Heuristique de révision et contraintes hétérogènes

Julien Vion

Heuristiques de révision et contraintes hétérogènes

Julien Vion

Université de Lille Nord de France, UVHC,
LAMIH CNRS UMR 8201,
59313 Valenciennes Cedex 9, France.
julien.vion@univ-valenciennes.fr

Résumé

La plupart des solveurs de contraintes utilisent une méthode de propagation basée sur l'algorithme générique AC-5 [24]. Le principe d'AC-5 est d'abstraire la notion de *révision de contrainte*. Chaque contrainte dispose d'un *propagateur*, doté d'un algorithme de propagation spécifique, de complexité et performance variables.

Plusieurs travaux ont, par le passé, montré que l'ordre dans lequel les contraintes sont révisées ont un impact non négligeable sur les performances de la propagation [27, 8, 22, 1]. Cependant, la plupart de ces articles se basent sur l'utilisation de propagateurs homogènes, en particulier pour des contraintes binaires définies en extension.

Cet article présente différentes idées et techniques permettant un ordonnancement fin des contraintes au sein d'un système de propagation.

Abstract

Most constraint solvers use the general AC-5 scheme to handle constraint propagation [24]. AC-5 generalizes the concept of *constraint revision*. Each constraint type can thus be shipped with its own revision algorithm, with various complexities and performances.

Previous papers showed that the order in which constraints are revised have a non-negligible impact on performances of propagation [27, 8, 22, 1]. However, most of the ideas presented on these papers are based on the use of homogeneous propagators for binary constraints defined in extension.

This paper give ideas to handle heterogeneous constraints in a general revision schedule.

1 Introduction

La méthode de résolution la plus courante pour résoudre un CSP consiste à effectuer successivement des étapes d'*hypothèse*, de *propagation* et éventuellement de *retour-arrière* (*backtrack*). La propagation consiste généralement à établir la *consistance d'arc*, c'est-à-dire à supprimer toutes les valeurs qui apparaissent inconsistantes (i.e., ne pouvant apparaitre dans aucune solution du CSP) du point de vue d'une contrainte donnée. L'algorithme obtenu, décrit ci-dessus, est appelé `MAC` (*Maintaining Arc-Consistency*). La fonction `AC` (décrite ci-après) réalise la propagation, consistant à supprimer les valeurs inconsistantes du domaine des variables. Elle peut renvoyer la valeur spéciale $\bot$ si le CN est trivialement inconsistant (i.e., le domaine d'une variable est vide, ou une contrainte ne peut plus être satisfaite), ou $\top$ si le CN est trivialement consistant (i.e., toutes les variables ont été réduites à des singletons). L'étape d'hypothèse consiste à effectuer une décision logique, ici notée δ (qui ne doit pas être trivialement impliquée par le CN pour éviter une récursion infinie), consistant le plus souvent à affecter une valeur à une variable non-singleton. Le choix de la variable à affecter peut être réalisé par une heuristique comme $^{dom}/_{ddeg}$ ou $^{dom}/_{wdeg}$ [7].

Fonction MAC($\mathcal{N}$) =

```
1 N' ← AC(N)
2 suivant N' faire
3     cas où ⊥ retourner false
4     cas où ⊤ retourner true
5     autres cas
6         Soit δ une hypothèse logique
7         retourner MAC(N' + δ) ∨ MAC(N' + ¬δ)
```

La suppression de valeurs inconsistantes du point de vue d'une contrainte donnée est appelée *révision de contrainte*. Les algorithmes de propagation comme AC-5 réalisent ces révisions jusqu'à l'obtention d'un point fixe. Dans le cas de l'arc-consistance, ce point

fixe est unique et peut être obtenu en un temps fini. Cependant, établir l'arc-consistance est NP-difficile dans le cas général (i.e., si l'on n'a aucune information sur la nature des contraintes). Pour cette raison, pendant longtemps, seuls les CSP n'impliquant que des contraintes binaires (exactement deux variables par contrainte) étaient traités. Le CSP binaire est aussi NP-complet, mais on peut établir l'arc-consistance sur un CN binaire en temps et en espace polynomiaux. Les contraintes binaires restent malheureusement bien moins expressives que les contraintes non-binaires [4].

Des dizaines d'algorithmes ont été proposés durant les 40 dernières années pour effectuer la phase de propagation : AC-1 à AC-8, plusieurs variantes d'AC-3, etc. La plupart de ces algorithmes peuvent être décomposés en un algorithme de *propagation* et un algorithme de *révision de contrainte* (NP-difficile ou limité aux contraintes binaires). Cependant, deux algorithmes, AC-5 (1992 [24]) et GAC-schema (1997 [5]), traitent la révision de contrainte de manière *abstraite* : l'idée générale d'AC-5 est de considérer que chaque contrainte du CN dispose de propriétés sémantiques pouvant être exploitées pour développer des algorithmes efficaces de révision de contraintes.

La plupart des solveurs de CSP sont basés sur AC-5, et proposent une « boite à outils » de contraintes usuelles, disposant chacune de leur propagateur [2]. Certaines de ces contraintes, définies uniquement à partir de leur propriété sémantique et pouvant impliquer un nombre arbitraire de variables sont appelées *contraintes globales* [3].

Tous les algorithmes d'arc-consistance depuis AC-3 [18], sont basés sur une *file de propagation*. Quand une valeur est supprimée du domaine d'une variable, toutes les contraintes impliquant cette variable peuvent avoir perdu leur propriété d'arc-consistance, et doivent être révisées afin de propager la valeur supprimée aux autres variables du réseau. La nature des données contenues dans la file de propagation (variables, contraintes et/ou valeurs), ainsi que l'ordre dans lequel les révisions sont effectuées ont un impact significatif sur les performances de la propagation. Plusieurs travaux ont cherché à déterminer un « bon » ordonnancement des révisions [27, 8, 1], mais se sont souvent limités aux CSP binaires et propagateurs généraux. Appliquer ces techniques de propagation à des propagateurs hétérogènes peut entrainer des comportement pathologiques : typiquement, un propagateur très lent ou très faible devrait être exécuté le moins souvent possible (on trouvera un exemple au début de la section 4.4).

