Un schéma générique pour intégrer des consistances fortes dans les solveurs de contraintes

Julien Vion **Thierry Petit** **Narendra Jussien**

École des Mines de Nantes, LINA UMR CNRS 6241,
4, rue Alfred Kastler, FR-44307 Nantes, France.
julien.vion@emn.fr, thierry.petit@emn.fr, narendra.jussien@emn.fr

Résumé

Nous présentons un schéma générique pour appliquer les consistances fortes avec les outils standard de programmation par contraintes. Ce schéma est notamment applicable aux solveurs événementiels. Nous proposons d'encapsuler un sous-ensemble des contraintes du modèle dans une contrainte globale. Notre approche permet de spécifier un niveau de consistance différent pour différents sous-ensembles de contraintes d'un même modèle. De plus, nous montrons comment des consistances fortes peuvent être appliquées à différents types de contraintes, dont des contraintes définies par l'utilisateur. Afin de souligner l'efficacité pratique de notre paradigme, nous proposons un nouvel algorithme à gros grain pour établir Max-RPC, nommé Max-RPC^{rm}. Nos expérimentations confirment l'intérêt d'utiliser des consistances fortes au sein des outils de programmation par contraintes, particulièrement lorsqu'on applique un niveau de consistance différent pour différents sous-ensembles de contraintes dans un même réseau.

1 Introduction

Cet article présente un schéma générique pour intégrer des consistances fortes dans les outils existants de résolution de problèmes en programmation par contrainte (PPC), et notamment les solveurs de contraintes événementiels. Nous expérimentons notre approche avec un nouvel algorithme « à gros grain » pour la consistance Max-RPC.

Les techniques les plus courantes pour résoudre des problèmes de satisfaction de contraintes sont basées sur l'application de consistances locales. Les consistances locales suppriment des valeurs qui n'appartiennent à aucune solution. Afin d'appliquer un niveau de consistance donné, des *propagateurs* sont associés aux contraintes. Un propagateur est complet s'il élimine toutes les valeurs qui ne peuvent pas satisfaire la contrainte. Une des raisons du succès de la PPC dans la résolution de problèmes réels est que ces propagateurs sont codés à l'aide d'algorithmes de filtrage qui exploitent la sémantique des contraintes. Ces algorithmes sont le plus souvent issus de techniques efficaces de recherche opérationnelle.

La plupart des solveurs de contraintes sont basés sur un schéma de propagation type AC-5 [20]. On les appelle des solveurs événementiels. Dans un tel schéma, chaque propagateur est appelé lorsque des événements surviennent sur les domaines des variables impliquées dans chaque contrainte. À un nœud donné de l'arbre de recherche, le filtrage est réalisé à l'intérieur de chaque contrainte. Une contrainte reçoit les événements relatifs à ses variables et réalise un filtrage, qui déclenche de nouveaux événements. Le point fixe est obtenu après un cycle de propagation des événements impliquant toutes les contraintes, lorsque plus aucun nouvel événement n'est déclenché. Dans un tel contexte, la consistance d'arc généralisée (GAC) est le niveau de consistance locale le plus élevé. Il correspond au cas où tous les propagateurs sont complets.

Il existe pourtant, dans la littérature, des consistances locales plus fortes que GAC [9, 6]. Elles nécessitent la prise en compte de plusieurs contraintes simultanément pour être appliquées. On considère parfois que ces consistances fortes ne peuvent pas (facilement) être intégrées aux outils de PPC, parmi lesquels, notamment, les solveurs événementiels. Ces outils ne proposent pas ces consistances plus fortes, qui, en conséquence, sont rarement utilisées pour résoudre des problèmes industriels.

Cet article démontre que les consistances fortes sont exclues à tort des outils de PPC. Nous présentons un paradigme générique pour ajouter facilement de telles

consistances aux solveurs de contraintes. Notre idée est de définir une contrainte globale [7, 2, 16], qui encapsule un sous-ensemble du réseau de contraintes initial. Le niveau de consistance souhaité est alors appliqué sur ce sous-ensemble de contraintes. On notera que, généralement, une contrainte globale représente un sous-problème ayant une sémantique bien fixée. Ce n'est pas le cas dans notre approche. La contrainte globale est ici utilisée pour appliquer une technique de propagation particulière sur un sous-ensemble de contraintes. Cette idée est similaire à celle utilisée pour la résolution de problèmes sur-contraints dans [17].

Dans la littérature, Bessière et Régin ont proposé d'augmenter la propagation sur des parties d'un réseau de contraintes, en résolvant des sous-problèmes à la volée [4]. Notre approche permet d'utiliser plusieurs niveaux de consistance locale sur plusieurs sous-ensembles de contraintes dans le même modèle. Notre démarche se rapproche de certains travaux récents sur les CSP binaires, relatifs à des heuristiques ayant pour but d'adapter dynamiquement le niveau de propagation à appliquer lors d'un processus de résolution [18]. Notre approche vise à appliquer efficacement différentes consistances fortes, sans limitation sur l'arité ou la nature des contraintes. Enfin, dans une contrainte globale il est possible de gérer les événements de suppression de la manière la mieux adaptée à la consistance que l'on veut appliquer (gestion orientée par les variables ou par les contraintes). Généralement, les solveurs événementiels ne permettent pas une gestion aussi fine de la propagation.

Nous expérimentons notre méthode avec la consistance forte Max-RPC [8], à l'aide du solveur de contraintes Choco [1]. Nous présentons un nouvel algorithme gros grain pour Max-RPC. Cet algorithme exploite des structures de données « stables au retour arrière », de manière similaire à AC-3^{rm} [15], l'un des algorithmes de consistance d'arc les plus efficaces.

2 Préliminaires

Un *réseau de contraintes* $\mathcal{N}$ est constitué d'un ensemble de variables $\mathscr{X}$, d'un ensemble de domaines $\mathscr{D}$, tels que le domaine $\mathrm{dom}(X) \in \mathscr{D}$ de la variable X soit l'ensemble fini des (au plus d) valeurs que la variable X peut prendre, et d'un ensemble $\mathscr{C}$ de e contraintes qui spécifie les combinaisons de valeurs autorisées pour des ensembles de variables données. Une instanciation I est un ensemble de couples variable/valeur (X, v), noté X_v. I est *valide* ssi pour toute variable X impliquée dans I, $v \in \mathrm{dom}(X)$. Une *relation* R d'arité k correspond à un nombre quelconque d'instanciations de la forme $\{X_a, Y_b, \ldots, Z_c\}$, où $a, b, \ldots, c$ sont des valeurs d'un univers donné. Une *contrainte* C d'arité k est un couple $(\mathrm{scp}(C), \mathrm{rel}(C))$, où $\mathrm{scp}(C)$ est un ensemble de k variables et $\mathrm{rel}(C)$ une relation d'arité k. $I[X]$ est la valeur de X dans I. Pour des contraintes binaires, C_{XY} désigne la contrainte telle que $\mathrm{scp}(C) = \{X, Y\}$. Étant donnée une contrainte C, une instanciation I de $\mathrm{scp}(C)$ (ou d'un sur-ensemble de $\mathrm{scp}(C)$), en considérant alors uniqement les variables de $\mathrm{scp}(C)$), *satisfait* C ssi $I \in \mathrm{rel}(C)$. I est alors *autorisée* par C.

