Pour plus de clarté, un retrait (x_j, v) sera souvent noté $(x_j \neq v)$

Définition 5 (Justification Suffisante)
Soit $P = (\mathcal{X}, \mathcal{D}, \mathcal{C}, \mathcal{H})$ un CSP à hypothèses.

Pour toute valeur v dans le domaine initial d'une variable x_j, et tout $h_i \in \mathcal{H}$, on dit que h_i est une justification suffisante de $x_j \neq v$ modulo une propriété de cohérence locale *a (ou "justification a-suffisante") si v appartient au domaine de x_j dans la fermeture par a-cohérence de $(\mathcal{X}, \mathcal{D}, \mathcal{C} \cup \mathcal{H} \setminus \{h_i\})$.*

On dit que h_i est une justification suffisante d'un retrait $r = (x_j \neq v) \in \mathcal{R}$, ssi c'est une justification a-suffisante de v.

Typiquement, si la dernière affectation a provoqué l'elimination de v pour x_j, elle est une justification suffisante de $(x_j \neq v)$.

De la même façon, les retraits invalident des tuples qui sont eux-mêmes supports pour des valeurs, et la notion de justification suffisante peut être étendue aux tuples :

Définition 6 (Ensemble de conflits, tuple invalide)

Un retrait $(x \neq v)$ et un tuple t sont en contradiction directe ssi $t[x] = v$.
L'ensemble des conflits d'un tuple est l'ensemble des retraits avec lesquels il est en contradiction directe : $CS(t) = \{(x_i \neq v) \in \mathcal{R}$ t. q. $t[x_i] = v\}$. Un tuple est invalide ssi il est en contradiction directe avec au moins un retrait.

Définition 7 (Justification suffisante d'un tuple)

Soit $t \in R_C$ un tuple dans la table d'un contrainte C. On dit que h_i est une justification a-suffisante pour t ssi, pour tout $x_j \in vars(C)$, $t[j]$ appartient au domaine de x_j dans la fermeture par a-cohérence de $(\mathcal{X}, \mathcal{D}, \mathcal{C} \cup \mathcal{H} \setminus \{h_i\})$.

Au sens de ces définitions, h_i est également une justification suffisante de t (respectivement de v pour x_j) si t est valide (resp. si v est dans le domaine courant x_j). En fait, pour les tuples valides (et les valeurs des domaines courants), toute hypothèse est une justification suffisante.

Notre algorithme est fondé sur le fait que l'affectation h_i est une justification a-suffisante pour le tuple t ssi, pour chaque variable x_j concernée, soit $t[x_j]$ est dans le domaine courant de x_j, soit h_i est une justification suffisante de chacune des valeurs du tuple qui ne sont pas dans les domaines courants. C'est le sens de la propriété suivante :

Propriété 2 *h_i est une justification suffisante d'un tuple invalide t ssi c'est une justification suffisante de tous les retraits de son ensemble de conflits.*

Preuve :
$\boxed{\Rightarrow}$ Supposons par l'absurde qu'il existe h_i justification a-suffisante d'un tuple t et $(x \neq v)$ un retrait appartenant à l'ensemble de conflits de t (i.e. $t[x] = v$) dont h_i ne soit pas une justification suffisante.
On note P_i^a la fermeture par a-cohérence de $(\mathcal{X}, \mathcal{D}, \mathcal{C} \cup \mathcal{H} \setminus \{h_i\})$. Puisque h_i est une justification a-suffisante de t , par définition, t est valide P_i^a. D'un autre coté, puisque h_i n'est pas une justification suffisante de $(x \neq v)$, v n'est pas dans le domaine de x dans P_i^a ; t est donc invalide dans P_i^a, de qui contredit la déduction précédente.
$\boxed{\Leftarrow}$ Réciproquement, soit h_i une justification a-suffisante de tous les retraits de l'ensemble des conflits d'un tuple t. Pour chacun de ces $(x_j \neq v_j)$, v_j est par définition dans le domaine de x_j dans P_i^a. Donc t est un tuple valide dans P_i^a - par définition de la notion, h_i est donc une justification suffisante de t.
□