Les contributions de cet article sont les suivantes :

1. établir un état de l'art d'algorithmes de propagation « à gros grain » supportant la notion de propagateur abstrait (section 3) ;
2. évaluer des structures de données permettant de gérer la file de propagation (section 4.2) ;
3. abstraire la notion d'heuristique d'ordonnancement des révisions au niveau des contraintes pour obtenir un ordre de priorité fin et adapté au propagateur de la contrainte (section 4.4).

2 Définitions et notations

Définition 1 (Réseau de contraintes, variable, domaine, contrainte)**.** Un *réseau de contraintes* $\mathcal{N}$ est un couple $(\mathscr{X}, \mathscr{C})$ constitué :
- d'un ensemble de n *variables* $\mathscr{X}$; un *domaine* $\mathrm{dom}(X)$ est associé à chaque variable $X \in \mathscr{X}$ et définit l'ensemble fini d'au plus d valeurs auxquelles X peut être *instanciée*, et
- d'un ensemble de e *contraintes* $\mathscr{C}$; chaque contrainte $C \in \mathscr{C}$ implique au plus k variables $\mathrm{vars}(C) \subseteq \mathscr{X}$; la contrainte définit les combinaisons de valeurs auxquelles ces variables peuvent être instanciées.

On appelle une combinaison de valeurs affectées à un ensemble de variables une *instanciation*. L'ensemble des contraintes impliquant une variable donnée X est noté $\mathrm{ctr}(X)$. Une contrainte peut être définie en *extension*, c'est-à-dire par une liste exhaustive d'instanciations autorisées ou interdites, ou en *intention*, c'est-à-dire par une fonction : $\prod_{X \in \mathrm{vars}(C)} \mathrm{dom}(X) \rightarrow \mathbb{B}$. Les contraintes dites *globales* définissent une propriété d'arité arbitraire que les variables qu'elle implique doivent vérifier (par exemple, *all different*).

Le problème de satisfaction de contraintes (CSP) consiste à décider si une solution au CN (c'est-à-dire une instanciation de toutes les variables satisfaisant toutes les contraintes du CN) existe. Quand toutes les contraintes peuvent être vérifiées en temps et en espace polynomiaux, le CSP est NP-complet.

Définition 2 (Arc-consistance, support)**.** Soit C une contrainte et $X \in \mathrm{vars}(C)$. La valeur $v \in \mathrm{dom}(X)$ est arc-consistante (AC) pour C ssi il existe une instanciation de $\mathrm{vars}(C)$, autorisée par C, qui instancie X à v (une telle instanciation est appelée un *support* de v pour C). C est AC ssi $\forall X \in \mathrm{vars}(C)$, $\forall v \in \mathrm{dom}(X)$, v est AC pour C.

Dans la littérature, la définition de l'arc-consistance est souvent restreinte aux CN binaires, et l'extension d'AC aux CN non-binaires est appelée AC généralisée (GAC), Hyper-AC ou consistance de domaine (*Domain Consistency*). Dans cet article, nous nous référons à AC pour les CN binaires et non binaires.

Définition 3 (Clôture)**.** Soit $\mathcal{N} = (\mathscr{X}, \mathscr{C})$ un réseau de contraintes. $\mathrm{AC}(\mathcal{N}, C)$ est la *clôture* de $\mathcal{N}$ par AC sur C, c'est-à-dire le CN obtenu à partir de $\mathcal{N}$ tel que $\forall X \in \mathrm{vars}(C)$, toutes les valeurs $v \in \mathrm{dom}(X)$ qui ne sont pas AC pour C ont été supprimées.

$\mathrm{AC}(\mathcal{N})$ est la clôture de $\mathcal{N}$ par AC, c'est-à-dire le CN obtenu à partir de $\mathcal{N}$ tel que $\forall C \in \mathscr{C}$, C a été rendu AC par clôture.

Pour tout $\mathcal{N} = (\mathscr{X}, \mathscr{C})$, pour tout $C \in \mathscr{C}$, $\mathrm{AC}(\mathcal{N}, C)$ et $\mathrm{AC}(\mathcal{N})$ sont uniques. Dans le cas général, calculer la clôture par AC d'un CN est NP-difficile. L'algorithme optimal GAC-schema admet une complexité en $O(ekd^k)$ [5].

Définition 4 (Propagateur)**.** Étant donné un CN $\mathcal{N} = (\mathscr{X}, \mathscr{C})$, le *propagateur* d'une contrainte $C \in \mathscr{C}$ est l'algorithme permettant d'obtenir $\mathrm{AC}(\mathcal{N}, C)$.

3 Algorithmes de propagation

Les algorithmes présentés dans cette section, inspirés d'AC-3 et AC-5, n'ont pas pour vocation d'être innovants, mais plutôt de présenter un état de l'art des algorithmes de propagation à gros grain. La plupart des solveurs disponibles (Choco [11], Gecode [21], Abscon [20], CSP4J [25], Eclipse…) fournissent une variante des algorithmes présentés ici, mais en proposant généralement un grain fin.

3.1 Propagateurs abstraits

Historiquement, la plupart des algorithmes d'AC ont été conçus en considérant que la seule information disponible pour une contrainte était le résultat de sa fonction de vérification (*check*). Dans ces conditions, la seule manière de d'obtenir l'arc-consistance consiste à trouver un support de chaque valeur pour chaque contrainte du CN, en énumérant toutes les instanciations possibles des variables impliquées par chaque contrainte, d'où une complexité minimale en $O(ekd^k)$ pour établir AC (en considérant un *check* en $O(k)$). AC-5 et GAC-schema proposent des hypothèses plus générales, aboutissant au total à trois niveaux d'abstraction, du plus bas au plus haut niveau :

- *Vérification de contraintes*, l'hypothèse historique ;
- *Recherche de support*, proposée par GAC-schema : chaque contrainte dispose d'un service permettant de trouver rapidement un support d'une valeur ;
- *Révision de contraintes*, proposée par AC-5.