Une *solution* d'un réseau $\mathcal{N}(\mathscr{X}, \mathscr{D}, \mathscr{C})$ est une instanciation I_S de toutes les variables $\mathscr{X}$ telle que (1.) $\forall X \in \mathscr{X}, I_S[X] \in \mathrm{dom}(X)$, et (2.) I_S satisfait toutes les contraintes de $\mathscr{C}$ (I_S est *valide* et *autorisée* par toutes les contraintes de $\mathscr{C}$).

2.1 Consistances locales

Définition 1 (Support, Arc-Consistance)**.** *Soit C une contrainte et $X \in \mathrm{scp}(C)$. Un* **support** *pour la valeur $a \in \mathrm{dom}(X)$ sur C est une instanciation $I \in \mathrm{rel}(C)$ telle que $I[X] = a$. $a \in \mathrm{dom}(X)$ est* **arc-consistante** *par rapport à C ssi elle a un support sur C.*

Pour une contrainte binaire, *i.e.*, impliquant X et Y, le support $I = \{X_a, Y_b\}$ de la valeur X_a peut être caractérisé par la valeur Y_b. On dit que Y_b supporte X_a. Dans ce papier, nous appelons l'arc-consistance AC lorsque toutes les contraintes sont binaires, et GAC dans le cas général.

Définition 2 (Fermeture)**.** *Soit $\mathcal{N}(\mathscr{X}, \mathscr{D}, \mathscr{C})$ un réseau de contraintes, Φ une consistance locale (e.g.,* AC*) et $\mathcal{C}$ un ensemble de contraintes $\subseteq \mathscr{C}$. $\Phi(\mathscr{D}, \mathcal{C})$ est la* fermeture *de $\mathscr{D}$ pour Φ sur $\mathcal{C}$,* i.e.*, l'ensemble des domaines obtenus d'après $\mathscr{D}$ où $\forall X$, toute valeur $a \in \mathrm{dom}(X)$ qui n'est pas Φ-consistante par rapport à une contrainte de $\mathcal{C}$ a été supprimée.*

En ce qui concerne GAC et pour la plupart des consistances, la fermeture est unique. Dans les solveurs de contraintes, un *propagateur* est associé à chaque contrainte afin d'appliquer GAC ou des formes de consistance plus faibles que GAC. Les consistances plus fortes que GAC [9, 6] nécessitent la prise en compte de plus d'une contrainte simultanément pour être appliquées. Cela les a exclues de la plupart des outils de résolution de PPC jusqu'à maintenant.

2.2 Consistance locales fortes

Dans cet article, nous nous intéressons aux consistances de domaine [9], qui suppriment des valeurs des domaines, laissant inchangée la structure du réseau.

Tout d'abord, concernant les réseaux binaires, le concept générique qui capture de nombreuses consistances locales est la (i, j)-consistance [11]. Un réseau de contraintes binaires est (i, j)-consistant ssi il

n'existe pas de domaine vide et si toutes les instanciations consistantes de i variables peuvent être étendues à une instanciation consistante impliquant j variables supplémentaires. Ainsi, l'AC est la $(1,1)$ consistance.

Un réseau de contraintes binaires $\mathcal{N}$ qui n'a pas de domaine vide est *Path Consistent* (PC) ssi il est $(2,1)$-consistant. Il est *Path Inverse Consistent* (PIC) [12] ssi il est $(1,2)$-consistant. Il est *Restricted Path Consistent* (RPC) [3] ssi il est $(1,1)$-consistant et, pour toutes les valeurs a ayant une extension consistante unique b à une variable, (a,b) forme une instanciation $(2,1)$-consistante. $\mathcal{N}$ is *Max-Restricted Path Consistent* (Max-RPC) [8] ssi il est $(1,1)$-consistant et pour chaque valeur X_a et chaque variable $Y \in \mathscr{X} \backslash X$, une extension Y_b de X_a est $(2,1)$-consistante (peut être étendue à une troisème variable). $\mathcal{N}$ est *Singleton Arc-Consistent* (SAC) [9] si chaque valeur est SAC : une valeur X_a est SAC ssi le sous problème obtenu en affectant a à X peut être rendu AC (le principe est similaire au *shaving* qui ne s'applique qu'aux bornes des domaines). $\mathcal{N}$ is *Neighborhood Inverse Consistent* (NIC) [12] ssi n'importe quelle affectation consistante de la valeur a à la variable $X \in \mathscr{X}$ peut être étendue à une instanciation consistante de toutes les variables dans le voisinage de X (voisinage relatif au graphe des contraintes).

Concernant les réseaux non binaires, l'arc-consistance relationnelle et la consistance de chemin relationnelle [10] (relAC et relPC) fournissent les concepts utiles pour étendre les consistances locales des réseaux binaires au cas non binaire. $\mathcal{N}$ est un réseau relAC ssi toute instanciation consistante de toutes les variables sauf une dans une contrainte peut être étendue à cette dernière variable en satisfaisant la contrainte. $\mathcal{N}$ est relPC ssi toute instanciation consistante de toutes les variables sauf une relativement à une paire de contraintes peut être étendue à la dernière variable en satisfaisant ensemble les deux contraintes. À partir de ces notions, de nouvelles consistances pour les réseaux de contraintes généraux inspirées des définitions de PC, PIC et Max-RPC ont été proposées [6]. Enfin, des résultats intéressants ont été obtenus en utilisant la consistance PWC. Un réseau est *Pairwise Consistent* (PWC) [13] ssi aucune relation (instanciations autorisées) n'est vide et si toute instanciation consistante d'une contrainte C peut être étendu de manière consistante à n'importe qu'elle autre contrainte ayant des variables en commun avec C.