On peut montrer que, lorsque la propriété de cohérence locale à assurer est l'arc-cohérence généralisée :

Propriété 3 *h_i est une justification suffisante d'un retrait $(x \neq v)$ modulo l'arc-cohérence (GAC) ssi, pour chacune des contraintes qui portent sur x, il existe un tuple t support de $x = v$ dont h_i soit une justification GAC-suffisante.*

Preuve :

$\boxed{\Rightarrow}$ Soit h_i est une justification suffisante d'un retrait $r = (x \neq v)$ modulo l'arc-cohérence (GAC). Supposons par l'absurde une contrainte C portant sur la variable x telle que h_i ne soit une justification suffisante d'aucun des tuples $t \in R_C$ supports de $x = v$, ce qui signifie que tous ces t sont invalides dans P_i^{GAC}. Donc v n'a aucun support par arc cohérence sur C dans P_i^{GAC} : il n'est donc pas dans le domaine de x dans la fermeture par arc cohérence de P_i^{GAC} : h_i n'est pas une justification suffisante de r.

$\boxed{\Leftarrow}$ Réciproquement, considérons une hypothèse h_i et supposons que, $\forall C$ portant sur x, $\exists t$ support de $x = v$ dont h_i soit une justification GAC-suffisante, c'est à dire que pour chacune contrainte C portant sur x, il existe une support t de $x = v$ valide dans P_i^{GAC} ; par définition de l'arc cohérence, v appartient donc au domaine de x dans P_i^{GAC}, i.e.
h_i est une justification GAC-suffisante de r. □

Des propriétés similaires peuvent être établies pour d'autres niveaux de cohérence locale induisant des retraits dans les domaines :

Définition 8
Soit v une valeur dans le domaine initial d'une variable x, $\mathcal{V}$ un ensemble de k variables et $\mathcal{C}_k$ l'ensemble des contraintes qui portent sur $\{x\} \cup \mathcal{V}$.
v est localement cohérente par rapport à $\mathcal{V}$ ssi il existe une affectation t de $\{X\} \cup \mathcal{V}$ telle que $t[X] = v$ et $\forall C \in \mathcal{C}_k, t[vars(C)] \in R_C$, où $t[vars(C)]$ est la projection de t sur les variables de C.

On dit que t est un support de v sur $\mathcal{V}$.

Définition 9
Un CSP est $(1, k)$ cohérent ssi, pour toute variable et toute valeur dans son domaine, pour tout groupe $\mathcal{V}$. de k variables, v admet un support sur $\mathcal{V}$.

Propriété 4
h_i est une justification $(1, k)$-suffisante d'un retrait $(x \neq v)$ ssi, pour chaque groupe $\mathcal{V}$ de k variables il existe support t de v dont h_i est justification $(1, k)$-suffisante.

Preuve : $\forall(x_1,...,x_k), \exists(v_1,...,v_k)$ support de $x=v$ tel que h_i justification $(1,k)$-suffisante de $(v_1,...,v_k)$ $\Leftrightarrow$ $\forall(x_1,...,x_k), \exists(v_1,...,v_k)$ support de $x=v$ tel que chacun des v_j appartienne au domaine de x_j dans la fermeture par $(1,k)$-cohérence de $(\mathcal{X},\mathcal{D},\mathcal{C}\cup\mathcal{H}\setminus\{h_i\})$ $\Leftrightarrow$ v appartient au domaine de x dans la fermeture par $(1,k)$-cohérence de $(\mathcal{X},\mathcal{D},\mathcal{C}\cup\mathcal{H}\setminus\{h_i\})$ (par définition de la $1,k$ cohérence)

$\Leftrightarrow$ h_i est une justification $(1,k)$-suffisante de $x\neq v$ □

3.2 Algorithme de maintenance des domaines alternatifs

		supports de $x_j=v_j$			
	justification	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		$\star$	$\star$	$\star$
résultante	h_1,h_2,h_4	h_1	h_2	h_2,h_4	$\emptyset$

TABLE 2 – Calcul du tableau de flags d'un retrait ($x\neq v$) sur une contrainte C ; les t_i sont les supports de $x=v$. Un $\star$ dans la case $(t_i, x_j\neq v_j)$ signifie que le tuple t_i est invalide lorsque $x_j\neq v_j$.