En réalité, l'hypothèse de *recherche de support abstraite* peut s'appliquer à tous les algorithmes d'AC (sauf AC-5) sans que ceux-ci ne perdent leur spécificité. C'est dans la manière de mémoriser les supports pour obtenir des propriétés d'incrémentalité que ces algorithmes différent. On peut ainsi extraire un propagateur *général* de chacun de ces algorithmes. Un exemple est donné ci-après avec le propagateur $\texttt{revise}^{\texttt{rm}}$.

3.2 Files de propagation

Indépendamment du propagateur général, les algorithmes d'AC sont généralement classés en deux familles, en fonction de la nature des données stockées dans la file de propagation. Les algorithmes dits « à grain fin » (AC-4 à AC-7 et GAC-schema) stockent chaque *valeur* récemment supprimée dans la file de propagation, et cherchent à exploiter cette information pour éviter des calculs inutiles. Les algorithmes « à gros grain » (variantes d'AC-3 et AC-8) stockent les variables modifiées (i.e., une valeur a été supprimée de leur domaine) et/ou les contraintes impliquant celles-ci, qui doivent donc être révisées. Bien que le grain fin soit indispensable pour concevoir des algorithmes optimaux, la différence théorique reste marginale (l'algorithme à gros grain GAC-2001 admet une complexité en $O(ek^2d^k)$ [6]), et les structures de données plus simples permettent souvent d'obtenir de meilleurs résultats en pratique. Dans la suite de cette section, nous décrivons des techniques permettant de limiter le plus possible les révisions inutiles dans ce cadre des files à gros grain.

L'algorithme AC-3 de Mackworth (1977 [18]) était un algorithme à gros grain « orienté arcs », c'est-à-dire qu'il utilisait une file de propagation contenant des *arcs* constitués de couples (*Variable*, *Contrainte*). La *Variable* du couple identifie une variable dont le domaine peut ne pas être AC pour la *Contrainte*. L'algorithme de révision doit alors chercher un support pour chaque valeur de la *Variable* pour la *Contrainte*. McGregor montre que le même comportement peut être obtenu en ne stockant que les variables modifiées dans la file (1979 [19]). Cependant, si l'on travaille avec des contraintes non-binaires, une telle file ne contient pas suffisamment d'information pour éviter autant de révisions inutiles : si deux variables impliquées par la même contrainte non-binaire se trouvent dans la file, le domaine de chacune de ces variables ne devrait être contrôlé qu'une seule fois, ce qu'un algorithme orienté variables ne permet pas de faire.

Boussemart *et al.* ont proposé en 2004 une version de l'algorithme basé sur une file de variables, disposant d'une structure de données complémentaire que nous appelons *modif* afin d'émuler les avantages de la file à base d'arcs pour une propagation orientée variables [8]. Cette structure de données associe à chaque contrainte la liste des variables modifiées depuis la dernière révision de celle-ci. *modif* permet de connaitre les arcs à réviser et contient donc les mêmes informa-

Algorithme 1: updateModif($modif, \Delta, C$)

```
1 modif(C) ← ∅
2 pour chaque Y ∈ Δ, C' ∈ ctr(Y) − C faire
3     modif(C') ← modif(C') ∪ {Y}
```

Algorithme 2: AC-V($\mathcal{N}, Q, modif$) =

```
1 si Q = ∅ alors retourner N
2 sinon
3     Choisir X dans Q, Q' ← Q − X
4     pour chaque C ∈ ctr(X) | modif(C) ≠ ∅
      faire
5         (N', Δ) ← revise(C, modif(C))
6         si Δ = ⊥ alors retourner ⊥
7         updateModif(modif, Δ, C)
8         Q' ← Q' ∪ Δ
9     retourner AC-V(N', Q', modif)
```

tions qu'une file de propagation orientée arcs. Cependant, regrouper les révisions par contrainte simplifie considérablement la conception des propagateurs, particulièrement si l'on fait l'hypothèse de propagateurs abstraits. On trouve aussi des mécanismes plus sophistiqués, par exemple dans le solveur Gecode [21, 12] : l'idée des *advisors* est d'abstraire, en plus du propagateur, la détection des révisions inutiles.

L'algorithme 1 montre comment la structure est mise à jour, après qu'une contrainte C ait supprimé des valeurs dans les variables Δ. C n'a plus besoin d'être révisée pour l'instant (ligne 1), puis, pour chaque contrainte C' impliquant une variable de Δ, on indique que la variable correspondante a été modifiée (en prenant soin de ne pas altérer $modif(C)$).

Cette structure de données peut également être utilisée pour concevoir un algorithme de propagation efficace basé uniquement sur une file de contraintes. Nous décrivons donc deux algorithmes qui nous serviront de base pour nos développements, l'un centré sur une file de variables, l'autre sur une file de contraintes. Nous conservons l'idée d'abstraire la notion de révision de contraintes, comme dans l'algorithme AC-5, mais avec des files à gros grain. La gestion de la structure *modif* présentée ici améliore légèrement les versions de Boussemart *et al.* : au maximum un surcoût en $O(k)$ dans l'algorithme 4 (ligne 3) contre $O(k^2)$ dans la version originale.

3.3 Propagation orientée variables

Dans l'algorithme AC-V (algorithme 2), la file de propagation Q contient la liste des variables modifiées, qui nécessitent une révision des contraintes les

Algorithme 3: AC-C($\mathcal{N}, Q, modif$) =

```
1 si Q = ∅ alors retourner N
2 sinon
3     Choisir C dans Q, Q' ← Q − C
4     (N', Δ) ← revise(C, modif(C))
5     si Δ = ⊥ alors retourner ⊥
6     sinon
7         updateModif(modif, Δ, C)
8         Q' ← Q' ∪ {C' | vars(C') ∩ Δ ≠ ∅}
9         retourner AC-C(N', Q', modif)
```

impliquant. Q et *modif* doivent être initialisés conformément à l'état connu de $\mathcal{N}$ (par exemple, au sein de `MAC` on pourra initialiser Q et *modif* avec les variables impliquées par les hypothèses δ).

À chaque itération de l'algorithme, une variable est sélectionnée (ligne 3), et toutes les contraintes impliquant cette variable sont traitées (boucle de la ligne 4). Notez que plusieurs variables impliquées par une même contrainte peuvent se trouver simultanément dans la file. Le test $modif(C) \neq \emptyset$ permet d'éviter de propager inutilement à chaque fois les mêmes contraintes.