Il est possible d'appliquer simultanément PWC et GAC. Cela amène d'autres notions : *Relationally Path Inverse Consistency* (relPIC) and *Pairwise Inverse Consistency* (PWIC) [19].

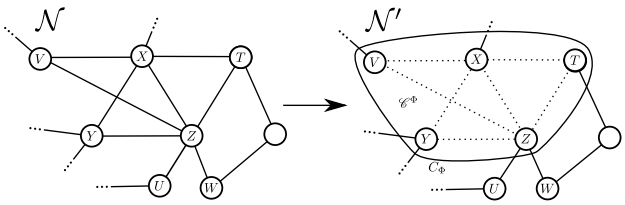

FIG. 1 – Une contrainte globale C_Φ, utilisée pour appliquer une consistance forte sur un sous-ensemble de contraintes $\mathscr{C}^\Phi$. $\mathcal{N}'$ est le réseau obtenu en remplaçant $\mathscr{C}^\Phi$ par la nouvelle contrainte globale.

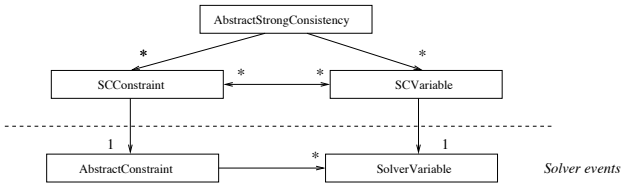

FIG. 2 – Diagramme d'intégration des consistances fortes dans des solveurs événementiels. Les flèches "→" représentent des relations d'agrégation.

3 Une contrainte globale pour les consistances de domaine

Cette section présente un schéma générique (orienté objet) dont le but est d'intégrer les consistances de domaine dans les solveurs de contraintes. Nous montrons aussi sa spécialisation à la consistance Max-RPC. Étant donnée une consistance locale Φ, le principe est de traiter le sous-ensemble de contraintes $\mathscr{C}^\Phi$ sur lequel Φ doit être appliquée. Ce traitement est effectué au sein d'une nouvelle contrainte globale C_Φ, ajoutée au réseau. Les contraintes $\mathscr{C}^\Phi$ sont alors connectées à C_Φ au lieu d'être inclues dans le réseau de contraintes (cf. Figure 1). Les événements relatifs aux contraintes de $\mathscr{C}^\Phi$ sont gérés dans un monde clos, indépendamment de la file de propagation du solveur.

3.1 Un schéma générique

Comme le montre la figure 2, `AbstractStrongConsistency` est la classe abstraite qui sera spécialisée pour implémenter C_Φ, la contrainte globale permettant d'appliquer un niveau de consistance consistance Φ. Le réseau de contraintes correspondant aux contraintes $\mathscr{C}^\Phi$ est stocké à l'intérieur de la contrainte globale. Ainsi, l'intégration de consistances locales au sein d'un solveur événementiel est souple et aisée à mettre en œuvre.

Les contraintes et les variables du réseau d'origine sont encapsulées afin de reconstruire le graphe de contraintes qui correspond aux contraintes $\mathscr{C}^\Phi$,

par l'intermédiaire des classes `SCConstraint` (Strong Consistency Constraint) et `SCVariable` (Strong Consistency Variable). Dans la figure 1, dans $\mathcal{N}'$ toutes les contraintes de $\mathscr{C}^\Phi$ sont déconnectées des variables du solveur. Les variables actives dans la contrainte globale sont encapsulées dans des `SCVariable`s, et les contraintes dans des `SCConstraint`s. Dans $\mathcal{N}'$, la variable Z est connectée aux contraintes C_{UZ}, C_{WZ} et C_Φ du point de vue du solveur. Dans C_Φ, la `SCVariable` Z est connectée via les `SCConstraint`s en pointillés aux `SCVariable`s T, V, X et Y.

3.1.1 Liaison des contraintes

Il est nécessaire d'identifier un dénominateur commun minimal parmi l'ensemble des consistances locales, qui pourra être implémenté en utilisant les services standard fournis par les contraintes du solveur. Dans la figure 2, cela est matérialisé par la classe abstraite `AbstractSolverConstraint`. Dans les solveurs de contraintes (*e.g.*, événementiels), les contraintes sont associées à des propagateurs. Certaines consistances locales, comme SAC, peuvent être directement implémentées à l'aide de propagateurs. Cependant, dans le cas général, les concepts génériques communs à l'ensemble des consistances sont la (i, j)-consistance, relAC et relPC (cf. section 2.2). Aussi, ces consistances sont davantage reliées aux notions d'*instanciation autorisée* et d'*instanciation valide*. Il est nécessaire de pouvoir itérer sur ces instanciations. En outre, les algorithmes qui ont une complexité en temps optimale mémorisent généralement les instanciations qui ont déjà été considérées. Cela implique le fait que les itérateurs délivrent les instanciations toujours dans le même ordre (le plus souvent lexicographique), et qu'il soit possible de démarrer l'itération à partir d'une instanciation quelconque.

Les itérateurs peuvent être implémentés à l'aide de *constraint checkers*. Un constraint checker teste si une instanciation donnée est autorisée par la contrainte, ou si elle ne l'est pas. Cependant, pour de nombreuses contraintes, des itérateurs plus efficaces peuvent être utilisés. Notre schéma propose la possibilité de spécialiser – ou pas – ces itérateurs, via les méthodes `firstSupp` et `nextSupp`. `AbstractSolverConstraint`, une sous-classe de la classe abstraite des contraintes du solveur, spécifie ces méthodes.

Itérateurs génériques. Les services `firstSupp` et `nextSupp` ne sont généralement pas disponibles par défaut dans les solveurs. Nous proposons une implémentation générique basée sur des *constraint checkers*. Cette implémentation est réalisée via un `Adapter` qui spécialise la superclasse `AbstractSolverConstraint`. Cela permet de gérer n'importe quelle contrainte du solveur avec un checker, comme le montre la figure 3.