Notre algorithme est un algorithme de type GAC4 [9] - la différence est qu'une valeur retirée va éventuellement être retirée plusieurs fois (à cause de plusieurs contraintes), et que cette nouvelle cause de retrait doit être propagée. Plus largement, un retrait $(x\neq v)$ doit être propagé à chaque modification de son vecteur de justifications. Il est défaillé ci dessous (Algorithmes 1 et 2). La modification de ce vecteur, comme celle des vecteurs de flag des tuples, étant monotone (une justification supposée suffisante peut disparaître, mais jamais redevenir suffisante), l'algorithme termine. Sans le mémoriser forcément, on peut parler des flags d'un tuple : c'est un vecteur de bits, qui, pour chaque h_k, porte la valeur vrai ssi h_k est une justification suffisante pour chacune des valeurs du conflit set de t.

En effet, pour que le choix utilisateur h_k soit une justification suffisante de $x\neq v$ sur C, il faut que la remise en cause de h_k aboutisse à ce qu'au moins un tuple de C compatible avec $x=v$ devienne à nouveau valide ; et un tuple t redevient valide par la remise en cause de h_k ssi tous les elements de son ensemble de conflit ont h_k comme justification suffissante. En d'autre termes les flags d'un tuple est l'intersection des flags des retrait qui contredisent le tuple. Finalement lorsque c'est une propriété d'arc cohérence qui est maitentenu, on obtient la propriété 5 :

Propriété 5

$$f_{(x\neq v)} = \bigwedge_{C,x=out(C)} \bigvee_{t\in Support(x,v,C)} \bigwedge_{r'\in CS(t)} f(r')$$

Preuve : L'ensemble des justifications AC-suffisantes d'un retrait est, d'après la proposition 3,

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

Or, d'après la proposition 2, l'ensemble des justifications AC-suffisantes d'un tuple est l'intersection des justifications AC-suffisantes des retraits de son ensemble de conflits. □

L'algorithme 2, inspiré de l'algorithme GAC4 [9] décrit la mise à jour des vecteurs de flags des retraits selon cette formule. Nous notons :

- n, le numéro d'instantiation. Vaut initialement 0.
- $Cpt(x,v,C)$, le nombre de tuples supports dans la contrainte C de $x=v$.
- $f_{(x_k\neq u)}$, l' ensemble de justifications suffisantes du retrait $(x_k\neq u)$. $f_{(x_k\neq u)}[h_k]=true \Leftrightarrow h_k$ est une justification suffisante de $x_k\neq u$. Vaut initialement Vrai^n
- $D_c(x_i)$ le domaine courant de la variable x_i.
- $D(x_i)$ le domaine initial de la variable x_i.
- $S_{i,v,C}$, l'ensemble des tuples t de la contrainte C tels que $t[i]=v$.
- Mem : Buffer
- $f(t)$: Ensemble des justifications suffisantes du tuple t. Initialement, vaut Vrai^n

```
Procedure Initialise((X, D_c, C_c) : CSP initial)
/* (X, D_⌋, C_⌋) supposé arc cohérent */
/* Tous les tupples sont supposés valides */
begin
    pour tous les Contraintes C faire
        pour tous les x_i ∈ vars(C), v ∈ D(x_i) faire
            Cpt(x_i, v, C) := 0;
        fin
        pour tous les t ∈ vars(C) faire
            f_t = Vrai^n;
            valide(t) = Vrai CS(t) = Faux^n;
            f(t) = Vrai^n;
            pour tous les x_i ∈ vars(C) faire
                Cpt(x_i, t[i], C) + +;
            fin
        fin
    fin
end
```