La ligne 5 fait appel au propagateur spécifique de C. Le propagateur renvoie le réseau filtré ainsi que l'ensemble $\Delta \subseteq \text{vars}(C)$ des variables modifiées, ou $\perp$ si une inconsistance a été détectée (par exemple, le domaine d'une variable a été vidé).

3.4 Propagation orientée contraintes

Dans cette version, appelée AC-C (algorithme 3), ce sont les contraintes à réviser qui sont stockées dans la file. L'algorithme obtenu est légèrement plus simple, puisque l'on n'a pas à gérer le cas des multiples révisions d'une même contrainte. La structure *modif* sera exploitée uniquement par les propagateurs.

3.5 Le propagateur général `revise`$^{\text{rm}}$

Pour illustrer l'utilisation du système de propagation spécialisable, nous montrons un exemple de propagateur nommé `revise`$^{\text{rm}}$. Il s'agit d'un propagateur issu de l'algorithme AC-3$^{\text{rm}}$ [15]. À l'image de GAC-schema, nous avons abstrait la notion de recherche de support (fonction abstraite `findSupport`). L'idée est de rechercher un support pour chaque valeur, et d'enregistrer ce support dans la structure *res*. Ainsi, la prochaine fois que l'on cherchera un support pour la même valeur, on commence par vérifier si le support enregistré (nommé *résidu*) est toujours valide.

Notez l'exploitation de la structure *modif* à la ligne 3 : si une seule variable a été modifiée, les au-

Algorithme 4: revise$^{\text{rm}}(C, \mathit{modif}) =$

```
1  N' ← N
2  Δ ← ∅
3  pour chaque X ∈ vars(C) | modif ≠ {X},
     v ∈ dom(X) | res(C, X, v) non valide faire
4      τ ← findSupport(C, X, v)
5      si τ = ⊥ alors
6          Supprimer v de dom(X) dans N'
7          si dom(X) = ∅ alors retourner ⊥
8          Δ ← Δ ∪ {X}
9      sinon pour chaque Y ∈ vars(C) faire
10         res(C, Y, τ(Y)) ← τ
11 retourner (N', Δ)
```

ters valeurs de la variable n'ont pas pu perdre leurs supports et il est inutile de les parcourir.

4 Gestion de la file de propagation

Note : Toutes les heuristiques présentées dans cet article s'entendent « valeur minimale d'abord ».

4.1 Travaux connexes sur les heuristiques d'ordonnancement des révisions

L'ordre dans lequel les contraintes sont révisées a un impact non négligeable sur les performances de la propagation, et plusieurs travaux ont été consacrés à étudier un « bon » ordre de propagation de ces contraintes. Le premier article sur le sujet est de Wallace & Freuder (1992 [27]). Leur étude compare l'impact de différentes heuristiques d'ordonnancement sur un algorithme de propagation AC-3 orienté arcs sur des CSP binaires. Les heuristiques conçues par Wallace & Freuder suivent le principe suivant : pour une propagation efficace, il faut supprimer des valeurs le plus tôt possible. Cependant, il est très difficile de prévoir si une contrainte est susceptible de supprimer des valeurs avant sa propagation.

Wallace & Freuder utilisent la dureté (proportion d'instanciations des domaines initiaux des variables interdites par la contrainte) pour estimer la force de celle-ci. Calculer la dureté d'une contrainte reste #P-difficile, et la dureté de celle-ci peut fortement évoluer au cours de la recherche, lorsque des valeurs sont supprimées des domaines des variables. Cette heuristique est donc inapplicable dynamiquement ou dans un cadre non-binaire. D'autres heuristiques moins complexes sont proposées : la plus efficace évalue uniquement la taille du domaine de la variable dans l'arc. Cependant, même sur de petits CSP binaires, les meilleurs résultats de Wallace & Freuder sont obtenus sur des heuristiques dites statiques, calculées une seule fois au début de la recherche. Une alternative intéressante, plus récemment proposée par Balafoutis & Stergiou (2010 [1]), est d'exploiter les poids des contraintes calculés par l'heuristique d'instanciation $^{dom}/_{wdeg}$ [7] pour estimer les contraintes les plus fortes sans surcout. Cette heuristique n'est évaluée que sur des CSP binaires homogènes.

Boussemart *et al.* ont étudié et évalué (2004 [8]) différentes heuristiques d'ordonnancement des révisions en utilisant des algorithmes de propagation orientés arcs, variables ou contraintes. Comme dans les travaux précédemment cités, ces évaluations se restreignent à des CSP binaires et homogènes. Le principal résultat de cette étude est que, dans ce contexte, la variante la plus efficace est un algorithme de propagation orienté variables, en propageant d'abord les modifications des variables dont le domaine est le plus petit (nous appelons cette heuristique *dom*). Ce résultat est très proche des conclusions de Wallace & Freuder (à ceci près que Wallace & Freuder considèrent la taille du domaine de la variable *à réviser*). On note qu'une propagation orientée contraintes (ou arcs), associée à une heuristique consistant à réviser en premier la contrainte dont le produit de la taille des domaine des variables impliquées est le plus petit (que nous appelons Πdom) parvient à limiter le nombre de révisions par rapport à l'approche orientée variables, mais le surcout représenté par le calcul de l'heuristique est trop important. Cependant, les structures de données employées par Boussemart *et al.* (cf ci-après) peuvent être considérablement améliorées.

Le solveur Gecode utilise une méthode de propagation relativement sophistiquée [22]. Ici, la stratégie diffère de celle proposée initialement par Wallace & Freuder : comme on ne peut pas facilement prévoir si une contrainte va filtrer des valeurs ou non, on cherche à minimiser le temps perdu en propageant les contraintes les plus rapides en premier. La technique utilisée ici ne peut être utilisée qu'avec une propagation orientée contraintes ou arcs. Un entier est associé à chaque contrainte : 1 pour des propagateurs à temps constant, 2 à 4 pour des propagateurs en $O(kd)$ (suivant la valeur de k), 5 et 6, respectivement, pour des propagateurs en $O(kd^2)$ et $O(kd^3)$, ou 7 pour les propagateurs plus lents [1]. L'entier est amené à évoluer en fonction de l'état de la recherche : par exemple, une contrainte ternaire dont une variable a été instanciée peut être considérée comme une contrainte binaire. Chaque entier identifie une file de propagation de type

1. En réalité, les nombres utilisés par Gecode vont de 7 pour les propagateurs les plus rapides à 1 pour les plus lents ; la notation est inversée ici pour correspondre au comportement des autres heuristiques.