Algorithm 1: nextSupport(C, I_{fixed}, I) : Instantiation

```
1 I_next ← nextValid(scp(C)\scp(I_fixed), I) ;
2 tant que I_next ≠ ⊥ ∧ ¬check(C, I_next ∪ I_fixed) faire
3 |  I_next ← nextValid(scp(C)\scp(I_fixed), I_next) ;
4 retourner I_next;
```

Algorithm 2: firstSupport(C, I_{fixed}) : Instantiation

```
1 I ← firstValid(scp(C)\scp(I_fixed)) ;
2 si check(C, I ∪ I_fixed) alors retourner I;
3 sinon retourner nextSupp(C, I_fixed, I);
```

Aucune modification n'est donc requise concernant les contraintes du solveur. Notre implémentation des méthodes `firstSupp` et `nextSupp` est donnée par les algorithmes 1 et 2. La complexité en temps est $O(d^{|\text{scp}(C)|-|I_{fixed}|})$. Elle est incrémentale : le processus complet d'itération sur toutes les instanciations autorisées et valides en appelant `firstSupp` et `nextSupp` avec le même paramètre I_{fixed} a la même complexité. Ces fonctions génériques sont principalement adaptées à des contraintes de dureté faible et de basse arité.

Itérateurs spécialisés. Pour certaines contraintes, *e.g.,* les contraintes définies par l'utilisateur, des algorithmes plus efficaces peuvent être utilisés pour `firstSupp` et `nextSupp` (c'est le cas par exemple pour des contraintes arithmétiques, ou encore des contraintes de table positives [5]). Ces algorithmes peuvent ne pas utiliser de constraint checker. La classe `AbstractSolverConstraint` spécialise `SolverConstraint` (cf. Figure 3). Ainsi, il est alors suffisant de spécialiser `AbstractSolverConstraint` pour définir de tels itérateurs.

Utilisation de propagateurs. Certaines consistances fortes comme la consistance de chemin (Path Consistency) peuvent être implémentées directement avec des propagateurs [14]. Notre paradigme permet de réaliser ces implémentations : les propagateurs des contraintes du solveur demeurent disponibles.

3.1.2 Liaison des variables.

Lier les variables est simple, car notre approche ne requiert que des opérations basiques sur les domaines, *i.e.*, itérer sur les valeurs du domaine courant. La classe `SCVariable` permet de représenter le réseau de contraintes $(\text{scp}(C_\Phi), \mathscr{D}, \mathscr{C}^\Phi)$. Un lien est conservé avec les variables du solveur pour effectuer les les modifications des domaines.

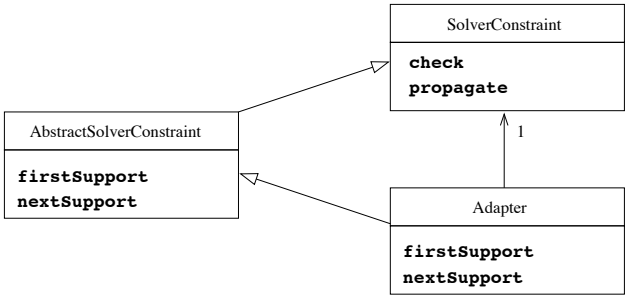

FIG. 3 – Une implementation générique des fonctions d'itération sur des supports, étant données les contraintes fournies par un solveur. Les flèches "—▷" représentent des relations de spécialisation.

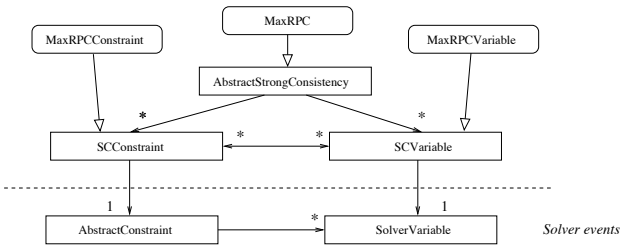

FIG. 4 – Diagramme d'intégration de MaxRPC dans un solveur événementiel.

3.1.3 Heuristiques basées sur le degré des variables.

Certaines heuristiques de choix de variables pour les réseaux binaires, comme $dom/ddeg$ ou $dom/wdeg$, sont liées à la structure du graphe de contraintes. Comme les contraintes $\mathscr{C}^\Phi$ ne sont plus connectées au modèle, elle ne sont plus prises en compte par ces heuristiques. Afin de pallier ce problème dans notre implémentation, nous avons permis à ces heuristiques de connaître directement le score d'une variable (en terme de degré dans le graphe de contraintes) via une requête à la classe `AbstractStrongConsistency`. La contrainte globale est, elle, capable de connaître le degré de chaque variable dans le réseau $\mathscr{C}^\Phi$ (c'est une donnée).

3.2 Une spécialisation concrète : Max-RPC

La figure 4 décrit la spécialisation de notre modèle pour coder la consistance locale forte Max-RPC [8], relative à des réseaux binaires. La classe `MaxRPC` définit la contrainte globale qui sera fournie à l'utilisateur. Elle étend la classe abstraite `AbstractStrongConsistency`, afin d'implémenter l'algorithme de propagation de Max-RPC. Cette consistance locale requiert de connaître les 3-cliques du graphe de contraintes, afin de tester les extensions des instanciations consistantes à n'importe quelle troisième variable. Les classes `SCConstraint` et `SCVariable` sont étendues afin de manipuler efficacement les 3-cliques.

4 Un nouvel algorithme à gros grain pour Max-RPC

Cette section présente Max-RPC^{rm}, un nouvel algorithme à gros grain pour Max-RPC, utilisé en section 5 pour expérimenter notre approche. Cet algorithme exploite des structures de données stables au retour arrière (backtrack-stable) inspirées d'AC-3^{rm} [15]. rm signifie *résidu multi-directionnel*; un résidu est un support qui a été stocké pendant l'exécution de la procédure prouvant qu'une valeur était AC. Lors des appels suivants, cette procédure teste simplement si ce support est toujours valide avant d'en chercher un nouveau. Ces structures de données ne nécessitent pas d'être restaurées ou ré-initialisées lors d'un retour-arrière. En conséquence, le maintien de la structure a un coût faible. Bien qu'étant en théorie non optimal dans le pire des cas, Lecoutre & Hemery ont montré dans [15] que AC-3^{rm} se comporte mieux que les algorithmes d'AC optimaux dans de nombreux cas.

4.1 L'algorithme

« À gros grain » signifie que la propagation dans l'algorithme est gérée soit au niveau d'une variable soit au niveau d'une contrainte, contrairement aux algorithmes comme AC-6 ou GAC-schema qui gèrent la propagation au niveau des valeurs.

4.1.1 Max-RPC^{rm}

Les algorithmes 3 à 6 décrivent Max-RPC^{rm} ; les lignes 6 à 8 de l'algorithme 3 et 5 à 8 de l'Algorithme 5 sont ajoutées à un algorithme AC-3^{rm} standard.