Algorithm 1: Initialisation

```
Procedure Propagate( (x_k = v) : hypothèse ;
(X, D_c, C_c) : CSP courant) Q := ∅;
pour tous les u ≠ v ∈ D_c(x_k) faire
    f(x_k ≠ u) ← Faux^n;
    si u ∈ D_c(x_k) alors
        f(x_k ≠ u, h_k) ← Vrai;
    fin
    Ajouter ((x_k ≠ u)) à Q;
fin
pour Q non vide faire
    Choisir et retirer un nœud (x_i ≠ v) de Q;
    si v ∈ D_c(x_i) alors
        Retirer v de D_c(x_i);
    fin
    pour tous les C portant sur x_i et tout tuple
    t de S_{i,v,C} faire
        Mem ← f(t);
        f(t) ← f(t) ∧ f(x_i ≠ v);
        si t était valide alors
            pour tous les x_j ≠ x_i ∈ vars(t) faire
                Cpt(x_j, t[j], C)^{--};
                si Cpt(x_j, t[j], C) = 0 alors
                    f_(x_i ≠ v) ← Vrai^n  /* init */ ;
                    Ajouter ((x_j ≠ v'), C) à Q;
                fin
            fin
            valide(t) = false;
        fin
        si Mem! = f(t)  Une justif de t a disparu
        alors
            pour tous les x_j ≠ x_i ∈ vars(t) faire
                mem' = f_{x_j≠t[j]};
                f_{x_j≠t[j],C} = f_{x_j≠t[j],C} ∨ f(t);
                f_{x_j≠t[j]} = f_{x_j≠t[j]} ∧ f_{x_j≠t[j],C};
                si mem' ≠ f_{x_j≠t[j]} alors
                    Ajouter (x_j ≠ t[j]) à Q;
                fin
            fin
        fin
    fin
fin
```

Algorithm 2: Propagation de la décision utilisateur $h_k = (x_k = v)$

Le surcout des calculs de vecteurs est négligeable (on les effectue en temps constant par conjonction ou disjonction de bitvecteurs)

Au lieu de rentrer une seule fois dans la boucle Q, comme dans un GAC4 classique, chaque retrait peut rentrer n fois, autant qu'il y a de changements possible dans un vecteur de flags. La complexité en pire cas est majorée par $O(n.e.d^k)$ avec e le nombre de contraintes, m le nombre de variables, n le nombre d'hypothèses d la taille maximale du domaine d'une contrainte et k l'arité maximale d'une contrainte. C'est donc la même que celle de la méthode naive fondée sur GAC-4, dont notre algorithme est inspiré : $n.O(e.d^k)$. A ceci pres que, dans la méthode naive, GAC-4 tourne exactement n fois, alors qu'ici il tourne au plus n fois.

En termes d'espace, sont mémorisés par GAC4 les ensembles $S_{i,v,C}$ de supports de chaque valeur - une structure qui est en $O(A)$, A étant l'espace pris pas le tuples admissibles contenus dans les tables des contraintes. Notre algorithme maintient de plus, pour chaque tuple t, un vecteur de n booléennes (les flags) soit un espace $O(T.n)$, T étant le nombre de tuples admissibles . Le problème produit au plus autant de retraits que de valeurs dans les domaines, pour chacun desquels on mémorise également un vecteur de n booléens - donc une structure en $O(d.m.n)$. Soit par rapport à l'algorithme de base, GAC4, un espace supplémentaire bornée par $O(n.(T + d.m))$.