Structure	Insert	M. à j.	Extr. min
Bit vector	$O(1)$	$O(1)$	$\Theta(\lambda)$
Binary heap	$O(\log \lambda)$	$O(\log \lambda)$	$O(\log \lambda)$
Binomial h. [26]	$O(1)$*	$O(\log \lambda)$	$O(\log \lambda)$
Fibonacci h. [9]	$O(1)$	$O(1)$*	$O(\log \lambda)$*

TABLE 1 – Structures de données pour files de priorité. λ est le nombre d'éléments présents dans la structure, * indique une complexité amortie.

FIFO ; celles-ci sont traitées par ordre de priorité (voir section suivante). Cette approche est moins fine que les précédentes, mais minimise la surcharge représentée par le calcul des heuristiques. Notons qu'il n'est plus possible de donner une priorité différente à deux contraintes de même type, même si l'une porte sur des domaines bien plus grands que l'autre. Gecode permet cependant de définir manuellement l'ordre de propagation de certaines contraintes pour gérer au mieux certains cas spécifiques [13]. Les solveurs Choco [11] et Minion [10] proposent une stratégie hybride : une file (ou pile) de variables d'abord, tout en reportant la propagation de certaines contraintes plus lentes (identifiées de manière statique, correspondant au niveau 3 ou 4 et plus de Gecode) dans une file à part.

On peut finalement constater que les heuristiques les plus efficaces d'après les articles de Wallace & Freuder ou Boussemart *et al.* (dom et Πdom) se révèlent également suivre le pricipe du « propagateur le plus rapide d'abord » : les algorithmes de propagation utilisés, extraits d'AC-3 pour des contraintes binaires, sont en $O(d^2)$. Choisir les domaines les plus petits en priorité aura donc tendance à sélectionner les propagateurs les plus rapides. Dans le cas de propagateurs hétérogènes, la situation est plus compliquée.

4.2 Structures de données pour les files de priorité

Que l'on utilise une méthode de propagation orientée variables ou contraintes, utiliser une heuristique d'ordonnancement dynamique des révisions nécessite l'utilisation de *files de priorité*. Ces structures supportent trois opérations : insertion d'un élément dans la structure, extraction du plus petit élément, et mise à jour de la valeur d'un élément. Il s'agit d'une problématique abondamment étudiée dans la littérature algorithmique et disposant de centaines d'applications. Le plus souvent, les files de priorité sont des *tas* (*heaps*), c'est-à-dire des structures de données arborescentes équilibrées, construites de sorte que la valeur d'un nœud de l'arbre soit toujours inférieure à celles de ses fils. Les arbres entrainent une complexité logarithmique pour la plupart des opérations.

Il existe de nombreuses variantes de tas, généralement capables d'obtenir des complexités amorties pratiquement constantes. La table 1 montre différentes structures que nous avons implanté et expérimenté dans le cadre de cette étude. La structure de données « Bit vector » est proche de celle utilisé par Boussemart *et al.* Les complexités sont indicatives : une structure sophistiquée entraine souvent un facteur constant plus important qu'une structure plus simple, et les complexités amorties cachent parfois des comportements pathologiques. La section expérimentale de cet article contient une étude empirique du comportement de ces structures de données pour la résolution de CSP.

Une autre approche possible pour la gestion des files de priorité est celle retenue par les développeurs de Gecode : plutôt que d'utiliser un tas, on restreint le nombre de priorités possibles à m valeurs (7 pour Gecode). La file de priorité est divisée en m files traitées successivement. La complexité pour ajouter ou supprimer un élément est alors constante, et la complexité pour extraire le plus petit élément est de $O(m)$. Des pointeurs sont maintenus sur la première et la dernière file non vides, afin d'éviter de parcourir inutilement certaines files inutilisées.

Il est finalement possible d'utiliser cette structure de données pour appliquer une approximation d'une heuristique quelconque : le logarithme de la fonction d'évaluation peut être utilisé pour choisir une des m files de priorité (logarithme à base 16 pour passer d'un nombre codé sur 31 bits à 3 bits). Ce logarithme peut être calculé efficacement en observant la position du bit le plus fort dans la représentation binaire d'un nombre. Ces approches seront comparées aux tas dans la section expérimentale de cet article.

4.3 Les faiblesses de la propagation orientée variables

Soit $\mathcal{N}$ un CN constitué de n variables X_1 à X_n avec $\text{dom}(X_i) = \{1, \ldots, n\}$. Les contraintes sont $X_i \leq X_{i+1}$ et alldifferent$(X_1, \ldots, X_n)$. Le CSP correspondant à $\mathcal{N}$ n'admet qu'une seule solution : $(X_1, \ldots, X_n) = (1, \ldots, n)$. Bien que le problème paraisse trivial (en instanciant $X_1 = 1$, on obtient immédiatement la solution par propagation), une stratégie de résolution classique (en utilisant par exemple l'heuristique d'instanciation $^{dom}/_{ddeg}$) aura peu de chance d'effectuer cette affectation correcte avant de nombreuses réfutations. Pire, une propagation orientée variables adopte un comportement pathologique : après une réfutation, la valeur supprimée est propagée, variable par variable, grâce aux contraintes $\leq$. Cependant, en utilisant une propagation orientée variable, la contrainte alldifferent est inutilement appelée après chaque modification de variable ! En utilisant une propagation orientée con-