Algorithme 3. Il contient la boucle principale de l'algorithme. Il est basé sur une file contenant les variables ayant été modifiées (*i.e.*, qui ont perdu des valeurs), qui peuvent entraîner la perte des supports dans les domaines des variables voisines. Dans l'exemple de la figure 1 (où l'on ne considère que les contraintes de $\mathscr{C}^\Phi$), si la variable X est modifiée, alors l'algorithme doit tester si toutes les valeurs de T ont encore un support relativement à la contrainte C_{XT}, si toutes les valeurs de V ont un support sur C_{XV}, et de même pour Y et Z. Cela est réalisé via les lignes 4 à 7 de l'algorithme 3. La fonction `revise` décrite par l'algorithme 5 contrôle quant à elle l'existence de tels supports. Elle supprime les valeurs et renvoie **true** ssi il n'y en a pas (et donc **false** si aucune valeur n'a été supprimée).

La variable modifiée qui a été prise en compte peut aussi avoir causé la perte de supports PC pour les

Algorithm 3: MaxRPC($P = (\mathscr{X}, \mathscr{C})$, $\mathscr{Y}$)

$\mathscr{Y}$: the set of variables modified since the last call to MaxRPC

```
1  Q ← Y ;
2  while Q ≠ ∅ do
3      pick X from Q ;
4      foreach Y ∈ X | ∃C_XY ∈ C do
5          foreach v ∈ dom(Y) do
6              if revise(C_XY, Y_v, true) then
7                  Q ← Q ∪ {Y};
8      foreach (Y, Z) ∈ X^2 | ∃(C_XY, C_YZ, C_XZ) ∈ C^3
       do
9          foreach v ∈ dom(Y) do
10             if revisePC(C_YZ, Y_v, X) then
11                 Q ← Q ∪ {Y};
12         foreach v ∈ dom(Z) do
13             if revisePC(C_YZ, Z_v, X) then
14                 Q ← Q ∪ {Z};
```

Algorithm 4: revisePC(C_{YZ}, Y_a, X) : boolean

Y: the variable to revise because PC supports in X may have been lost

```
1  if pcRes[C_YZ, Y_a][X] ∈ dom(X) then
2      return false ;
3  b ← findPCSupport(Y_a, Z_{res[C_YZ, Y_a]}, X) ;
4  if b = ⊥ then
5      return revise(C_YZ, Y_a, false) ;
6  pcRes[C_YZ, Y_a][X] ← b ; return false;
```

contraintes situées du côté opposé de la 3-clique. Dans la figure 1, si X est modifiée, alors les supports de V et Z sur C_{VZ}, les supports de Y et Z sur C_{YZ} et les supports de T et Z sur C_{TZ} doivent être testés. Cela est réalisé par les lignes 8 à 14 de l'algorithme 3 et par la fonction `revisePC` (Algorithme 4).

Algorithme 5. Il itère sur les supports de la valeur X_a à réviser, (ligne 3 et 17), dans la recherche d'une instanciation PC $\{X_b, Y_a\}$. La consistance de chemin de l'instanciation est testée par un appel à `findPCSupport` (Algorithme 6) sur chaque variable Z qui forme une 3-clique avec X et Y. `findPCSupport` retourne soit un support de l'instanciation $\{X_b, Y_a\}$ dans le domaine de Z, soit la valeur $\perp$ si aucun support ne peut être trouvé. Si aucun support PC pour Y_a ne peut être trouvé, la valeur est supprimée et la fonction retourne **true**.

4.1.2 Résidus

La fonction `revise` teste tout d'abord la validité du résidu (ligne 1). Les résidus sont stockés dans la

Algorithm 5: revise(C_{XY}, Y_a, *supportIsPC*) : boolean

Y_a: the value of Y to revise against C_{XY} – supports in X may have been lost

supportIsPC: **false** if one of $pcRes[C_{XY}, Y_a]$ is no longer valid

```
1  if supportIsPC ∧ res[C_XY, Y_a] ∈ dom(X) then
2      return false ;
3  b ← firstSupp(C_XY, {Y_a})[X] ;
4  while b ≠ ⊥ do
5      PConsistent ← true ;
6      foreach Z ∈ X | (X, Y, Z) form a 3-clique do
7          c ← findPCSupport(Y_a, X_b, Z) ;
8          if c = ⊥ then
9              PConsistent ← false ;
10             break ;
11         currentPcRes[Z] ← c ;
12     if PConsistent then
13         res[C_XY, Y_a] ← b ; res[C_XY, X_b] ← a ;
14         pcRes[C_XY, Y_a] ← currentPcRes ;
15         pcRes[C_XY, X_b] ← currentPcRes ;
16         return false ;
17     b ← nextSupp(C_XY, {Y_a}, {X_b, Y_a})[X] ;
18 remove a from dom(Y) ;
19 return true ;
```

Algorithm 6: findPCSupport(X_a, Y_b, Z) : value

```
1  c1 ← firstSupp(C_XZ, {X_a})[Z] ;
   c2 ← firstSupp(C_YZ, {Y_b})[Z] ;
2  while c1 ≠ ⊥ ∧ c2 ≠ ⊥ ∧ c1 ≠ c2 do
3      if c1 < c2 then
4          c1 ← nextSupp(C_XZ, {X_a}, {X_a, Z_{c2−1}})[Z] ;
5      else
6          c2 ← nextSupp(C_YZ, {Y_b}, {Y_b, Z_{c1−1}})[Z] ;
7  if c1 = c2 then return c1 ;
8  return ⊥ ;
```

structure de données globale $res[C, X_a]$ (complexité spatiale en $O(ed)$).

L'algorithme utilise aussi des résidus pour les supports PC, stockés dans la structure *pcRes* (complexité en espace $O(cd)$, où c est le nombre de 3-cliques du réseau). L'idée est d'associer le résidu trouvé par la fonction `revise` avec la valeur PC trouvée pour chaque troisième variable dans la 3-clique. De cette façon, à la fin du processus, $(X_a, res[C_{XY}, X_a], pcRes[C_{XY}, X_a][Z])$ forme une 3-clique dans la microstructure du graphe de contraintes pour chaque 3-clique (X, Y, Z) du réseau, et ce pour tout $a \in \text{dom}(X)$.