4 Résultats Expérimentaux

Nous avons testé ces algorithmes sur un benchmark issus du monde industriel. Il relativement petit (32 variables, de domaines de taille 2 à 10, et 35 contraintes binaires) [1]. Il s'agit d'une souffleuse, c'est-à-dire une machine créant de manière automatique des bouteilles dans différentes matériaux.

Chaque point de l'échantillon simule une session de configuration. Il consiste en une série d'affectations, selon un ordre aléatoire, de toutes les variables du problème. A chaque affectation de variable, les domaines des variables libres sont filtrés par arc cohérence et les domaines alternatifs de toutes les variables déjà affectées sont calculés. L'ordre dans lequel les variables sont affectées est choisi selon un distribution uniforme, et pour chaque variable, la valeur affectée est choisie de manière équiprobable parmi les valeurs restant dans son domaine (i.e. choisie dans son domaine courant). Notre échantillon contient 1000 séquences d'affectation.

Pour chaque séquence $(x_1, \ldots, x_m\}$ nous mesurons, à chaque affectation $x_k = v_k$, le temps nécessaire pour

1. Il peut être retrouvé sur `ftp://ftp.irit.fr/pub/IRIT/ADRIA/PapersFargier/Config/___Souffleuse.csp`

calculer les domaines alternatifs des variables précédemment affectées (i.e. x_1 à x_i). Tout le processus est répété pour les deux algorithmes, qui jouent donc sur les mêmes séries d'affectations.

La figure 2 présente les résultats de cette expérimentation. On a porté en abscise la position de l'affectation dans la sequence. La courbe en plein représente le temps moyen nécessité par la méthode naïve (croissante, ronds), celle en pointillé (stable, carrés) la moyenne des temps nécessités de notre algorithme.

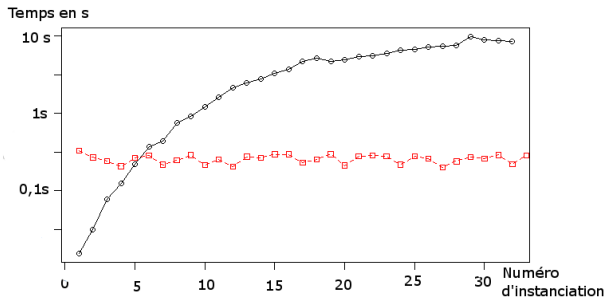

FIGURE 2 – Comparaison du temps de calcul de la méthode naïve (en noir, △, croissante) et de notre algorithme (en rouge □, stable.).

Les résultats sont excellents : l'algorithme que nous proposons offre des résultats plus rapides dès que plus de 5 variables sont affectés - i.e. dès que le nombre de domaines alternatifs à calculer dépasse 5. Comme on pouvait s'en douter, le temps nécessité par l'algorithme naif croît avec le nombre de variables concernées, alors que le temps nécessité par notre algorithme reste stable.

5 Conclusion et perspectives

Dans ce travail, nous avons proposé un nouveau concept, celui de domaine alternatif d'une variable selon un niveau de cohérence locale. Nous avons présenté, pour la maintenance des domaines alternatifs modulo l'arc cohérence généralisée, une extension de l'algorithme GAC4 qui consiste en une prolongation de la propagation des retraits au delà des domaines des variables concernées. Par opposition à une méthode naive qui consisterait à effectuer from scratch autant de propagations qu'il y a de domaines alternatifs à calculer, notre méthode mémorise des justifications (limitées) des retraits. Elle consomme plus d'espace mémoire que la méthode naive et nécessite des contraintes définies en extension. Nos premieres experimentations, sur un cas réel, suggèrent qu'elle est rapidement plus efficace que la méthode naive.

L'espace nécessaire reste une limite de la méthode ; le surcoût en espace, par rapport à l'algorithme GAC4 depend directement du nombre de variables dont on veut calculer le domaine alternatif. Cela dit, il faut garder à l'esprit que l'on ne demandera pas en pratique au système de présenter les domaines alternatifs de toutes les variables : l'utilisateur est opérateur humain avec une charge mentale limitée et il n'est pas évident qu'il puisse (ou même désire) se représenter efficacement un grand nombre de domaines alternatifs. Dans une application de configuration par exemple, on ne présentera les domaines alternatifs que de certaines variables, par exemple celles correspondant au sous composant que l'on est en train de configurer.