Contrainte	Algo. de propag.	Complexité	Évaluateur
$X\{<,\leq\}Y$	Arith. intervalles	$O(1)$	2
$\pm X + c = Y$ (intervalles)	Arith. intervalles	$O(1)$	3
$X = Y\{+,\lvert-\rvert\}Z$ (intervalles)	Arith. intervalles	$O(1)$	4
$\bigvee(\dots)$	*Watched literals*	$O(k)$	$\log_2 \lvert C\rvert$
alldifferent$(\dots)$	L.-O. *et al.* [17]	$O(k \log k)$	$\lvert C\rvert \log_2 \lvert C\rvert$
$X \neq Y$	Algo. bit-à-bit	$O(d)$	$\min(\lvert X\rvert, \lvert Y\rvert)$
$\pm X + c = Y$	Algo. naïf	$O(d)$	$\lvert X\rvert + \lvert Y\rvert$
$X = Y\{+,\times,\lvert-\rvert\}Z$	revise$^{\text{rm}}$ [15]	$O(d^2)$	$\lvert X\rvert \lvert Y\rvert + \lvert X\rvert \lvert Z\rvert + \lvert Y\rvert \lvert Z\rvert$
relations binaires	revise$^{\text{bit+rm}}$ [16]	$O(d^2)$	$\lvert X\rvert \lvert Y\rvert /3$
liste de λ supports	STR2 [14]	$O(k\lambda)$	$\lambda \lvert C\rvert$
liste conflits	revise$^{\text{rm}}$	$O(kd^k)$	$\lvert C\rvert \prod_{X\in\text{vars}(C)} \lvert X\rvert$
$X \iff C(\dots)$	Algo. naïf	Complexité $C + \neg C$	$eval(C) + eval(\neg C)$

TABLE 2 – Propagateurs et évaluateurs des contraintes de CSP4J. $|C|$ représente l'arité de C, $|X|$ est $|\text{dom}(X)|$. Certains propagateurs sélectionnent dynamiquement un algorithme de propagation basé sur l'arithmétique des intervalles quand le domaine des variables s'y prête. Certains propagateurs incluent une détection limitée de contrainte *entailed*.

traintes et en donnant la priorité aux contraintes $\leq$, plus rapides à propager et plus fortes, le propagateur de *alldifferent* n'est appelé qu'une seule fois, ce qui entraine une propagation bien plus rapide. Pour donner une idée de l'impact de la méthode de propagation, notre solveur CSP4J sur notre machine de test (cf section expérimentale ci-après) propage la réfutation $X_2 \neq 1$ en environ 6,5 s pour $n = 2\,000$ avec AC-V contre 0,1 s avec AC-C. Ce problème sera appelé *bigleq-n* dans la section expérimentale de l'article.

Un autre exemple classique est le problème des *n-dames*, modélisé sous la forme des trois contraintes alldifferent$(X_1, \dots, X_n)$, alldifferent$(X_1 - 1, \dots, X_n - n)$ et alldifferent$(X_1 + 1, \dots, X_n + n)$. L'implantation de ce modèle nécessite l'introduction de variables auxiliaires pour représenter les variables $X_i \pm i$, reliées aux variables principales par des contraintes d'égalité (à une constante près). Ces contraintes devraient être propagées d'abord. Ainsi, MAC-$^{dom}/_{ddeg}$ avec AC-V trouve la première solution au problème des 40-dames en 86 s, contre 41 s pour AC-C.

4.4 Heuristiques d'ordonnancement des révisions fines et hétérogènes

Pour déterminer la priorité des contraintes, nous définissons une heuristique comme suit : un *évaluateur* dynamique est associé à chaque contrainte. Un évaluateur est une fonction qui renvoie un nombre entier positif défini sur 31 bits (0 à $2 \cdot 10^9$ environ). L'objectif de cet évaluateur est de pouvoir émuler à la fois les heuristiques proposées par Wallace & Freuder, Boussemart *et al.* ou Balafoutis & Stergiou (dans ce cas, l'évaluateur sera le même pour chaque contrainte), ainsi que de définir une priorité contrainte par contrainte, à la manière de Gecode, mais de manière plus fine et plus souple. Nous avons défini une série d'évaluateurs pour les différentes contraintes implantées dans notre solveur CSP4J ; ces évaluateurs définissent la priorité des contraintes en fonction de leur temps estimé de propagation. La table 2 répertorie ces évaluateurs. Dans la suite de cet article, nous appellerons *eval* l'heuristique d'ordonnancement des révisions utilisant les évaluateurs définis dans cette table.

On peut finalement combiner n'importe quelle heuristique avec le principe développé par Balafoutis & Stergiou [1] : en divisant le résultat donné par l'évaluateur par le poids de la contrainte, on obtient alors les heuristiques $^{eval}/_{w}$ et $^{\Pi dom}/_{w}$.

5 Expérimentations

Ces expérimentations sont réalisées à l'aide de notre solveur CSP4J [25]. Celui-ci a été exécuté sur une machine virtuelle Java OpenJDK IcedTea7 2.1 Server 64 bits pour Linux, fonctionnant sur un processeur Intel Core 2 Quad Q6700. L'heuristique d'instanciation utilisée est $^{dom}/_{ddeg}$ avec résolution pseudo-aléatoire des égalités (la graine est fixe) et redémarrages à progression géométrique. L'heuristique $^{dom}/_{wdeg}$ serait bien plus performante, cependant elle interagit avec l'heuristique de révision et les résultats ne seraient pas comparables. Avec la stratégie de résolution choisie, l'arbre de recherche parcouru par `MAC` est rigoureusement le même quelle que soit la méthode de propagation utilisée. Les différents algorithmes de révision utilisés sont ceux indiqués sur la table 2. Nous avons sélectionné une série d'instances de CSP représenta-