Dans l'exemple de la figure 5, à la fin du processus, $res[C_{XY}, X_a] = b$, $pcRes[C_{XY}, X_a][Z] = a$,

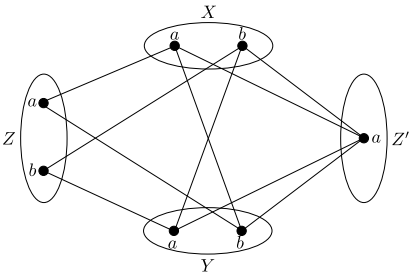

FIG. 5 – Exemple avec deux 3-cliques (micro-structure)

$pcRes[C_{XY}, Y_a][Z'] = a$, et ainsi de suite. L'algorithme exploite la bidirectionalité des contraintes : si Y_b est un support pour X_a avec $\{Z_a, Z'_a\}$ comme supports PC, alors X_a est aussi un support pour Y_b avec les mêmes supports PC (cf. lignes 14 et 15 de l'algorithme 5).

Si un support PC obsolète est détecté (ligne 1 de `revisePC`) alors un autre support est recherché. Si aucun ne peut être trouvé, alors le support courant de la valeur courante n'est pas PC, et un autre doit être trouvé (ligne 5 de 4).

4.2 Light-MaxRPC^{rm}

En considérant les consommations élevées en mémoire et en temps causée par la structure de données *pcRes* et les appels successifs de la fonction `revisePC`, nous proposons de modifier l'algorithme 3 en supprimant la boucle **foreach do** des lignes 8 à 14. `revisePC` et *pcRes* n'ont alors plus d'utilité et peuvent être supprimées, ainsi que les lignes 11, 14 et 15 de l'algorithme 5 (il s'agit des parties grisées des algorithmes). L'algorithme ainsi obtenu réalise une approximation de Max-RPC, qui est néanmoins plus forte qu'AC. Il assure que toutes les valeurs qui n'étaient pas Max-RPC avant l'appel à Light-MaxRPC^{rm} sont filtrées.

La consistance qui résulte de l'application de cet algorithme n'est pas monotone. Elle dépend de l'ordre dans lequel les variables sont extraites de $\mathscr{Q}$. Nos expérimentations en section 5 montrent empiriquement que le filtrage de Light-MaxRPC n'est que légèrement plus faible que celui de Max-RPC sur des problèmes aléatoires, alors qu'il permet des gains conséquents en complexité spatiale et temporelle.

4.3 Complexité

Dans la suite de l'article, c désigne le nombre de 3-cliques du graphe de contraintes ($c \leq \binom{n}{3} \in O(n^3)$), g le degré maximal d'une variable, et s le nombre maximal de 3-cliques qui partagent une même (unique) contrainte. Si le graphe de contraintes n'est pas vide (au moins deux variables et une contrainte) alors $s < g < n$. Les complexités sont décrites en termes de tests de contraintes, qui sont supposés constants en temps.

Proposition 1. *Après une phase d'initialisation en $O(eg)$,* MaxRPC^{rm} *a une complexité temporelle dans le pire des cas en $O(ed^3 + csd^4)$.*

Squelette de preuve. La phase d'initialisation détecte et lie les 3-cliques aux contraintes et variables, en $O(eg)$.

La boucle principale de l'algorithme dépend de la file de variables $\mathscr{Q}$. Sachant que les variables sont ajoutées à $\mathscr{Q}$ lorsqu'elles sont modifiées, elles peuvent être ajoutées chacune au plus d fois, impliquant que cette boucle soit exécutée $O(nd)$ fois. Cette propriété reste vraie lorsque l'algorithme est appelé plusieurs fois entre chaque appel, en supprimant une valeur du domaine d'une variable à chaque fois. Cet algorithme est incrémental, notamment lorsqu'on le maintient au sein d'un processus de recherche systématique d'une solution. On considère séparément les deux parties de la boucle principale.

1. La boucle **foreach do** (lignes 4 à 7 de l' algorithme 3).

   Elle peut être exécutée $O(g)$ fois. Dans le pire des cas le réseau entier est exploré de façon homogène, donc la boucle est amortie avec le facteur $O(n)$ de la boucle principale pour une complexité globale en $O(e)$. La boucle **foreach do** des lignes 5 à 7 implique $O(d)$ appels à `revise` (soit au total $O(ed^2)$).

   La fonction `revise` (Algorithme 5) appelle en premier `firstSupp` ($O(d)$ avec l'algorithme 2 sans hypothèse sur la nature des contraintes). La boucle **while do** est réalisée $O(d)$ fois. Les appels à `nextSupp` (Ligne 17) font partie de la boucle. La boucle **foreach do** (lignes 6 à 11) peut être réalisée $O(s)$ fois. Elle implique un appel à `findPC-Support`, en $O(d)$. En conséquence, `revise` est en $O(d + sd^2)$. La complexité globale de cette partie est donc $O(ed^3 + esd^4)$. Le facteur $O(es)$ est amorti en $O(c)$, soit $O(ed^3 + cd^4)$.

2. La boucle **foreach do** (lignes 8 à 14 de l'algorithme 3).

   Le nombre de pas de cette boucle est amorti avec la boucle principale en $O(cd)$. Chaque pas exécute $O(d)$ appels à `revisePC`, dont la complexité dans le pire des cas est plafonnée par `revise` à la ligne 3 de l'algorithme 4. Cette partie de l'algorithme est donc en $O(cd^2.(d + sd^2)) = O(csd^4)$.

La phase d'initialisation et les deux parties entraînent une complexité temporelle en $O(eg + ed^3 + csd^4)$. □

Avec un graphe de contraintes complet, la complexité en temps est $O(n^4d^4)$. Elle peut être comparée à la borne inférieure $O(eg + ed^2 + cd^3)$ ($O(n^3d^3)$ pour un graphe complet) d'un algorithme optimal pour Max-RPC. Si `revise` est appelée à cause de la suppression d'une valeur qui n'apparaît dans aucun support, alors sa complexité est réduite à (sd). En pratique, cela arrive très régulièrement, ce qui explique le bon comportement pratique de notre algorithme. Comme Light-MaxRPC^{rm} supprime la deuxième partie de l'algorithme, sa complexité est $O(eg + ed^3 + cd^4)$. Pour les deux variantes de l'algorithme, la phase d'initialisation en $O(eg)$ est réalisée avant le premier appel à l'algorithme 3, et seulement une fois lorsqu'on maintient Max-RPC pendant la recherche.

Enfin, on peut comparer ces complexités pour maintenir un niveau de complexité strictement supérieur à AC avec celle d'AC-3^{rm} (en $O(n^2d^3)$ pour un graphe complet).