Les concepts développés dans cet article, nous l'avons vu, sont assez proches des notions de solution robuste et de restauration de valeur. Dans le travail present, nous nous sommes focalisés sur le calcul des domaines alternatifs. Cela dit, une partie des information utilisées par l'algorithme peut mieux être exploitée lors de l'interaction avec l'utilisateur : une justification suffisante d'un retrait ($x \neq v$) peut être vue comme une restauration de taille 1 de v, et être présentée en tant que telle lorsque l'utilisateur cherche à libérer une valeur interdite par les choix courants.

Ce travail suggère de nombreux développements et perspectives. Tout d'abord, il faut évidement améliorer notre algorithme, par exemple en proposant une version paresseuse, et compléter nos expérimentations. Il faudra ensuite réfléchir à ce qui peut être fait pour le maintien des domaine alternatifs dans les CSP qui ne limitent pas leurs contraintes à des représentations sous forme de table. Enfin et surtout, il faudrait savoir prendre en compte l'interaction dans son ensemble : pour l'instant, nous n'avons considéré que l'affectation de valeur ; reste à traiter la relaxation par l'utilisateur d'un ou plusieurs de ses choix. Cette adaptation supposera une hybridation avec les algorithmes de maintient de cohérence dans les CSP dynamiques [2].

Références

[1] Jérôme Amilhastre, Hélène Fargier, and Pierre Marquis. Consistency restoration and explanations in dynamic cspsapplication to configuration. *Artificial Intelligence*, 135(1-2), 2002.

[2] Romuald Debruyne and Christian Bessière. From restricted path consistency to max-restricted path consistency. In *CP*, pages 312–326, 1997.

[3] E. Gelle and R. Weigel. Interactive configuration using constraint satisfaction techniques. In *Proceedings of PACT-96*, pages 37–44, 1996. la construction et la maintenance d'un CSP est plus facile que celles d'un ensemble de règles ; on ne

peut attribuer de source à une règle (marketing, restriction technique) alors qu'on le peut pour une contrainte ; CSP continus ; interaction avec l'utilisateur via un outil de CAO et non par des symboles alphanumériques.

[4] E. Hebrard, B. Hnich, and T. Walsh. Super solutions in constraint programming. *Integration of AI and OR Techniques in Constraint Programming for Combinatorial Optimization Problems*, pages 157–172, 2004.

[5] Hanna Kropsu-Vehkapera, Harri Haapasalo, Olli Jaaskelaine, and Kongkiti Phusavat. Product configuration management in ict companies : The practitioners' perspective. *Technology and Investment*, 2(4) :0–0, 2011.

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

[7] S. Mittal and B. Falkenhainer. Dynamic constraint satisfaction problems. In *Proceedings of AAAI'90*, pages 25–32, 1990. Première définition des DynamicCSP ; caractérisation de contraintes d'activité ; implémentation à l'aide d'un ATMS.

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

[9] R. Mohr, G. Masini, et al. Good old discrete relaxation. 1988.

[10] Samir Ouis, Narendra Jussien, and Olivier Lhomme. Explications conviviales pour la programmation par contraintes. In *JFPLC*, pages 105–, 2002.

[11] D. Sabin and E. C. Freuder. Configuration as composite constraint satisfaction. In *AI and Manufacturing Research Planning Workshop*, pages 153–161, 1996. Composite CSP = CSP récursif à qui on ajoute l'état "inactif" au domaine de chacune des variables.

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

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

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

[15] R. Weigel, C. Bliek, and B.V. Faltings. On reformulation of constraint satisfaction problems (extended version). In *Artificial Intelligence*. Citeseer, 1998.