			AC-V/*dom*				AC-C/*eval*			
	n	e	Bit v	B$^{\text{ary}}$ h	B$^{\text{ial}}$ h	Fib. h	Bit v	B$^{\text{ary}}$ h	B$^{\text{ial}}$ h	Fib. h
bigleq-400	400	400	731,5	666,8	702,3	712,4	**19,7**	**18,9**	**18,4**	22,0
bqwh-18-141-0-ext	141	879	37,4	37,4	37,6	38,7	42,5	**31,5**	**33,4**	37,3
bqwh-18-141-47-glb	141	36	**93,9**	**90,3**	**90,9**	**93,3**	**88,0**	**85,6**	**85,4**	**87,1**
crossword-lex-vg5-7	35	12	123,9	124,8	123,9	125,1	**84,3**	**82,9**	**81,3**	**82,9**
frb40-19-2-ext	40	410	45,0	45,0	45,1	45,5	53,4	**35,7**	**37,3**	45,0
js-e0ddr1	1 480	1 205	**305,2**	**280,3**	**289,2**	**304,8**	730,1	368,5	358,7	434,0
langford-3-13	65	27	101,5	104,4	101,4	102,1	**41,3**	**42,6**	**41,9**	**42,1**
lemma-24-3	552	924	70,4	70,1	69,9	72,0	65,1	**58,5**	**60,4**	**62,6**
os-taillard-5-100-4	625	500	**489,9**	**473,3**	**459,5**	**491,7**	701,3	**452,4**	**449,5**	**493,2**
queens-40	120	83	88,9	84,7	83,2	86,7	42,6	**38,2**	**38,9**	**38,7**
ruler-44-9-a3	45	38	17,3	17,6	18,4	17,7	**7,7**	**7,3**	**7,3**	**7,9**
scen11	8 546	7 866	662,7	653,3	657,8	661,1	**616,4**	**576,2**	**587,8**	**590,5**
series-20	77	40	14,8	14,5	15,1	14,9	**9,0**	**9,2**	**9,1**	**9,5**
tsp-20-193-ext	61	230	106,1	112,8	110,0	109,8	**92,4**	**89,9**	**89,3**	**97,0**

TABLE 3 – Performances des différentes files de priorité. Temps de résolution en secondes avec l'heuristique d'instanciation $^{dom}/_{ddeg}$. Les meilleurs temps (dans une marge de 10 %) sont surlignés.

tives, utilisées lors de la compétition internationale de solveurs CPAI'08 [23]. Les instances sont choisies pour être de difficulté moyenne (pouvant être résolue en 10 à 1 000 secondes).

Une première série d'expérimentations, dont les résultats sont indiqués dans la table 3, concerne le choix de la structure de données pour les files de priorité. Les résultats mettent en avant les tas binaires et binomiaux, sans qu'il soit réellement possible de les discriminer, que ce soit pour une propagation orientée variables ou contraintes.

Finalement, la table 4 montre une comparaison des différentes heuristiques de révision. Pour les heuristiques $^{dom}/_{wdeg}$, $^{\Pi dom}/_{w}$ et $^{eval}/_{w}$, on incrémente le poids d'une contrainte quand elle détecte une inconsistance au cours de la propagation, bien que l'heuristique d'instanciation $^{dom}/_{ddeg}$ soit utilisée. En plus des heuristiques présentées précédemment, nous expérimentons de simples files FIFO (réalisées à l'aide de deux listes simplement chainées) pour les variables et les contraintes. La colonne AC-C/8 FIFO/Arity correspond au fonctionnement du solveur Gecode (avec les évaluateurs simplifiés basés sur l'arité de la complexité du propagateur). La dernière colonne est l'hybride entre l'heuristique *eval* et la structure de données composée de 8 files FIFO décrite à la fin de la section 4.2. C'est cette dernière version qui apparait comme étant la plus robuste : elle permet à la fois de discriminer des contraintes de manière plus fine que l'approche Gecode, ce qui permet un gain sensible sur certains problèmes (par exemple, quand ils utilisent des contraintes en extension comme les *frb*, *bqwh-...-ext*, *crossword*), et à la fois de présenter un surcoût minimal pour les problèmes pour lesquels l'ordre des révisions n'a que peu d'impact (par exemple, les problèmes dont les domaines sont toujours des intervalles, comme les problème *js-...* ou *os-...*). En tout état de cause, les propagations orientées variables ne sont jamais parmi les plus performantes.

Les heuristiques prenant en compte le poids des contraintes semblent également peu performantes, mais il convient de rappeler que celles-ci avaient été conçues pour interagir positivement avec l'heuristique d'instanciation $^{dom}/_{wdeg}$, qui n'a pas été utilisée ici.

6 Conclusion & perspectives

Dans cet article, nous avons décrit AC-V et AC-C, deux algorithmes de propagation de contraintes à gros grain spécialisables et utilisant respectivement une file de propagation orientée variables et contraintes. Nous avons rappelé en quoi la gestion de la file de propagation est importante lors de la conception d'un algorithme de propagation, nous avons évalué plusieurs structures de données pouvant être utilisées pour ce faire. Malgré des performances correctes pour les *tas binaires* ou *binomiaux*, la meilleure alternative pour une majorité des problèmes testés s'est révélée être l'utilisation de plusieurs FIFO de priorité croissante.

Après avoir montré comment une propagation orientée variables peut entrainer un comportement pathologique, nous avons proposé une nouvelle manière de contrôler l'ordre de révision dans une approche orientée contraintes. Nous avons montré expérimentalement que même si le comportement pathologique n'est

	AC-V	AC-V/B$^{\text{ary}}$ h		AC-C	AC-C/Binomial heap				AC-C/8 FIFO	
	FIFO	*dom*	$\frac{dom}{wdeg}$	FIFO	Πdom	$\Pi dom/w$	*eval*	$eval/w$	Arity	log *eval*
bigleq-400	651,0	666,8	667,4	643,3	58,5	231,3	**18,4**	222,3	**18,6**	**18,9**
bqwh-18-141-0-ext	41,8	37,4	43,8	36,0	31,7	52,4	33,4	53,6	35,3	**25,6**
bqwh-18-141-47-glb	94,8	90,3	93,2	**82,7**	**80,0**	**87,7**	**85,4**	89,7	**87,0**	**82,6**
crossword-lex-vg5-7	140,4	124,8	148,6	**87,5**	93,6	90,7	**81,3**	93,2	91,9	**80,9**
frb40-19-2-ext	57,5	45,0	60,6	41,7	**39,0**	85,2	**37,3**	87,8	41,6	**36,9**
js-e0ddr1	311,1	280,3	314,0	**189,0**	228,4	**195,7**	358,7	240,3	275,5	**187,0**
langford-3-13	109,9	104,4	117,2	45,5	**44,4**	45,0	**41,9**	74,3	**41,5**	**40,6**
lemma-24-3	68,9	70,1	69,7	**60,2**	**61,1**	**60,9**	**60,4**	**59,9**	**59,5**	**60,1**
os-taillard-5-100-4	504,6	473,3	509,1	**302,3**	337,0	**312,0**	449,5	408,9	357,0	**310,3**
queens-40	77,0	84,7	97,0	45,0	44,8	43,5	**38,9**	58,7	**35,6**	**36,3**
ruler-44-9-a3	21,0	17,6	21,1	8,0	8,1	8,4	7,3	12,0	**6,1**	**6,7**
scen11	811,0	653,3	711,8	767,1	**597,1**	**578,2**	**587,8**	**594,6**	779,6	**588,1**
series-20	15,3	14,5	16,4	**9,9**	10,4	10,8	**9,1**	11,2	**9,6**	**9,0**
tsp-20-193-ext	126,6	112,8	131,3	**82,0**	**83,6**	**86,1**	89,3	103,6	**79,5**	**84,5**