5 Expérimentations

Nous avons implémenté le diagramme de la Figure 4 à l'aide du solveur Choco [1], en utilisant les algorithmes pour Max-RPC décrits précédemment. Dans nos tests, MaxRPC^{rm} et sa variante allégée sont comparés à AC-3^{rm}. Sur les figures, chaque point est le résultat médian relatif à 50 problèmes binaires aléatoires ayant des caractéristiques variables. Un problème aléatoire est caractérisé par un quadruplet (n, d, γ, t), dont les éléments représentent respectivement le nombre de variables, le nombre de valeurs, la densité du graphe[1] et la dureté des contraintes[2].

Pre-processing. La figure 6 compare le temps et la mémoire consommés lors d'une propagation initiale sur des problèmes assez gros (200 variables et 30 valeurs). Dans ces tests, les contraintes formant des 3-cliques sont inclues dans la contrainte globale. Une basse densité du graphe de contraintes entraîne un petit nombre de 3-cliques. Les résultats sont cohérents avec les complexités théoriques. La faible consommation en espace de Light-Max-RPC^{rm} est mise en évidence par les résultats, comparativement à Max-RPC. Si l'on compare ces résultats à ceux de [9], appliquer une consistance forte via une contrainte globale n'entraîne pas de surcoût significatif. L'algorithme AC-3^{rm} du solveur Choco utilise des structures d'allocation paresseuses, ce qui explique la chute de consommation de mémoire après le seuil.

[1] La densité est la proportion de contraintes dans le graphe par rapport au nombre maximum possible de contraintes : $\gamma = e/\binom{n}{2}$.

[2] La dureté est la proportion d'instanciations interdites pour chaque contrainte.

Max-RPC vs AC. La Figure 7 décrit les résultats obtenus avec un algorithme de recherche systématique, où différents niveaux de consistance locale sont maintenus pendant la recherche. L'heuristique de choix de variables est *dom/ddeg* (le processus de pondération des contraintes avec *dom/wdeg* n'est pas défini lorsque plus d'une contrainte peut participer à la suppression de toutes les valeurs d'un domaine). Nous utilisons le problème $(105, 20, 5\%, t)$ comme référence (courbes en haut à gauche) et nous augmentons successivement le nombre de valeurs (en haut à droite), de variables (en bas à gauche) et la densité (en bas à droite). Les résultats dans [8] ont montré que maintenir Max-RPC (à l'aide d'un solveur dédié) est intéressant pour des problèmes de grande taille et de faible densité, comparativement à maintenir l'AC. Les algorithmes d'AC utilisés dans [8] sont maintenant assez obsolètes comparés à AC-3^{rm}. Nos résultats montrent que les nouveaux algorithmes que nous proposons pour Max-RPC sont compétitifs par rapport à AC-3^{rm}, à la fois en termes de nœuds et de temps, sur des problèmes de grande taille et de faible densité.

Combiner des consistances locales. Le tableau 1 montre l'intérêt de la nouvelle possibilité qu'offre naturellement notre approche : combiner deux niveaux de consistance différents pour résoudre un même réseau de contraintes. La première ligne correspond au résultat médian sur 50 instances de problèmes $(35, 17, 44\%, 31\%)$ forcés à être satisfiables. A l'inverse, les *seeds* de $(105, 20, 5\%, 65\%)$ sont choisis de telle sorte que toutes les instances soient insatisfiables. Le premier problème est mieux résolu en utilisant AC-3^{rm}, le deuxième est mieux résolu avec Max-RPC. La troisième ligne décrit les résultat d'un problème qui concatène les deux précédents, en les liant par une unique contrainte additionnelle. Sur les deux dernières colonnes, l'AC est maintenue sur les parties denses du graphe de contraintes, et Max-RPC sur le reste du réseau. L'heuristique de choix de variable *dom/ddeg* entraîne le fait que le problème dense et satisfiable soit résolu en premier, et ensuite que le solveur *thrashe* pour prouver qu'il n'existe pas de solution à cause de la partie du réseau de faible densité.

Combiner les deux niveaux de consistance permet d'améliorer la résolution, ce qui souligne l'intérêt de notre approche. Les deux dernières lignes présentent les résultats obtenus avec de plus gros problèmes, qui confirment les précédents.

6 Conclusion et perspectives

Dans cet article, nous avons présenté un schéma générique pour appliquer des consistances fortes lorsqu'on utilise les outils standard de PPC. Cette ap-

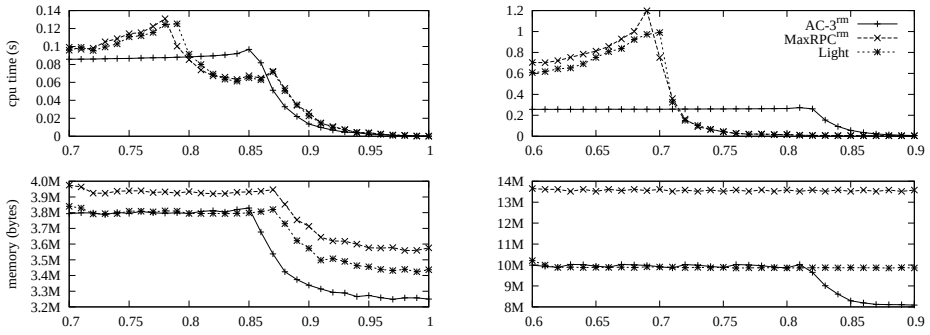

FIG. 6 – Propagation initiale : temps cpu et mémoire vs dureté, sur des problèmes aléatoires binaires homogènes (200 variables, 30 valeurs).

		AC	MaxRPC	Light	AC+MaxRPC	AC+Light
$(35, 17, 44\%, 31\%)$	*cpu (s)*	**6.1**	25.6	11.6	non	non
	nodes	21.4k	5.8k	8.6k	applicable	applicable
$(105, 20, 5\%, 65\%)$	*cpu (s)*	20.0	19.4	**16.9**	non	non
	nodes	38.4 k	20.4 k	19.8 k	applicable	applicable
$(35, 17, 44\%, 31\%)$	*cpu (s)*	96.8	167.2	103.2	90.1	**85.1**
$+(105, 20, 5\%, 65\%)$	*nodes*	200.9k	98.7k	107.2k	167.8k	173.4k
$(110, 20, 5\%, 64\%)$	*cpu (s)*	73.0	60.7	**54.7**	non	non
	nodes	126.3k	54.6k	56.6k	applicable	applicable
$(35, 17, 44\%, 31\%)$	*cpu (s)*	408.0	349.0	272.6	284.1	**259.1**
$+(110, 20, 5\%, 64\%)$	*nodes*	773.0k	252.6k	272.6k	308.7k	316.5k

TAB. 1 – Combinaison de deux niveaux de consistance dans un même modèle.

proche permet d'utiliser simultanément plusieurs niveaux distincts de consistance locale pour résoudre un même problème. L'intérêt de cette possibilité est confirmé par nos résultats expérimentaux. De plus, notre paradigme ne pose aucune hypothèse sur la nature et l'arité des contraintes, et permet donc une application des consistances fortes à de nombreux problèmes. Enfin, nous avons proposé Max-RPC^{rm}, un nouvel algorithme « à gros grain » pour Max-RPC.