TABLE 4 – Temps de résolution (en secondes) avec l'heuristique d'instanciation $dom/ddeg$. Les meilleurs temps (dans une marge de 10 %) sont surlignés.

pas rencontré, une propagation orientée contraintes est meilleure dans une grande majorité des cas qu'une propagation orientée variables. La méthode que nous avons proposée est compétitive avec les approches retenues par les meilleurs solveurs disponibles, tout en étant plus évolutive : il est très probable qu'une heuristique capable de prendre en compte la probabilité qu'a une contrainte de supprimer des valeurs sera des plus intéressantes. Notre méthode permet de proposer de nouveaux évaluateurs basés sur cet aspect de la propagation. Ils restent encore à concevoir et à évaluer.

Références

[1] T. Balafoutis and K. Stergiou. Evaluating and improving modern variable and revision ordering strategies in CSPs. *Fundam. Inform.*, 102(3–4) :229–261, 2010.

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

[3] N. Beldiceanu and E. Contejean. Introducing Global Constraints in CHIP. *Mathl. Comput. Modelling*, 20(12) :97–123, 1994.

[4] C. Bessière, G. Katsirelos, N. Narodytska, and T. Walsh. Circuit complexity and decompositions of global constraints. In *Proc. IJCAI'09*, pages 412–418, 2009.

[5] C. Bessière and J.-C. Régin. AC for General Constraint Networks : Preliminary Results. In *Proc. IJCAI'97*, pages 398–404, 1997.

[6] C. Bessière, J.-C. Régin, R.H.C. Yap, and Y. Zhang. An Optimal Coarse-Grained Arc Consistency Algorithm. *Artificial Intelligence*, 165(2) :165–185, 2005.

[7] F. Boussemart, F. Hemery, C. Lecoutre, and L. Saïs. Boosting systematic search by weighting constraints. In *Proc. of ECAI'04*, pages 146–150, 2004.

[8] F. Boussemart, F. Hemery, and C. Lecoutre. Revision ordering heuristics for the CSP. In *Proc. CPAI'04 workshop held with CP'04*, pages 29–43, 2004.

[9] M.L. Fredman and R.E. Tarjan. Fibonacci Heaps and Their Uses in Improved Network Optimization Algorithms. *Journal of the ACM*, 34(3) :596–615, 1987.

[10] I.P. Gent, C. Jefferson, and I. Miguel. Minion : A Fast, Scalable, Constraint Solver. In *Proceedings of ECAI'06*, pages 98–102, 2006.

[11] F. Laburthe et al. CHOCO : Implementing a CP kernel. In *Proc. of the TRICS'2000 workshop held with CP'2000*, pages 71–85, 2000.

[12] M. Z. Lagerkvist and C. Schulte. Advisors for incremental propagation. In *Proc. of CP'07*, pages 409–422, 2007.

[13] M.K. Lagerkvist and C. Schulte. Propagator groups. In *Proc. of CP'2009*, pages 524–538, 2000.

[14] C. Lecoutre. STR2 : Optimized simple tabular reduction for table constraints. *Constraints*, 16(4) :341–371, 2011.

[15] C. Lecoutre and F. Hemery. A Study of Residual Supports in Arc Consistency. In *Proceedings of IJCAI'2007*, pages 125–130, 2007.

[16] C. Lecoutre and J. Vion. Enforcing Arc Consistency using Bitwise Operations. *Constraint Programming Letters*, 2 :21–35, 2008.

[17] A. López-Ortiz, C.-G. Quimper, J. Tromp, and P. van Beek. A fast and simple algorithm for bounds consistency of the alldifferent constraint. In *Proc. IJCAI'03*, pages 245–250, 2003.

[18] A.K. Mackworth. Consistency in Networks of Relations. *Artificial Intelligence*, 8(1) :99–118, 1977.

[19] J.J. McGregor. Relational Consistency Algorithms and their Application in Finding Subgraph and Graph Isomorphisms. *Information Sciences*, 19 :229–250, 1979.

[20] S. Merchez, C. Lecoutre, and F. Boussemart. Abscon : a prototype to solve CSPs with abstraction. In *Proceedings of CP'01*, pages 730–744, 2001.

[21] C. Schulte et al. Generic Constraint Development Environment (Gecode). `http://www.gecode.org/`, 2005-2010.

[22] C. Schulte and P.J. Stuckey. Efficient Constraint Propagation Engines. *ACM Transactions on Programming Languages and Systems*, 31(1) :1–43, 2008.

[23] M. van Dongen, C. Lecoutre, and O. Roussel. Third International CSP Solvers Competition. `http://www.cril.univ-artois.fr/CPAI08`, 2008.

[24] P. van Hentenryck, Y. Deville, and CM. Teng. A Generic AC Algorithm and its Specializations. *Artificial Intelligence*, 57 :291–321, 1992.

[25] J. Vion. Constraint Satisfaction Problem for Java. `http://cspfj.sourceforge.net/`, 2006–2012.

[26] J. Vuillemin. A Data Structure for Manipulating Priority Queues. *Communications of the ACM*, 21 :309–314, 1978.

[27] R.J. Wallace and E.C. Freuder. Ordering Heuristics for Arc Consistency Algorithms. In *Proceedings of NCCAI'92*, pages 163–169, 1992.