Nos travaux futurs viseront à appliquer ce paradigme sur des problèmes pratiques concrets, en utilisant d'autres consistances fortes. En outre, étudier les critères pertinents pour décomposer automatiquement un réseau de contraintes afin d'y appliquer différents niveaux de consistance constitue une perspective très intéressante.

Références

[1] Choco : An open source Java CP library. `http://choco.emn.fr/`, 2008.

[2] N. Beldiceanu, M. Carlsson, and J.-X. Rampon. Global constraint catalog. Technical Report T2005-08, SICS, 2005.

[3] P. Berlandier. Improving domain filtering using restricted path consistency. In *Proc. IEEE-CAIA'95*, 1995.

[4] C. Bessière and J.-C. Régin. Enforcing arc consistency on global constraints by solving subproblems on the fly. In *CP*, pages 103–117, 1999.

[5] C. Bessière and J.-C. Régin. Arc consistency for general constraint networks : preliminary results. In *Proc. IJCAI'97*, pages 398–404, 1997.

[6] C. Bessière, K. Stergiou, and T. Walsh. Domain filtering consistencies for non-binary constraints. *Artificial Intelligence*, 172(6-7) :800–822, 2008.

[7] C. Bessière and P. van Hentenryck. To be or not to be... a global constraint. In *Proc. CP'03*, pages 789–794. Springer, 2003.

[8] R. Debruyne and C. Bessière. From restricted path consistency to max-restricted path consistency. In *Proc. CP'97*, pages 312–326, 1997.

[9] R. Debruyne and C. Bessière. Domain filtering consistencies. *Journal of Artificial Intelligence Research*, 14 :205–230, 2001.

[10] R. Dechter and P. van Beek. Local and global relational consistency. *Theoretical Computer Science*, 173(1) :283–308, 1997.

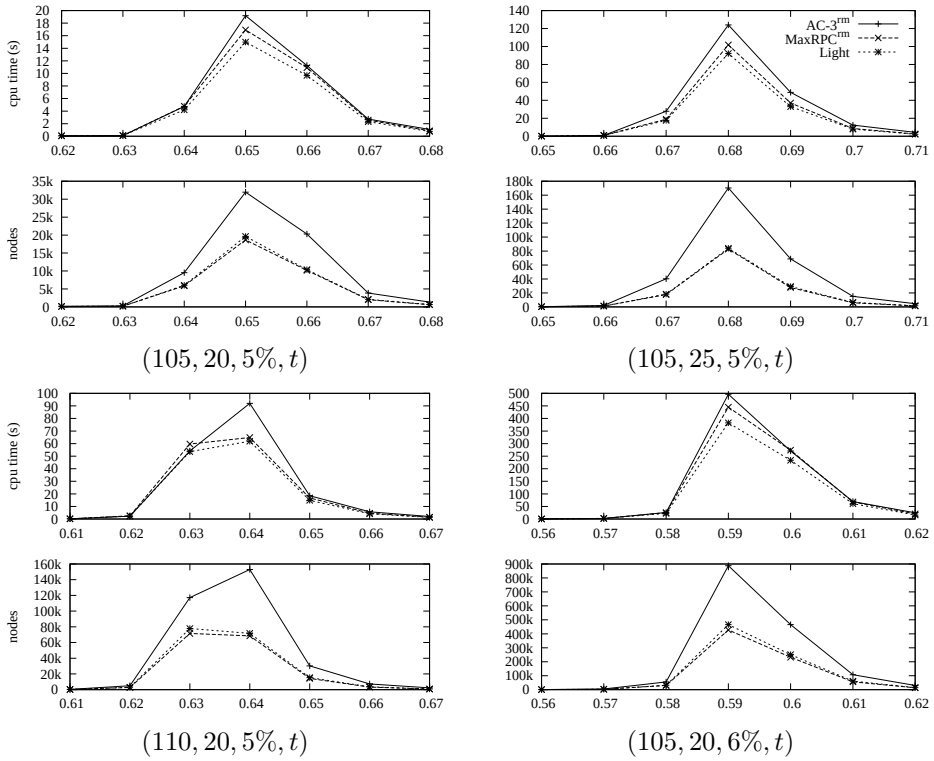

FIG. 7 – Recherche d'une solution : temps cpu et nœuds vs dureté, sur des problèmes aléatoires binaires homogènes (105-110 variables, 20-25 valeurs).

[11] E.C. Freuder. A sufficient condition for backtrack-free search. *Journal of the ACM*, 29(1) :24–32, 1982.

[12] E.C. Freuder and C.D. Elfe. Neighborhood inverse consistency preprocessing. In *Proc. AAAI'96*, pages 202–208, 1996.

[13] P. Janssen, P. Jégou, B. Nouguier, and M.C. Vilarem. A filtering process for general constraint-satisfaction problems : achieving pairwise-consistency using an associated binary representation. In *Proc. of IEEE International Workshop on Tools for Artificial Intelligence*, pages 420–427, 1989.

[14] C. Lecoutre, S. Cardon, and J. Vion. Path Consistency by Dual Consistency. In *Proc. CP'07*, pages 438–452, 2007.

[15] C. Lecoutre and F. Hemery. A study of residual supports in arc consistency. In *Proc. IJCAI'07*, pages 125–130, 2007.

[16] J.-C. Régin. A filtering algorithm for constraints of difference in CSPs. In *Proc. AAAI'94*, pages 362–367, 1994.

[17] J-C. Régin, T. Petit, C. Bessière, and J.-F. Puget. An original constraint based approach for solving over constrained prolems. In *Proc. CP'00*, pages 543–548, 2000.

[18] K. Stergiou. Heuristics for dynamically adapting propagation. In *ECAI*, pages 485–489, 2008.

[19] K. Stergiou and T. Walsh. Inverse consistencies for non-binary constraints. In *Proc. ECAI'06*, volume 6, pages 153–157, 2006.

[20] P. van Hentenryck, Y. Deville, and C.M. Teng. A generic arc-consistency algorithm and its specializations. *Artificial Intelligence*, 57 :291–321, 1992.